

The First Language - A Case for Python?

Tony Jenkins

School of Computing, University of Leeds, Leeds, LS2 9JT.

tony@comp.leeds.ac.uk

Contents

1. [Introduction](#)
2. [Python](#)
3. [The First Program](#)
4. [Python in Practice](#)
5. [Conclusions](#)
6. [Acknowledgments](#)
7. [More Details](#)
8. [A Note on Naming](#)
9. [References](#)
10. [Author Details](#)

Abstract

Python is a fully featured object-oriented programming language. Its commercial use is widespread and increasing, but its use in a teaching setting appears as yet to be limited (but is also increasing).

This paper presents some features of Python that might be of interest to those who teach programming, and reports on the experience of using Python as a "pre-programming" language as part of a course aimed at a typical cohort of programming students. It thus considers the potential that Python may have to be used as a "first language".

The results of this work appear to be encouraging, and some initial evaluation has taken place. It is clear that there are many fewer cognitive obstacles to overcome before a novice can start writing programs with Python than with many other languages. It does indeed appear that Python has something to offer as an introductory programming language.

Introduction

There has been much (many might argue that there has been too much) written about the ideal first programming language to teach to novice programming students. It must be made clear from the outset that the purpose of this paper is not to evangelise about the possibilities of using Python to teach novices. That could well start a language war, and language wars (while often amusing to an outside observer) are always bloody and always eventually futile. Rather the purpose here is to raise the possibility of using Python as an introductory language, and to report on some preliminary experiences of doing just that.

It can be argued that the language used in a first programming course is irrelevant, and I confess that I have in the past argued that myself. The outcomes of this work have made me reconsider my previous position. The argument goes that the aim of the course is to convey programming principles that can be applied in a range of situations, not the specific details of one particular language. This thinking is based on the notion

that a novice who learns to program reasonably well in, say, C++ will have little difficulty coming to terms with Java or indeed with any other language that the future might produce. This argument is reasonable as far as it goes, but it is impossible to ignore some features of the programming language that the students must grapple with and, hopefully, eventually master.

It is a shame that the languages that our students encounter when they come to learn to program are languages that are designed for commercial use by experienced commercial programmers. Modern languages like Java and C++ contain a host of features that most students will never need, and which we should probably admit are a mystery to many of the students' teachers. The development environments that they use are often designed for commercial programming and contain a baffling array of features that are beyond their comprehension. The problem is that novice mistakes can often generate the most arcane error messages because novices make errors that commercial programmers never would. This produces errors that are comprehensible only to the truly initiated or enlightened.

The reason for this curious state of affairs where novices grapple with tools that were not designed with their needs in mind seems to be that students demand to be taught skills and languages that they perceive to be in demand in industry. They see job advertisements for Java programmers, and they want to learn Java. They do not know that Java is a poor first language to learn, and institutions and teachers choose not to disillusion them. That, at least, is the theory. And presumably the students' teachers agree, or we would not teach them C++ or Java.

Let us turn to history. A long time ago, Niklaus Wirth recognised this problem when he proposed a language specifically designed for teaching, Pascal ([Wirth, 1971](#)). He recognised a "vicious circle" whereby students demanded to learn a language that is in demand by industry, and industry then demanded students skilled in this language since this is the skill that is available. Wirth deprecated this "stagnation" and proposed Pascal as an ideal teaching language that could serve as a fine introduction to a variety of other languages. Wirth believed that a student who mastered Pascal could reasonably easily learn C, C++, or any other language. Of course, no serious computing department would now consider teaching Pascal; there is no commercial demand for Pascal programmers.

Most departments now seem to have chosen Java as their first language. There are some fine initiatives to make Java a suitable first language, notably the BlueJ environment ([Kölling, 2004](#)), but Java remains a commercial language designed for commercial programmers. Students must learn (or be told to ignore the details of) a lot of Java before they can write even the simplest program. The API is massive, as is its documentation, and intimidating for the novice. We might also pause to consider why, if we need to develop something like BlueJ in order to teach Java, we are teaching Java in the first place?

Python is a relatively new object-oriented language. It has commercial use, and this use is increasing. It is possible that it offers some pedagogic advantages over the ubiquitous C++ and Java; it is certainly time to find out. This paper reports on an attempt to investigate just this. But first some features of the language for the uninitiated.

Python

Python is described as "an interpreted, interactive, object-oriented programming language" ([van Rossum](#)). The claim is that Python combines "remarkable power with very clear syntax"; this for a start should be of interest to those who teach programming.

Python is free and is effectively available for any computer system that has a C compiler. Downloads of binaries are readily available, and if all else fails the source code can be obtained and compiled. Python code is portable between operating systems at both source and byte code levels.

As with much free software, there is a plentiful supply of free on-line tutorials and documentation. Some of these have now found their way into print (for example Alan Gauld's [Learn to Program using Python](#)) while others remain stoically free (for example Downey et al's excellent [How to Think Like a Computer Scientist: Learning with Python](#)). Following the principle of the "Bazaar" rather than the "Cathedral", the quality of these tends to be high and many are constantly revised, improved, and updated. Students and teachers alike are sure to be keen on high quality freely available teaching materials.

This all seems rather promising. Here is a language that claims power and a clear syntax for which quality teaching materials are available free of charge. It would be foolish not to investigate further. But first it would be, of course, foolish to carry on if Python cannot claim widespread commercial use.

That there is commercial use is apparent immediately on visiting the [Python home page](#). A visitor is greeted with a quote from the no less a person that the "director of search quality" at Google, and learns that Python is "an important part of Google" and that "we're looking for more people with skills in this language". Clicking on, there are more testimonials. Python was used by Industrial Light and Magic in the making of Star Wars: Episode II, and is used in NASA's engineering data management applications. No shortage of commercial use, then. And it appears to be commercial use in interesting cutting-edge applications.

Of course, similar applications at other organisations make use of Java or C++, or even Perl. It is time to look at some syntax and language features to evaluate what a novice will have to learn to write a Python program.

The First Program

The first program that any programmer writes is, of course, the same. It simply displays the message "hello, world" or something similar on the screen. This is a fine tradition, and one that will serve to investigate the very basic elements of what a student must learn.

Without further ceremony, here is that program in Python:

```
print "hello, world"
```

That's it.

On a Unix system the program can be executed from a command line:

```
% python first.py
```

Or an extra line can be added to point the way to the Python interpreter so that the program will function as a self-contained script:

```
#!/usr/bin/python  
print "hello, world"
```

On a Windows system it is sufficient to double-click the icon representing the file containing the program. The icon itself is a rather splendid smiling python.

Let us not make the mistake of thinking that this is trivial, though. There is much that the novice must learn in order to get this far and achieve this admittedly modest success. To write this program a novice must understand many things. They must have some conception of what a computer program is, perhaps to the level of understanding that it is a sequence of commands that a computer processes. Here there is only one command to understand (`print`), and the purpose of that is surely clear enough. There remain the hurdles of learning how to create the program, execute it, and find the output (and these are hurdles that should not be underestimated), but once these are done the novice is up and programming. With Python, of course, they can

be programming in whatever computer environment they prefer and if, for example, they choose to work in Windows at home and Unix at University, their code will work correctly with no changes on either platform.

Now to compare the first program with the equivalent program in other languages, starting for not particularly good reason with C++. The practical hurdles remain much the same, of course, so the focus must be on the details of the syntax and on the concepts that a novice must understand. It is essential, of course, that a novice does understand these concepts since it is surely totally unacceptable in higher education to tell a student to ignore something and come to understand it later.

The traditional first program in C++ can be written in many ways, and will differ slightly on different computer systems, but a reasonable attempt is:

```
#include <iostream>

using namespace std;

int main ()
{
    cout << "hello, world" << endl;

    return 0;
}
```

This simple program is so laden with concepts that it is difficult to know where to start. Reading from the top there is the notion of an included library (without which there can be no output), and there is the need for a namespace (what exactly is "std"?). In the program itself there is a function, defined by what amounts to a prototype, with a returned value (why is it zero? does it matter?). The single line that has an effect starts with the mysterious rune `cout`, which few would guess had anything to do with displaying text on a screen. And what precisely is the `<<` operator achieving? Are those curly brackets important? `endl`?

It is possible to teach C++ by presenting novices with a template and telling them to ignore all the details within it. But they will worry about it; it is very disconcerting to be working with something that is not fully understood. Worse, they may be tempted to look at the template; they may tinker with it and cause all sorts of errors in all sorts of unexpected places. This approach is a recipe for misery and disaster.

These details of syntax are aside from the fact that the Python program is at most two lines long while the C++ comes in at a comparatively hefty seven. It is not unreasonable to suppose that the practical hurdles of creating and editing are more difficult for the novice when the program is longer.

Happily, in C++ it is possible to write this program without understanding anything about objects. Not so in Java! Here the novice must understand at least something about objects and methods and how they are declared and used. Some idea of command-line arguments would also help. One version of the program might resemble:

```
public class Hello {
    public static void main (String args[])
    {
        System.out.println ("hello, world");
    }
}
```

This is roughly the same length as the C++, but there remains roughly the same number of concepts to understand. The best that can be said about the Java in comparison to the C++ is that there is no need to include a library and at least `System.out.println` is less cryptic than `cout`!

But this is not a *proper* Java program. There are no objects. Wars have of course been fought in true

Lilliputian style over whether to teach objects first, last (or, bizarrely, not at all) in Java. The "objects-laster" would join the "objects-not-at-all" in being quite happy with this version of the program. The "objects-laster" would, however, cringe and implement a `HelloWorld` object:

```
public class HelloWorld {
    public HelloWorld ()
    {
        System.out.println ("hello, world");
    }

    public static void main (String args[])
    {
        HelloWorld hw = new HelloWorld ();
    }
}
```

Quite. We are now up to eleven lines. We have a constructor, another new concept.

Brevity of a program is not everything, but it is important. Novices really do struggle with the basic mechanical details of creating their programs, and it must be easier to create a program of two lines than a program of seven. But there is more to it than brevity. Python's significant advantage here is the number of concepts that must be understood before a program can be written. The number is still significant, and would occupy several lectures, but it is so much smaller than the number that is needed in order to write an equivalent trivial program in C++ or Java. Python is object-oriented, but we can write a "hello, world" program without creating an object!

Some More Features

A full discussion of the features of Python is way beyond the scope of this paper. Suffice to say that it is a fully featured and fully object-oriented language, supporting all the things that would be expected from such a language. It is written in C, and inherits much from that language (and so also has much in common with C++ and Java). In use it feels rather like Perl, although the syntax is thankfully much less opaque.

There are some other potentially interesting (and idiosyncratic) features of Python that hold interest for a teacher, and that are worth discussing in a little more detail. Firstly it is important to appreciate that Python is interactive. It is possible to use the interactive Python prompt to try out program fragments, to find the values of expressions, or for more general experimentation. This makes the language less of a "language" and more of an "environment" and makes the whole programming experience more interactive and accessible. It also means that a partially correct program will generally partially run; Python is interpreted, and the part of the program up to an error will often execute. Students like this; it is less "black and white" than just having a program that simply will not compile.

The first specific detail worthy of a mention here is indentation. Students of programming do not, of course, indent their code (unless there are marks for it or death threats are issued). Students of Python programming, however, have no choice since indentation determines the flow of control. They must indent whether or not there are marks available or their lives are in peril. An example:

```
if name == "Arthur":
    print "I am Arthur, King of Britons"
```

The indentation (it does not matter how much the indent is) indicates that the `print` statement is inside the conditional statement. Contrast this with:

```
if name == "Robin":
print "Boldly Run Away!"
```

where the statement is outside the conditional statement and would always be executed. It is perhaps also worth noting in passing that this code fragment is illegal since there is nothing to execute inside the conditional statement; the Python interpreter would report an error when the statement was reached.

The rules for indentation are as would be expected. A new block requires further indentation, and the block is only ended when the indentation ends. Moreover it is not possible to indent incorrectly; if the indentation does not show a logical program flow an error message is generated.

This may seem a trivial issue, but to anyone accustomed to trying, often in vain, to persuade students to properly and consistently indent their code, this must at least be interesting.

Another feature is Python's ability to quickly and easily create objects. In C++ or Java this requires a fair amount of coding before any progress can be made but in Python it is sufficient to declare that a class exists:

```
class Knight:
    pass
```

(the `pass` statement here is simply fulfilling the requirement that a block must have a statement inside it, the statement itself does nothing). An object can be created:

```
galahad = Knight ()
```

This object can be given attributes. Variables in Python are declared implicitly by their first use, and are given a type determined by their initial value; the same applies to attributes.

```
galahad.favouriteColour = "Blue"
galahad.favouriteBird = "Swallow"
```

Since Python is also interactive, this provides a fine environment in which a student can experiment with objects and their attributes.

With these features, and considerable commercial use, Python must surely have something to offer as an introductory language.

It was time to find out.

Python in Practice

The first step in evaluating the teaching potential of a new language is obviously to try it out, and an opportunity was available at the start of the 2002/03 session. Changing language completely would be a significant risk (and would have required a language war), so this was not possible. What was possible was to introduce some Python as an introductory part of a programming course before the students moved on to their intended final language (as it happens, C++).

The School of Computing is a large department, providing three single-subject degree programmes and contributing to many joint-honours variants. In 2002/03 there were approximately 300 students in the class for introductory programming, coming from a wide range of degree programmes and a wide range of levels of programming experience. Students who already had significant programming experience were given the opportunity to follow a different scheme if they wished, but the main class still presented the usual range of experience and interest. A free textbook ([Downey et al's excellent text](#)) was chosen and made available at a cost to cover copying; virtually all the students invested.

Roughly the first seven weeks of the course (which covered approximately twenty weeks in total) were

devoted to Python. It was possible in this time to cover all the main issues of procedural programming; objects were left out for simplicity but they could have been covered from the start in a more complete course. The syllabus built up the programming concepts in the usual way, starting with the basics of variables and values and moving through the various control statements. These seven weeks provided a reasonably coherent subset of Python.

This seemed to go quite well. Most students were quickly able to write simple programs, and many were soon happily experimenting with other features of the language. The power of the language means that many were soon writing programs that were reading and sorting collections of data, something that would normally have been a topic near the end of the C++ course. There was ample anecdotal evidence that many were achieving results way beyond what they might have been achieving at the same time with C++.

After the seven weeks, the class moved on to C++. Teaching this was easier than in previous years since all the students now had at least some programming background, and the students seemed to struggle less than in previous years. A problem arose when it became apparent there were a few cases of "negative transfer" where a concept in Python failed to map directly to the equivalent in C++. Interestingly, in most cases it seemed to be Python that was the more intuitive. Two specific examples of this follow.

In Python, the `*` operator is overloaded for repetition of strings. So this:

```
print "Ni!" * 3
```

produces the expected:

```
Ni!Ni!Ni!
```

There is no C++ equivalent to this; the syntax to achieve the same thing is fairly horrible.

Secondly, Python has features that allow variables to loop across a range, so that this:

```
for c in "African Swallow":  
    print c
```

functions as a `for` loop moving across the characters in the string. There is no C++ equivalent to this without using more complex loops, and without understanding index values and something about the inner workings of strings and arrays.

In many cases the students reported that they were feeling restricted by C++. It was certainly rather difficult to explain to them why the more complicated C++ way of achieving some result was the right way to do something!

It would have been a shame to leave Python behind after the introductory part of the course, and so a small group of students was recruited to take their studies further. This group was randomly chosen from a group of volunteers and represented a range of experience from complete novices to competent C programmers. They attended an extra tutorial each week, and were bribed to ensure attendance.

There were two aims to the tutorial. The first was to cover (in a fairly formal "lecture" way) the equivalent Python to the C++ that was being considered in the main lectures. Effectively this meant discussing the facilities in Python for defining classes (as opposed to individual objects) and issues such as inheritance, overloading and polymorphism. At the same time the students were asked to reflect on their experience of learning to program, and to try to reflect on whether others would find Python an easy language to learn.

It became apparent straight away that the Python versions of the C++ material required in some sense "less teaching". There was less to explain, and certainly fewer intricate details to master. The additional Python

class was in fact soon well ahead of the main class in terms of the material covered.

The facility to define objects and to experiment with objects, adding attributes as required, was found very useful and several students reported that this gave them an inkling of what an object really was.

The students' views at the end of their course were favourable towards the use of Python. Some even went so far as to question strongly why they had been forced to endure C++. Most had found Python useful as a tool for prototyping their C++ work. The omens were favourable.

Conclusions

This paper presents the experience of a single, informal, experiment in deploying Python as a first programming language. No attempt is being made here to claim that Python is an ideal, or even an appropriate, first language. I do not want a language war; the loudest professors always win a language war.

No claim is made that a strictly scientific formal evaluation has taken place. That said, the experiences of this study do seem to indicate that it has something to offer, and that it should be evaluated further. Obviously a more in-depth study at a range of institutions would be needed.

Students have all sorts of issues to grapple with as they attempt to learn to program. The language that they use is only one of these, but it is one over which teachers can (and should) exercise control. There is, perhaps, the temptation to teach whatever appears to be the flavour of the month in the job sections or whatever appears to look good in the prospectus. But this is wrong.

None of the students in the Python tutorials said that the language they would be taught had had any influence on their choice of course. They expressed a vague preference for "something we might have heard of", but nothing more.

It is time to take another look at what we teach. Python might, perhaps, be something we should be teaching. It might also be something that the students want to learn.

A final anecdotal note. The final coursework for the C++ course was a reasonably complex task involving a class simulating an animal tracking application. I wrote the model solution in C++; it took about two hours. One of the students decided to write it in Python instead; it took him less than an hour. There *has* to be something in that.

Acknowledgments

This work was supported by the LTSN-ICS Development Fund and the Teaching Quality Enhancement Fund at the University of Leeds. Thanks are due to all the students who took part, and in particular the small group who volunteered for an extra hour or programming each week.

More Details

More details of this project, including copies of the resources used to teach Python in this course and further details of the students' views can be found on-line at <http://www.comp.leeds.ac.uk/tony/python/>.

A Note on Naming

Python is named after "Monty Python's Flying Circus". It is usual when writing about Python to use examples drawn from the Python oeuvre.

References

- Downey, A, Elkner, J, Meyers, C. How to Think Like a Computer Scientist: Learning with Python. <http://www.ibiblio.org/obp/thinkCSPy/>.
- Gault, A. Learn to Program Using Python. Addison Wesley Professional, 2001.
- Kölling, M. BlueJ - Teaching Java. <http://www.bluej.org/>.
- van Rossum. G. Python Language Website. <http://www.python.org/>.
- Wirth, N. The Programming Language Pascal. Acta Informatica, Vol. 1, pp 35-63, 1971.

Author Details

Tony Jenkins is a Senior Teaching Fellow in the School of Computing at the University of Leeds. He has experience in the IT industry of programming in more languages for more years than he cares to remember. He likes COBOL, and thinks there should be more of it about. Tony has been teaching introductory programming in Pascal and more recently C++ at Leeds for the last decade. He has recently been forced to start teaching Java (which is at least not as bad as C++).

Tony's special area of responsibility and interest is the teaching of programming to those students who enter Higher Education with no previous experience. He has some unusual ideas about approaches to teaching these students and has presented papers at international conferences on the subject. Other research interests include the affect of gender in introductory programming, and issues surrounding motivation and retention. The University of Kent awarded Tony an MSc in 2001 for research into the motivation of students of programming. His first book, on programming in C++, was published in April 2003, and the Java version will follow in 2004.

Tony thinks that teaching introductory programming is the best job in the world, and can't quite understand why more people don't want to do it. He is also a firm believer that students should have fun while they do it.

Valid