Python in Education: Raising a Generation of Native Speakers

Frank Stajano
AT&T Laboratories Cambridge
24a Trumpington Street, Cambridge CB2 1QA, UK
http://www.uk.research.att.com/~fms/
and
University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK
http://www.cl.cam.ac.uk/~fms27/

Abstract

Primarily because of its young age, Python is still a language that people only discover after having digested a few others: while many of its users love it enthusiastically, almost nobody is a native speaker of it, in the sense of having been exposed to it before any other.

As computer literacy evolves from desirable to necessary for people from all backgrounds, computer professionals and academics are responsible for taking a long-term view on how best to educate the next few generations of computer users. Form shapes contents, so the influence of a clean yet expressive first language in establishing good mental models and programming habits should not be underestimated.

This paper discusses how Python, with its high level of abstraction and judicious balance of simplicity, conciseness and versatility, is an excellent choice to introduce the fundamental ideas of the art of programming.

1 Introduction

In many programming contexts scripting languages are steadily gaining popularity as the most effective way of getting the job done. From rapid application development to system administration and from web automation to scientific computing, many experienced developers embrace scripting enthusiastically once they discover how much time and frustration they save compared to solving the same problem with a lower-level system language. A lucid analysis of this shift is offered by Ousterhout [7] who remarks that "computers become faster and cheaper in comparison to programmers" and hence that reduced execution speed is a fair price to pay in return for increased productivity and expressive power.

What only a minority seem to realise, though, is that a cleanly designed scripting language such as Python is also eminently suitable for introducing a virgin mind to the art of programming. Some will consider this suggestion as highly dangerous, on the grounds that new programmers should be introduced from the start to such fundamental principles as type and variable declaration: for them, to start a fresh mind on a scripting language might cause long-term brain damage of the sort that can be observed on those, such as yours truly, who started with BASIC in the Seventies or Eighties when home computers first came out. This paper aims to dispel such fears and to highlight, on the contrary, the virtues of Python as an effective language for introductory programming courses.

I stand behind this thesis with passion and dedication, but unfortunately without yet having had a chance to validate it by teaching a course or writing a textbook; this makes this paper an opinion piece rather than a scientific report substantiated by experimental data. I was however delighted to discover, after having written most of my first draft, a now well-known visionary plan aiming in the same direction (and beyond!), originated by none less than Guido van Rossum himself [13]. In recognition of these two circumstances I shall keep this paper brief—a strategy that most readers will undoubtedly appreciate.

2 Form shapes contents

From the point of view of the Python evangelist (which cannot be totally ignored given the conference at which this paper is being presented) the suggestion of raising programmers on Python might appear like a brilliant proactive marketing strategy: instead of the obsolete method of converting programmers by showing them how much better Python copes with their specific problems, one gets in their brain ahead of the competi-

Appears in *Proceedings of* 8^{th} *International Python Conference*,
24–27 January 2000, Washington, D.C.

tion, before they even stand a chance to become polluted by the infidels.

Regretfully, the evangelists will have to face disappointment, since the main motivation for this proposal is not to recruit new followers to the Faith. Python here is a means, not an end, and indeed for those who will go on to become professional programmers it will only be a stepping stone towards other languages.

Many beginners' courses are, for obvious reasons, market driven, in the sense that they introduce the students to programming using the language that the students wish to be able to proudly display on their *curriculum vitæ*: this typically means Java in the luckier cases, or Visual Basic otherwise[1]. Without intending to discuss the issue of which language would actually be most beneficial to those students' future career, we wish however to investigate which language is most suitable for their introduction to programming. It must be emphasised that the two issues are largely orthogonal: anyone taking up programming as a professional occupation will by necessity have to learn a number of different languages over the years, so there is no fundamental need to use for the initial training the same commercially accepted language that the student expects (rightly or wrongly) to be using once employed, as long as that other language is also taught in due course.

On the contrary, just like the knowledge of more than one natural language makes an author more proficient and more aware of certain expressive nuances and subtleties that might otherwise go unnoticed, so the knowledge of several programming languages, with their different approaches to the task of expressing an algorithmic computation, broadens the mental horizon of the programmer and highlights alternative and sometimes more appropriate ways of solving the problem at hand.

Form shapes contents: being able to express a certain operation easily may make the difference between finding and not finding a solution. As Hofstadter [3] appropriately notes in the context of what used to be called artificial intelligence, the choice of one programming language over another is akin to an invitation to solve a certain class of problems:

> The "space" of all possible programs is so huge that no one can have a sense of what is possible. Each higher-level language is naturally suited for exploring certain regions of "program space"; thus the programmer, by using that language, is channelled into those areas of program space. He is not *forced* by the language into writing programs of any particular type, but the language makes it *easy* for him to do certain kinds of things. Proximity to the concept, and a gentle shove, are often all that is needed for a major discovery—and that is the reason for the drive towards languages of ever higher levels.
>
> Programming in different languages is like composing pieces in different keys, particularly if you work at the keyboard. If you have learned or written pieces in many keys, each key will have its own special emotional aura. Also, certain kinds of figurations "lie in the hand" in one key but are awkward in another. So you are channelled by your choice of key. In some ways, even enharmonic keys, such as C-sharp and D-flat, are quite distinct in feeling. This shows how a notational system can play a significant role in shaping the final product.

It is thus appropriate to choose, for an introductory programming course, a language which makes it easy to express the important ideas without getting lost in irrelevant detail. As a cultural imprinting, the student needs to learn the fundamental concepts of program construction and the basic skills of specifying, expressing and debugging an algorithm. At this initial stage it helps if the notational system supports just that, without getting in the way with too much syntax or low-level detail. Python's clear conciseness is what is needed to visualise the algorithm in 15 lines as opposed to two pages of scaffolding, brackets and ancillary declarations. One is not forced first to write a concise pseudocode version and then some real code to implement it: with Python, the pseudocode is in fact also real code, and already executable.

A word of warning. Kernighan and Pike [4] appropriately insist on style and clarity as the foremost qualities of a good program—and it is somewhat refreshing to see the C demigods inviting readers to caution when handling the more cryptic and less safe features of the language for which they were partly responsible. Python tutorials, on the other extreme, all too often put the emphasis on fun, not on style, perhaps in the overenthusiastic hope that the cleverly designed layout and syntax of the language will make the source readable for free. While Python's structure may give it a head start on clarity compared to C, it would be wise to bank on this advantage by emphasizing to students the value of readability and good style, which require a conscious effort in Python like in any other language, including English.

[1] Other languages such as C++ may constitute even more desirable CV items, but are typically outside the scope of a *beginners'* course.

3 The target audience

We have spoken so far of introductory courses meant for future programmers, i.e. people that would naturally continue to intermediate and advanced courses, where they would learn new skills and new languages as appropriate. But the more novel and provocative aspect of this proposal to use Python in education sees this only as the marginal case. Future programmers are people who *choose* to use computers. The really relevant case—the one towards which the community of computer professionals and academics has the greatest responsibility—is that of the vast majority of people who will be *forced* (by us) to use computers and will probably never attend an intermediate or advanced programming course: historians, surgeons, lawyers, musicians and so on. At present, they may not receive any programming training at all. The idea put forward here is that they all deserve to get such training, in moderate quantity, at around secondary school age, and that Python would be a good language to teach them. The Van Rossum proposal [13] is even more radical, advocating that programming should be taught at elementary school level as a basic skill, on a par with natural language and arithmetic.

Establishing the most appropriate age at which to introduce programming in the curriculum is left open for debate; here we shall only remark that the implementation details may depend on other factors than just the intrinsic merits of the new subject. As Wilkes says about historical studies [14]:

> It is always easy to make a strong case for including new topics in a syllabus. The problem is to decide what to leave out to make room for the new material and, for this reason, discussions of syllabus reform always resolve themselves into discussions of what to cut out, not what to put in.

But, apart from the age question, the crucial message is that all children, and not just the technically minded, would benefit from learning the principles of programming, as opposed to just being taught how to drive prebuilt applications as users.

Spreading the knowledge about programming has at least two advantages: the first is to somewhat demystify computers and give those future users the intellectual tools to understand, at least qualitatively, what goes on under the hood. Instead of I-don't-want-to-know-about-it-ly[2] blaming all malfunctions on voodoo or Murphy, they will have some insight into possible causes for the problem and might be able to devise, if not a fix, a safer or less resource-hungry way of doing what they want.

The second advantage is to give back to ordinary people the power to solve simple problems. In the days of the Apple II and of its even less powerful home-oriented followers from Commodore, Sinclair and Acorn, the distinction between using a computer and programming one was blurred. The BASIC interpreter in ROM was more than just a programming language: it was also an operating system (insofar as there was one) and a glue language to plumb things together. Despite the then-popular claim that "a computer can do anything", those machines did very little; but at least their users were in control and there was no barrier to writing short programs to do simple things. Over the subsequent 20 years, home computers have become unbelievably more powerful (as well as better suited to actually performing useful tasks, such as organising one's collection of comics), but BASIC in ROM has gone. In fact, from the mid-Eighties, most home computers have been shipping without a programming environment at all. And even BASIC, for all its many perversions, was not as bad as what unofficially replaced it as the native programming tool of the average machine, namely the MS-DOS batch "language"...

It is easy, with hindsight, to recognise that this need for elementary programmability is perfectly solved by scripting languages. Only a tiny fraction of computer users will be professional programmers, but all users deserve to be taught how to write useful five-line scripts to automate simple repetitive tasks[3].

4 Is Python a good first language?

As a separate issue from the desirability of introducing schoolchildren to programming, let's come back to Python in particular, and to its suitability as a first language.

One might think, especially in the light of what we just said, that two types of students ought to be distinguished, namely the future professional programmer and the future non-technical user, since the requisites for the respective ideal first languages might be quite different.

In the first case, one might wish to emphasise good programming habits that favour program correctness and maintainability, while in the second case one might lean more towards giving the students some directly usable practical skills, especially if this is expected to be their first and last programming tutorial.

[2] `<0.5 wink>` to Tim Peters who, with characteristic wit, introduced such insidiously contagious linguistic acrobatics in `comp.lang.python` years ago.

[3] This brings to mind the magic power of Expect [5], which allows your script to drive character-based applications as if it were an interactive user; but unfortunately the tool's usefulness is reduced by the modern prevalence of graphical interfaces. Besides, Expect is far too complicated for a non-techie.

For some, the requirements for the first case will mark Python as unsuitable. Faculty member Peter Robinson [10], in presenting the criteria for the choice of language to be taught initially to the computer science undergraduates at the University of Cambridge, lists "mathematical basis", "strong typing" and "functional emphasis" as the three main objectives, accompanied by "a friendly environment for experimenting", which translates as "an interpreted language". This because the students "need to start with a sound foundation for programming that can establish the principles which will subsequently be applied in many different languages".

The value of being able to see a program as a formal description of an abstract algorithm, and to analyse algorithms mathematically, is emphasised. From these requirements Robinson justifies Cambridge's choice of ML which, as an interesting side effect, puts all students on an equal footing: even those with previous programming experience have never heard of the language before and are typically accustomed to imperative rather than functional programming.

Python cannot score as well as ML against these criteria, especially for what concerns the mathematical purity: although it includes functional constructs and supports functions as first class objects, Python lacks ML's sophisticated type inference mechanisms and is certainly not a "pure" functional language—or a pure anything else for that matter. In fact, the following comment on C++ by Stroustrup [11] applies equally well to Python, whose computational model fruitfully borrows from various inspiring sources.

> Too often, "hybrid" is used in a prejudicial manner. If I must apply a descriptive label, I use the phrase "multi-paradigm language" to describe C++.

The reason why an educator, notwithstanding Robinson's valid points, might still choose Python over ML for a first course to computer science undergraduates is probably one of simplicity. While functional concepts can still be presented and explored in Python where appropriate, the core of the course can proceed using a notation that, while less rigorous and more detached from the mathematical foundations of computing, is nevertheless going to be much closer to the usual pseudocode used in language-neutral textbooks. The outstanding introductory volume by Goldschlager and Lister [1], which Robinson himself elsewhere lists as required cover-to-cover reading for the computer science undergraduates even before they turn up at Cambridge [9], presents its algorithms in a format and programming style that is going to look rather more familiar to a native Python speaker than to a native ML speaker.

Besides, one should not entirely dismiss the possibility that the choice of ML, while technically and pedagogically sound, might have an element of Cambridge elitism in it, as hinted at in the following extract from Haemer's witty and enjoyable chronicle of the 1994 VHLL symposium [2].

> While [speaker Andrew Koenig] isn't put off by some of ML's unusual properties, Andy admits that it is the twenty-first programming language he's learned. I didn't have the impression that he was the norm. Neither, I think, did he.
>
> Early on, he warned us that "ML makes far fewer concessions to practicality than you're used to, but the payback is occasional breathtaking elegance." True enough, but it wasn't long before John Ousterhout asked whether it would be unfair to characterise ML as a language for people with excess IQ points. Someone in the audience volunteered that ML is a standard undergraduate programming language at Cambridge, without saying whether that meant "yes" or "no."

Python's principal advantage as an introductory language is its high abstraction level, which is appropriate to introduce the fundamental concepts of algorithms without getting distracted by irrelevant details such as machine level micro-optimisations and memory allocation issues. Naturally, the future computer scientists will have to be introduced to such details at some stage—but this can happen later, in the context of explaining them how Python's own building blocks were made.

Anyone with an interest in the didactics of programming should not miss the excellent article by Stroustrup [12] on learning Standard C++. Since C++ is so large, it must necessarily be conquered in stages; and, Stroustrup argues, an all too frequent mistake is to take the C subset as the first such stage.

> In my considered opinion, that's not a good [choice]. The C-first approach leads to an early focus on low-level details. It also obscures programming style and design issues by forc[ing] the student to face many technical difficulties to express anything interesting.

What he instead suggests is an approach that, among other things, "presents code relying on relatively high-level libraries before going into the lower-level details (necessary to build those libraries)", "presents common and useful techniques and features before details", and "focus[es] on concepts and techniques (rather than

language features)". To use Python to introduce programming is consistent, in spirit, with the above guidelines. Of course, since Stroustrup discusses studying C++, he implicitly targets students who have already committed to taking programming seriously; but the same guidelines define a sensible strategy even for our "type 2" students, i.e. those for whom the fundamentals of programming are a useful piece of 21^{st} century culture but not the foundations of a career.

Someone might even argue that the ultimate goals for a first course to "type 1" and "type 2" students are not that different after all. In both cases we want to culturally imprint the students with a good methodology and an archetypal idea of how a complex construction such as a piece of software should be built in order to be manageable—indeed this may be what that makes the course more generally useful, even to the "type 2" students who will never write a program longer than ten lines. Moreover, again in both cases, we want to retain the students' interest by letting them practice with real problems (such as file manipulation and Internet programming) rather than just abstract data structures; and Python's standard library is just what we need here.

At a structural level, the absence of a direct language ancestor saves Python from perversions dictated only by the need for compatibility. The language is fairly clean, in the sense of being based on the regular combination of a few powerful orthogonal constructs.

At a syntactic level the minimalistic punctuation, while conceptually irrelevant, is actually a significant advantage during the learning phase. Meyer, in the second edition of his masterpiece [6], confesses his disappointment at discovering that students learning Eiffel considered the absence of statement-delimiting semicolons as the best feature of the language:

> Some of the Discardists were very forceful, in particular a university professor who said that the main reason his students loved the notation is that they do not need semicolons — a comment which any future language designer, with or without grandiose plans, should find instructive or at least sobering.

It is in fact only fair to remark that, if the block structure of language-neutral pseudocode can be expressed clearly and unambiguously by nothing more than a legible page layout, there are few good excuses to justify an abundance of brackets, semicolons and begin-end markers in the real code just to simplify the compiler. Making indentation count as block structure also has the beneficial side effect of forcing everyone to indent their source code correctly, under penalty of it not working.

Sometimes even historical accidents are useful. Python's explicit `self` argument to the methods of a class instance, which seasoned OO programmers may view as an inelegant artifact of the minimalistic implementation, helps less experienced programmers understand who is calling what on what else, and how object oriented programming actually works. When I was once asked to explain subtle distinctions between class and instance methods, it was helpful to be able to point out that the class method wouldn't have worked in that example because, lacking `self`, it couldn't find the object to which the operation applied. The same explanation would have been more nebulous in "more advanced" OO languages.

5 It can't all be good!

It is reasonable to expect that, under at least *some* aspects, Python will be unsuitable as a first language, since it was not originally designed for this specific purpose. Van Rossum [13] openly admits this possibility and declares himself ready to modify the language where appropriate. He quotes case sensitivity of identifiers and truncation of integer division as features that actual teaching experience proved to be confusing. But there may in fact be more substantial sources of problems.

First and foremost, the whole conceptual model of reference counted objects may at times produce side effects that, for people who haven't been introduced to the delights of pointers and explicit memory management, are justifiable only through black magic. The canonical example is in the Python FAQ [8], entry 4.50: the following apparently innocuous attempt at a two-dimensional matrix will generate a mischievous structure in which poking an element in one row makes it appear in all the others too:

```
A = [[None] *2 ] *3
```

A less fundamental point, but nonetheless one that never fails to puzzle newcomers, is the distinction between lists and tuples ("So why can't you get away with just lists?"), and the related business of mutable vs. immutable.

Finally, for a language to be used to teach programming concepts, not being able to specify types in, say, the signature of a function is probably a drawback.

These aspects of Python's behaviour are so deeply rooted in its current structure that it is hard to imagine them being removed in a future revision of the language even if actual practice were to show that they effectively cause serious problems to the young students. It will be extremely interesting to see how the problem is tackled if

it effectively arises. Breaking compatibility is allowed: the challenge is to devise a new construct that still fits in naturally and elegantly—in other words, one that still has the Python nature.

6 Conclusions

Python has done very well as a general purpose scripting language. As with some of its direct competitors, its high level data structures and high level primitive operations allow most typical data manipulation tasks to be expressed in tens rather than hundreds of lines of code. Unlike other scripting languages, however, its clean design and its good support for modularity and objects have allowed developers to fruitfully keep on scripting way past the canonical now-move-to-a-system-language limit of a thousand lines or so. Indeed it is thanks to this circumstance that, in a self-sustaining virtuous circle, many of the excellent contributions in the language's rich standard library have come to life.

But, despite this success in the rapid application development arena, Python is still under-utilised. The same cocktail of well chosen language trade-offs that gave Python the edge in the hands of experienced developers also makes Python a great resource for the important task of teaching programming. Newcomers—for whom the first language is a system to organise their new thoughts about algorithms as much as it is a tool for building programs that work—find in Python a clean notation that invites expression at the appropriate level for their experience; they may familiarise with general programming concepts in an environment built out of the regular composition of a few powerful constructs, rather than out of the irregular accumulation of cute hacks, compatibility kludges and special cases. As an added bonus, they get a language that enables them to perform useful practical work on their own data (Python's versatility with text manipulation and Internet protocols comes immediately to mind) rather than being confined to toy worlds populated by lists, trees and integers. Finally, they get a language that will take them a while to outgrow—as demonstrated by the multitude of professional developers who, even as non-native speakers, have now adopted Python for a substantial portion of their work.

Some of the best names of Pythonland have already committed to a substantial concerted effort that will validate the above claims and make Python, with the invaluable feedback loop of actual school experience, an even better first language. Now that the *Computer Programming for Everybody* project rationale [13] is out, there is in fact much less need than before for a paper such as this one. At this stage, the contribution of this piece is primarily to offer some extra food for discussion, from a slightly different point of view and in a small and easily digested format.

Actually, within the gastronomic metaphor, the most appropriate image is perhaps that of an appetiser: this little piece doesn't give the reader any meaty substantiated facts—nor any *cheesy* language comparison tables, for that matter—but if a potential educator (or publisher, or author...) finds it sufficiently enticing to want to find out more and maybe bite a bigger chunk of the action, this appetiser will have served its humble purpose.

7 Acknowledgements

It was a pleasure to discuss some of these ideas over lunch with Frank Willison at IPC7 in 1998, as we clearly both believed in Python's great potential as a teaching language.

I am also grateful to the anonymous referees (particularly the one thanks to whom I bought the delightful *The Practice of Programming* [4]) for their pertinent and encouraging comments.

References

[1] Les Goldschlager, Andrew Lister and Timothy R. Lister. *Computer Science: A Modern Introduction*. Prentice Hall, Englewood Cliffs, Mar 1988. ISBN 0-13-165945-6.

[2] Jeff Haemer. "Very High Level Languages Symposium Report". *;login:*, **20**(1):5–10, Feb 1995. ISSN 1044-6397. (The symposium, organised by Usenix, was held in Santa Fe, New Mexico, from 1994-10-26 to 1994-10-28.).

[3] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, New York, 1979. ISBN 0-465-02685-0.

[4] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999. ISBN 0-201-61586-X.

[5] Don Libes. *Exploring Expect*. O'Reilly and Associates, 1995. ISBN 1-56592-090-2.

[6] Bertrand Meyer. *Object-oriented Software Construction, 2^{nd} ed.* Prentice Hall, 1997. ISBN 0-13-629155-4.

[7] John K. Ousterhout. "Scripting: Higher Level Programming for the 21^{st} Century". *IEEE Computer*, **31**(3):23–30, Mar 1998. ISSN 0018-9162.

http://www.scriptics.com/people/john.ousterhout/scripting.html.

[8] Python Software Activity. http://www.python.org/doc/FAQ.html.

[9] Peter Robinson. “Preparing to study Computer Science at Cambridge”. http://www.cl.cam.ac.uk/Teaching/Preparation.html.

[10] Peter Robinson. “From ML to C via Modula-3 an approach to teaching programming”, Dec 1994. http://www.cl.cam.ac.uk/~pr/mlm3/mlm3.html.

[11] Bjarne Stroustrup. “Why C++ is not just an Object-Oriented Programming Language”. *OOPS Messenger*, **6**(4):1–13, Oct 1995. ISSN 1055-6400. http://www.research.att.com/~bs/oopsla.pdf.

[12] Bjarne Stroustrup. “Learning Standard C++ as a New Language”. *The C/C++ Users Journal*, May 1999. ISSN 1075-2838. http://www.research.att.com/~bs/new_learning.pdf.

[13] Guido van Rossum. “Computer Programming for Everybody (Revised Proposal): A Scouting Expedition for the Programmers of Tomorrow”. CNRI Proposal 90120-1a, Corporation for National Research Initiatives, Jul 1999. http://www.python.org/doc/essays/cp4e.html.

[14] Maurice V. Wilkes. “Historical Studies In Science And Technology And The Uses To Which They Can Be Put”. *Notes and Records of the Royal Society of London*, **53**(1):3–10, Jan 1999.