Exploring Python

by
Timothy A. Budd

This page is blank.

(C’est ne pas un page ??).

Preface

To the student: Why Python?

Given so many other programming languages in the world, why should you learn Python as your first exposure to computer programming? Well, the real answer is that your instructor selected the both the language and this textbook for the course, so what choice do you have? Nonetheless, let me explain why this was a very good decision.

Python is easy, Python is fun, Python is educational, and Python is powerful.

Let's start with the powerful. If you look at the case studies toward the end of the book, you will see that you will end up learning how to do some very interesting things. Tasks like writing your own blog, automatically solving soduko puzzles, reading your iTunes database, or writing a wiki. None of these take more than a page or two of Python code. This is considerably smaller than the equivalent programs would be in almost any other programming language.

Is it easy? Let me fudge and say it is eas*ier*. Computer programming in any language takes skill, organization, logic, and patience. Python is no different in this regard. What makes Python attractive is that you can begin so quickly and easily. Your first Python program can be as simple as 2 + 3:

```
>>> 2 + 3
5
```

Thereafter the path to learning how to create your own complex applications is, we hope, at least clearly laid out, even if it will take some effort on your part.

Active Learning

This book follows an attitude towards teaching that has been termed *active learning*. Rather than treating you, the student, as a passive repository into which knowledge is poured, active learning tries to engage you, the student, as a fully equal partner in the process of learning. Rather than simply telling you how some feature works, I will usually suggest experiments that you can perform to discover the answer on your own.

There are several benefits to this approach. First, it makes you use a different part of your brain than you would if you were simply reading. Second, it gives you, the student, a greater sense of ownership of the knowledge. Third, experimentation is often the fun part of programming. Lastly, by encouraging you to experiment in the discovery of simple information, I hope to instill habits that you will continue to carry with you throughout your programming career. Together, the intent is that active learning helps you more easily retain and use the information you have learned.

What is Python?

For those looking for buzzwords, Python is a high-level, interpreted, reflective, dynamically-typed, open-source, multi-paradigm, general-purpose programming language. I could explain each of those terms in detail, but in the end the result would still not convey what makes Python programming different from other languages. There is no one thing in Python that is not found in other languages, but it is the elegant design and combination of these features into a single package that makes Python such a pleasure to use.

Python is sometimes described as a *scripting language*, for the simple reason that thousands of working programmers daily use the language in this fashion. That is, they use Python as a tool to quickly and easily glue together software applications and components written in many different languages. But such a categorization is far too narrow, and Python can justly be described as a general-purpose language, one that can you can use for almost any programming task you would like to perform.

This is not to say that Python is the only programming language you will ever need or will ever learn. A working computer scientist should know how to use many different tools, and that means he or she should have an appreciation of many different types of language. For example, because Python is interpreted, the resulting programs are often not as fast as those written in lower-level languages, such as C or C++. On the other hand, programs are much easier to write than they are in C. So there is a trade-off, an engineering compromise of the type common in computer science. Is less time in execution of the final program worth spending more time in development and debugging? For the beginning student, and in fact for the vast majority of computer programs, the answer is clearly no. (Another way to express this trade-off is to ask, "whose time is more important, your time or the computer's?") Low-level languages such as C have their place, but only for the small group of computer programs for which ultimate execution time is critically important. You may eventually work on such systems, but not in your first programming course.

Another important category of programming languages are those tied to a specific application. A good example of this category is the language PHP, a programming language used to create interactive web pages. (See www.php.org). A general-purpose language, such as Python, cannot hope to be as easy to use in this application area. But PHP is extremely clumsy to use for purposes other than web pages. If, or when, you start extensive work in such an application area you will want to learn how to use these tools.

Bottom line, Python is an excellent place to start. And to stay, for many of your programming tasks. But you should not assume that it is the last language you will ever need or learn. Fortunately, languages have many features in common with each other. A solid foundation in one language (such as Python) makes it much easier to learn a second

(or third, or forth). An appendix at the back of this book provides hints as to how one should approach the task of learning a new language.

History of Python

Python was designed by Guido van Rossum while he was working at the CWI (the Centrum voor Wiskunke and Informatica; literally "center for wisdom and informatics") a world-class research lab in the Netherlands. The CWI group he was associated with designed a programming language called ABC. I was fortunate to spend a year with this group in 1985. ABC was clearly intended as a pedagogical tool for teaching programming, and a great deal of work went into developing both the language and associated teaching material.[1] The language ABC had a number of features that were impressive for the time: a tightly integrated development environment, interactive execution, high level data types (lists, dictionaries, tuples and strings), dynamic memory management, strong typing without declaration statements and more. The idea to use indentation for nesting, and eliminate the brackets or BEGIN/END keywords found in most other languages, was taken directly from ABC. So was the idea of dynamic typing. Software development in ABC was both rapid and enjoyable, and totally unlike almost any other competing language. (The one exception might be Smalltalk, which was just becoming well known in 1985. Indeed, during my time at the CWI I was writing a book on Smalltalk, and part of the work I performed during that year was to explain to my colleagues in the ABC group the basic ideas of Object-Oriented programming, which I myself was only just beginning to understand).

Guido started designing Python around 1990. For those familiar with the earlier language the heritage of ABC in Python is clear. Guido discarded some of the annoying features of ABC, and kept all the best ideas, recasting them in the form of a more general-purpose language. By then the mechanisms of object-oriented programming were well understood, and the language included all the latest features. He added a number of features not found in ABC, such as a system for modularization and including libraries of useful utilities. Python was released to the world in 1991, and very quickly attracted a loyal following. Python's design turned out to be general enough to address a much wider range of applications than ABC. (To be fair, the designers of ABC were focused on teaching, and never intended the language to be general-purpose). The features that programmers appreciated in 1990 are still the same today: ease of use, rapid software development, the right set of data types that help to quickly address most common programming problems.

Python, Monty

[1] See the Wikipedia entry on ABC for further discussion of this language. The Wikipedia entry for Python has a much more complete history of the language. There is also a Wikipedia entry that explains the concepts of active learning. Wikipedia is found on the web at www.wikipedia.org.

The name, by the way, owes nothing to the reptile and everything to the 1970's BBC comedy series *Monty Python's Flying Circus*. Many die-hard Python programmers enjoy making sly references to this series in their examples. You don't need to have seen *Monty Python's Life of Brian, The Meaning of Life, And Now for something Completely Different,* or *Monty Python and the Holy Grail* or even *Spamalot* in order to become a Python programmer, but it can't hurt, either.

To the Instructor

I will begin this section with the same question I used at the start of the preface to the students. Why is Python a better programming language for the first course than, say, C, C++, Java, C#, Delphi, Ada, or Eiffel, just to name a few alternatives? The answer, as I suggested earlier, is that students will find that Python provides a much easier entrance into the world of programming, yet is complete enough to provide a comprehensive introduction to all the important ideas in programming, and is fun to use.

The fact that Python can be used in both an interactive and textual style makes the barrier for the beginning student extremely low. This is not true for other languages. To write even the simplest Java program, for example, the instructor must explain (or worse, not explain and leave as a magic incantation) ideas such as classes, functions, standard input, static variables, arrays, strings, and more. In contrast, the first Python program can be as simple as 2 + 3:

```
>>> 2 + 3
5
```

The positive influence of interactive execution for the beginning student cannot be overstated. It permits (and the conscientious instructor should encourage) an exploratory and active approach to learning. To find out how something works, try it out! This empowers the student to take control of his or her own voyage of discovery, instead of simply playing the role of a passive container into which the instructor (or the book) pours information. I have discussed this *active learning* approach in my earlier remarks for the student.

But the fact that simple things are easy to write in Python should not be an excuse to imagine that the language is just a toy. It is a credit to the good design skills of Guido van Rossum (the language designer) and countless others that simple ideas are simple to express, and complex ideas can also be illustrated with simple examples. In what other language might an introductory textbook include examples of a blog, a wiki, or an XML parser?

Python is also an excellent vehicle for teaching computer science. All the basic concepts of programming (ideas such as values, variables, types, statements, conditionals, loops, functions, recursion, classes, inheritance, just to name a few) can be found in Python. The student gaining experience with these topics in this language is therefore in an excellent position to more easily learn other languages at a later time. An appendix offers some

general hints on how to go about learning a second, or third, programming language. These hints work for both the student coming to Python with experience in a different language, as well as student to whom this book is directed, those learning Python is their first language.

Organization of this Book

The first eleven chapters of this book present a more or less conventional introduction to programming. Students learn about variables, types, statements, conditionals, loops, functions, recursion, classes and inheritance. What makes my approach different from that found in many other books is an attitude of exploration. Basic ideas are explained, and then the reader is lead through a process of experimentation that helps them find and test the limits of their understanding. By making the learning process active, rather than simply a matter of absorption, we engage the reader in a wider range of cognitive operations, and hopefully make the material both more enjoyable to learn and easier to remember.

The chapters after the first eleven represent a series of case studies. These explore the use of Python in a number of representative programming tasks. These tasks include the creation of a blog, a sudoku solver, a wiki, reading an iTunes database as an example of parsing XML, and more. These are intended to both illustrate good Python programming style, and to open the readers mind to the range of problems that can be addressed using the language. The case study chapters should be examined only after the students have examined the first eleven chapters, as they assume a familiarity with that material. After the first eleven chapters, however, the organization is much less linear. Instructors should feel free to present the latter material in whatever order they wish, or pick and choose chapters as fits their needs.

Although the basic syntax of Python is covered in the first chapters and in an appendix, the book cannot be considered to be a substitute for a reference manual. Much of the power of Python derives not from the basic language syntax, but from the wide range of libraries that have been developed to address problems in different domains. Fortunately, excellent reference material can be found at the web site www.python.org. Similarly, the scope of this book is purposely limited. Programs are generally restricted to no more than two pages in length. Readers interested in larger examples are encouraged to look at the much more comprehensive, encyclopedic, and **heavy** book *Programming Python*, by Mark Lutz (published by O'Reilly).

Advanced Packages and Libraries

There is a huge amount of exciting and fun activity occurring right now in the Python universe. Unfortunately, most of this requires the programmer to download and install at least auxiliary libraries, if not complete applications. Examples include the integration of OpenGL and Python for 3-D visualization, game development systems such as PyGames, visual development environments such as Alice, and much, much more. For a number of reasons I have resisted talking about these topics in this book. First, I doubt if many

students encountering programming for the first time using Python will have the ability, even after a term or two experience with Python, to install such systems on their own. Second, the speed at which changes are occurring in this arena is phenomenal. Almost anything I could say in print would have a high likelihood of being obsolete, or even wrong, by the time the book went to press. On the positive side, if I am successful in my goal of encouraging the student to embrace the ideas of active learning, then by the time they are finished with this book they should have not only the knowledge, but the self-confidence, to find information on the internet on their own (and the internet is now where the most reliable and up-to-date information is to be found). Just try googling with the phrase "Python OpenGL", or whatever topic you want to explore. To those students, I say: good hunting, and have fun!

Acknowledgements

I'm sure there will be many.

Table of Contents