Scientific Programming 17 (2009) 279–282
DOI 10.3233/SPR-2009-0285
IOS Press

Book Review

Dan Nagle
George Mason University, Computational and Data Sciences, 4311-G Bob Ct., Fairfax, VA 22030, USA
E-mail: dnagle@gmu.edu

Python for Software Design, by Allen B. Downey, Cambridge University Press, Cambridge, 2009, ISBN-13:9780521725965

Python for Software Design is subtitled *How to Think Like a Computer Scientist*. It has a Preface, 19 chapters, an Appendix and 251 pages, including the index. The author wants to teach computer science to first-year students. So the approach is to use a larger number of shorter chapters to keep the lessons to a size and complexity suitable for freshmen. After some experience with Java, the author switched to Python as the teaching language.

I'm in computational science not computer science, and I've been programming for decades. What does this book have for me? Do I want to think like a computer scientist? Do all computer scientists think alike? Oh, well, I've heard of Python being used for computational scientific tasks (Jonathan E. Guyer et al., FiPy: Partial differential equations with Python, *CiSE* **11**(3), 6–15; M. Tobis, PyNSol: Objects as scaffolding, *CiSE* **7**(4), 84–91; K.-A. Mardal et al., Using Python to solve partial differential equations, *CiSE* **9**(3), 48–51), so let's keep an open mind and see what's here.

Starting with the Preface, we get a little of the history of the book, including the original courses the author taught using Java. The decisions to make the course simpler, and the chapters shorter, is explained as a response to the classroom response to the previously used material, and some suggestions from some high school teachers who were also using the material. The goals of making careful definitions of terms, building gradually, and keeping focus on the programming rather than on the language are mentioned. The results of these goals leading to Python, and the conversion of the teaching material to Python are described. I like the list of contributors. If you found a bug in an earlier version of the book, you got your name in print (which obviously encourages bug reporting). I won't mention it for every chapter, but each chapter ends with a **Debugging** section, a **Glossary** section, and a **Problems** section. The first two are helpful summaries, and placing them near the problems likely helps the students as well. There are also a few problems throughout the chapters as well.

Chapter 1 introduces us to Python, with an easy explanation of high-level languages versus low-level languages. Low-level languages are defined to be machine languages, and the advantages of using high-level languages are explained. This includes an explanation of the differences between interpreters and compilers. Now, we're ready to download and start using the Python interpreter, with some `1+1` giving `2` examples. What is a program? It's defined as input, output, math, conditional execution, and repetition. Then, we move to debugging. Here, classes of errors are described as syntax errors, runtime errors, and semantic errors. We get a description of formal versus natural languages before writing our "`Hello, world!`" program. From here, I read the book with the IDLE window (a Python shell, it downloaded and installed with the rest of the Python for Mac software from python.org) open on my Mac. Now that I'm writing this review, I've still got IDLE beside the text window.

Chapter 2 introduces the ideas of values, expressions, and statements. First, we distinguish between the value and the type of a name, and use the `type()` function to make tests. Next, we'll look at variables. Python variables need no declaration (usually). One must distinguish variables from keywords. The usual naming rules apply. Finally, we see how Python forms expressions, and quickly learn the difference between integer division and floating point division. We need to know Python's order of precedence, and legal and illegal operations, depending upon the types of the operands. Here we find comments. The author seems adept at choosing clear names so commentary may be kept at a minimum.

Chapter 3 introduces us to the idea of Python functions. First we find the type conversion functions, building on the material from Chapter 2. Next, we'll look at the math functions, and that means we'll need to see how to import the math module and how to use the dot notation to specify that the name is to be taken from the math module. We'll see how to combine functions, and how to write our own. This brings us to the subject of flow of execution. We'll also need to know that the scope of parameters and variables. To help the student understand all this, stack diagrams are used. Lastly, fruitful and void functions (functions that do or do not return values) are distinguished.

Chapter 4 examines interface design, using the turtleworld modules. Unfortunately, I was unable to download the Swampy software, the thinkpython.com site doesn't seem to be around anymore. After a while, I tracked the software to a subversion repository, but it seemed to want me to register in order to login, so I gave up regretfully. At any rate, we'll start this chapter by executing some functions that move turtles around the screen (turtles drag a paintbrush behind them). Next, function calling is combined with control structures to make the turtles follow complete paths. Now, we can examine encapsulation and generalization to make our functions easier to use. And so armed, we can start to think about interface design. This leads to the ideas of refactoring our work so far, leading, in turn, to the idea of a development plan. The first idea of a development plan is to make something work simply, then embellish it gradually. Since we may now revisit old code, the notion of `docstrings` is introduced.

Chapter 5 examines conditionals and recursion. First, we must understand boolean expressions, and move to logical operators. Then we can learn conditional execution, with and without an alternative path (that is, the `if` statement with and without an `else` clause). Else-if clauses are expressed by the `elif` keyword in chained conditionals. An alternative to chained conditionals is nested conditionals. Now we're ready to examine recursion. So we find a simple countdown example of recursion, and analyze it into its recursive case and its base case. An infinite recursion is shown to reveal the error message it generates. Lastly, we see how to get simple input from the keyboard.

Chapter 6 discusses fruitful functions. All the user-written functions so far have had only side-effects, usually printing something. Now, we'll start writing some to return values. Since we've already seen print functions, we can learn incremental development by putting print functions inside our own functions, to check arguments and partial results. The idea is to start with something simple, and make small changes, so when something goes wrong, you'll have a very good idea where. Now that we have functions, we can have one function of our own call another of our own, even itself. So equipped, we can tackle the factorial and the Fibonacci favorites. We can place guardian statements in our functions to check our assumptions.

Chapter 7 discusses iteration. First, we must understand that one may assign a value to a variable more than once, and that the new value may be computed from the old value (but only if the variable already has a valid old value). Next, recast the countdown problem as iteration rather than recursion. Now we confront a dilemma: How do we know whether our condition for exiting the loop will ever be met? The answer, in general, is, of course, we don't. A simple example suffices. What if we want to exit the `while` loop early? Use the `break` statement. When dealing with floating point data it is difficult to test for exactness, so the idea of two numbers being close enough must be used. Now that we've seen Newton's method compute a square root, we've seen an example of an algorithm.

Chapter 8 brings us to strings, which we've been using since the beginning without much understanding of their properties and limitations. Strings appear similar to an array of characters, they have a length, and characters can be selected with an array-like notation. An interesting point is that when a negative index is applied to a string, one is merely indexing the string from the end rather than from the beginning. There's a plus operator to concatenate strings, and a slice notation to extract substrings. New strings may be made, but existing strings are immutable. Various string methods are shown, and along the way, the difference between methods and functions is introduced.

Chapter 9 describes ways to search for strings embedded in other strings, and along the way we learn some file operations. Several ways to search are described, often through the whole file, a line at a time. We also learn where to get the file of all valid English words for Scrabble and cross-word puzzles!

Chapter 10 introduces us to lists. A list is a sequence of values where the values can be of any type (including another list, which is said to be nested). Unlike a string, a list is mutable. So, while a list item is addressed with the same notation as a character in a string, this may appear on the left-hand side. Various list operations are introduced, including concatenation and slices. Then we meet the list methods. We learn the design patterns of maps, filters, and reductions. Some ways of removing an item from a list are shown. Now comes the subtlety of aliasing. Since strings are immutable, only one is created within the interpreter for each unique string. With lists, two names assigned identical lists are not aliased. However, if one list is assigned to another, they are aliased. The consequences of this for argument passing are examined.

Chapter 11 introduces dictionaries. With a list, the items could be of any type. A dictionary is also made of a set of items, but here the keys may also be of almost any type (the keys of the list are the integers starting with zero). The keys are hashed to reduce search times. How to make a histogram from a dictionary is shown. One's attention is directed to the unpredictable order of the dictionary items. Reverse lookup is explained. Lists and dictionaries are compared, which leads to a discussion of hashes. The dictionaries are used to create memos so previously computed values of the recursive Fibonacci calculation need not be recomputed. This leads to a discussion of global variables. We conclude the chapter with a discussion of long integers. Trust me, it's all related!

Chapter 12 discusses tuples. The selection operator selects an item from a tuple, just like it does for strings, lists, and dictionaries. Slices and ranges work as before. Tuple assignment, and how to use that to swap tuples, is shown. Use of tuples as return values (for example, from `divmod()`) is shown. Use of tuples to simplify argument lists, how to make variable-length argument lists, and how to unpack them is shown. Tuples are compared with lists, and with dictionaries.

Chapter 13 discusses selection of data structures, now that we have several from which to chose. A number of cases involving analysis of long strings to search for patterns of words is shown, and the trade-offs for the various data structures is explored. Random numbers get a mention. Particularly interesting to me was the Markov analysis (that is, which word follows immediately after which other word, and how many times).

Chapter 14 discusses files. We've already been using files for a bit. But here we'll learn opening, reading and writing. All of which leads us to discuss exceptions. So we see the exception block and the handling block sequence. Use of databases (which require strings) and therefore also formatting is discussed. Next, we meet pipes. So now that we have operations on files and external programs, we learn one last tidbit about writing modules.

Chapters 15–18 would likely have been one long chapter in a book written by another author. But here, it's been split into 4 separate one. I think that's helpful, as it breaks a long topic into smaller pieces for sequential consideration. It also facilitates the well-paced style of repetition kept fresh with something new added.

Chapter 15 discusses classes and objects. We'll start with how to define classes, and how to get some diagnostics about them. Once we have instances of a class, we can assign values to its attributes. Now we can start to define points, then rectangles. This leads to the use of objects as return values from functions. We note that objects are mutable, so a function may modify its arguments. We learn some more about aliasing, which leads to learning about shallow copies versus deep copies.

Chapter 16 discusses classes and functions. Now that we have objects whose class we defined ourselves, we'll want to display their values both for output and as a debugging aid. We'll also write functions to produce new objects from old. Sometimes it will be convenient to simply modify the object passed as an argument to the function. Of course, it's always possible to change the function definition to return a new, modified object rather than changing the argument. This leads to a discussion of prototyping versus planning. Planning and generalization may lead to simplification and therefore, to reliability.

Chapter 17 discusses classes and methods. The lack of any obvious link between functions taking objects of a class and the class itself leads to the idea of methods. So the `object` and `self` notation for methods is introduced. The rationale for methods is discussed, and a few more examples are given. Then, we turn to the `init` (initializer or constructor) method for a class. From there, operator overloading and type-based dispatch are discussed, leading to a discussion of polymorphism.

Chapter 18 discusses inheritance. Individual objects and class attributes are discussed. The example of a poker program is used. We want to be able to compare objects. So we learn how to encode, and then sort cards. Learning to shuffle leads to the random number generator. With methods to add and remove cards, we are ready to deal. Inheritance allows us to make a "hand" class, which is derived from the "deck" class. So dealing is nothing beyond removing a card from the deck, and adding it to a hand. The idea of class diagrams leads to the "IS-A" and "HAS-A" relationships.

Chapter 19 discusses Tkinter, a binding to Tk/Tcl (although Tk/Tcl isn't mentioned, one may check the python.org web site). It's very nice that several bindings to graphics are defined for Python. Often, the most difficult part of a programming assignment is to get the graphics right in order to gain some understanding from all the lovely numbers.

Finally, the Appendix discusses debugging. The hints range from the practical to psychological counseling. All are useful and valuable, however.

So what has happened? Am I now thinking like a computer scientist? I don't know. As a computational scientist, I use a lot of these ideas everyday. I guess I just learned them informally, along with my war stories and battle scars. But it was nice to get a refresher on some of the new terminology. I hadn't heard "checking the input" called "guardian statements" before. But how many times can one read about if-statements? (Here's the logical condition. Okay. Here's the block of code. Okay. I'll forego the joke about the movie star's seventh husband.) On the positive side, it was useful to get some clearly explained rationale for the ideas of object-oriented programming. That's helpful.

For all my cynicism, I liked this book. The presentation is neat and clean, I might even say cheerful. And I learned a lot, not least of all where higher level languages are going, and the terminology used to express that. All of which makes talking to colleagues easier and more beneficial. And I gleaned some factoids along the way. (I wasn't expecting to learn the complaint phone number for the BBC. No, I won't call them to see if it works, but it's nice to have it, just in case.) I liked the pace of presentation. I liked the constant stirring of topics: a new feature, a hint on debugging, a few words on programming style, some thoughts on programming principles, then on to the next new feature. It really is a nice mix. If one is writing a textbook on scientific programming (rather than computer science as this one is), one could well learn some style tips here.

Now, what about Python for computational scientists? Well, this book isn't really the right source for that. It's aimed towards freshmen's first programming class. So I'll look elsewhere. I have handy (Jonathan E. Guyer et al., FiPy: Partial differential equations with Python, *CiSE* **11**(3), 6–15), so let's see what's here. FiPy is a package for solving differential equations, which the authors apply to problems arising in materials research at NIST. The authors specifically disclaim that FiPy is not the first nor the last word in solving such equations. On the other hand, I can read their code snippets, and after only one book on Python along with a few examples in IDLE, I think I understand what they're trying to say. This must be some kind of breakthrough. Very concise, high-level, fully object-oriented code, and it's actually readable!

So, I'm ready to investigate Python a little deeper. Let's return to the Python web site, where I got the interpreter and accessories. There appear to be interest groups: on interfacing to C (and therefore to Fortran through Fortran's Interoperability with C features); and C++. So maybe I can get some of my existing code to play well with Python. There's some threading primitives, so all thoughts of parallel programming aren't forgotten. And, if I can get to C, then there's always MPI, which may mean starting multiple interpreters around the cluster. How that would work in practice is unclear to me now, but I've not really investigated it.

One shortcoming for prime-time scientific programming was, to me anyway, that there is only one floating point precision. Native Python has rather blunt control of floating point exceptions, more precise control is offered by the NumPy add-on, see below. I saw no control of rounding of floating point operations (which may be difficult with an interpreter), nor of input/output formatting operations. And I saw no unformatted input/output. Perhaps I simply didn't find it on the web site. I might not have looked in all the right places. I wouldn't necessarily expect it in a freshman programming text.

Let's also check the NumPy (numpy.scipy.org) web site. This appears to complete some of the missing scientific computing pieces from the python.org offering. Here are more complete interfaces to IEEE 754, and array classes. There are also extensive sets of math routines and interfaces to Fortran. For computational science, one should investigate NumPy before proceeding too far. One should also investigate SciPy (www.scipy.org) for a collection of packages of interest to computational scientists.

So where does Python fit in the world of scientific programming? It's a crowded world for interpreters. One has Octave and Scilab already. R is well-established in the world of statistics. There are several commercial offerings, with various goals. For pure text processing, there's Awk, Perl, Ruby and more. Python is a very high-level language, fully capable of expressing an object-oriented programming style clearly. I think it has a place in the world. I do intend to keep the interpreter I downloaded on my system.

But this is supposed to be a book review, and I seem to have rambled off into a discussion of Python. I very much like *Python for Software Design.* I would recommend it for teaching a first year computer science course. What about computational science? I hope that instructors in computational science will learn some pedagogical lessons from it. Repeatedly, the book showed code that was simply readable. The feature, its rationale, its uses, and debugging hints are together for collective reference (like an object?). And ideas are repeated as they naturally reappear. Is that how computer scientists think? I don't know. But if that's how they teach, they're doing a fine job. When trying to teach the more difficult ideas of floating point errors, control of step size, mesh refinement, and parallel programming, computational scientists could learn something from *Python for Software Design.*