Teaching scientific programming using Python

Michael Williams

December 3, 2002

michael.williams@st-annes.oxford.ac.uk

Copyright © 2002 Michael Williams; this document may be copied, distributed and/or modified under certain conditions, but it comes WITHOUT ANY WARRANTY; see the Design Science License for more details.

A copy of the license is:

- included in the LaTeX distribution of this document—see http://users.ox.ac.uk/ sann1276/python.
- always available at http://dsl.org/copyleft/dsl.txt.

Abstract

Python is a modern, object-oriented programming language with clean, readable syntax. Its design was informed by experiences with other teaching languages so it is considered suitable for such use. The current undergraduate-level Physics course teaches computer programming using Pascal. This report discusses the possibility of replacing this with Python.

A Python equivalent of the current Pascal handbook was designed and a group of volunteer students took the course. Although several problems with the course and Python itself were discovered, I conclude that Python's superior readability makes it a more suitable introductory language than C and it's real world applicability make it preferable to Pascal.

CONTENTS

1 Introduction **3**
1.1 The current course . . . 3
1.2 Python . . . 5
1.2.1 What is Python? . . . 5
1.2.2 Python's design philosophy . . . 6
1.2.3 Project rationale . . . 7

2 The Python trial **11**
2.1 The course handbook . . . 11
2.2 The trial . . . 12
2.3 Software . . . 12
2.3.1 Python . . . 12
2.3.2 IDLEfork . . . 13
2.3.3 Numeric Python . . . 13
2.3.4 Gnuplot . . . 13
2.4 Questionnaire . . . 14

3 Results **15**
3.1 Questionnaire results: All students . . . 15
3.2 Questionnaire results: Categorised by ability . . . 21

4 Discussion **27**
4.1 Aims . . . 27
4.2 Methodology . . . 27
4.3 Is Python's use feasible? . . . 28
4.3.1 Portability . . . 28
4.3.2 Speed with which it can be taught . . . 28
4.3.3 Demonstrators . . . 29
4.3.4 Integrated Development Environment . . . 29
4.3.5 Teaching of generic programming concepts . . . 30
4.4 Is Python preferable? . . . 31
4.4.1 Readability and the speed with which Python can be learnt . . . 31
4.4.2 Python's use outside Education . . . 32
4.4.3 Python's peculiarities . . . 33
4.5 Trial experiences and questionnaire results . . . 34
4.5.1 Indentation a block-delimiter . . . 34
4.5.2 Error messages . . . 35
4.5.3 Syntax errors . . . 36
4.5.4 Arrays . . . 36

4.5.5 Reading and writing files . . . 37
4.5.6 The range function and fencepost errors . . . 38
4.5.7 The interactive interpreter . . . 39
4.5.8 Python compared to Pascal . . . 40
4.6 Conclusion . . . 40

A The questionnaire **45**

B Fixing IDLEfork **47**

List of Abbreviations

ANSI	American National Standards Institute
API	Application Program Interface
FAQ	Frequently Asked Questions
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
OO	Object-oriented
OOP	Object-oriented programming
OS	Operating System
PDF	Portable Document Format
SIG	Special Interest Group
VHLL	Very High Level Language

CHAPTER

ONE

Introduction

1.1 The current course

The University of Oxford's Physics department is one of the largest Physics departments in the United Kingdom. Each October it admits around two-hundred students to study for either the 3-Year BA or the 4-Year MPhys. Both groups of students follow the same course until late in the third year.

The students are perhaps amongst the most able produced by British and international schools. Typical entrance requirements are three 'A' grades at A Level, which must include Mathematics and Physics. This enables all aspects of the first-year course to assume familiarity with and aptitude for school-level mathematics (in particular algebra and calculus). In fact many students arrive with more advanced qualifications, such as the A Level in Further Mathematics.

However, it is not possible to assume any experience with computers. When the Pascal programming course was first introduced ten years ago, most students had not used a computer before coming to university. This is no longer the case; the majority arrive with a familiarity with, almost invariably, the Microsoft Windows environment. Very few have used Unix systems. A minority arrive with programming experience.

The first year of the Physics course involves three main components: physics (mechanics and special relativity, waves and optics, and electronics and electromagnetism), mathematics, and the practical course. The physics and mathematics are taught using a combination of lectures and tutorials, and are examined at the end of the first year and again towards the end of the third year[1].

The introduction to computer programming forms a compulsory element of the practical course. During the first year students must attend the various laboratories for a total of sixteen days, two of which are allocated to the teaching of programming[1]. In the computing course's current form the first day is spent working through a handbook which aims

[1] The time allocated to programming may be increased to four days for the 2002-3 academic year. This report assumes a two-day course.

to provide an introduction to the environment and sufficient exercises such that they are able to write short, procedural programs in Pascal[2].

Students are then asked to solve a more substantial computational problem on the second day. These problems are representative of the requirements and algorithms most often used in the numerical solution of various physical problems. Each student is assigned a problem by a laboratory supervisor. CO11 is ostensibly the easiest, whereas CO16 requires considerable thought[3].

The Pascal course is taught using terminals running NeXTStep, a variant of the Unix operating system with an integrated and consistent graphical interface. These are ageing machines so they are being replaced. Over the last year a gradual substitution of NeXTs with SunRays has taken place. SunRays are terminals connected to a server running Solaris 5.7, which is also a Unix variant (although without such an integrated graphical environment)

Although Pascal is a commonly used teaching language, it is widely acknowledged as being little more than a toy language. Its deep-rooted problems were described by Brian Kernighan in his paper ”Why Pascal is not my favorite programming language”[4].

Its decline as was slowed somewhat by the extensions provided by various dialects, but these could not avoid the languages fundamental limitations and also served to balkanize the language’s user base. Other programming language’s have benefited from a respected ANSI/ISO specification (e.g. C) or a dominant commercial implementation (e.g. Java).

Furthermore, although free or cheap Windows compilers are available (such as GNU Pascal) their installation requires considerable computing knowledge, and not a task the inexperienced student is likely to relish.

Pascal’s fate as a serious programming language appears to have been sealed. It is now rarely used, even in academia. Having been the standard language for the “Advanced Placement” college entrance exams in the USA for nearly 20 years, it was replaced by C++ in 1999 (which, incidentally, was almost immediately replaced with Java).

Perhaps Python provides an alternative to Pascal that, as well as being suited to teaching programming fundamentals, is also considered more than a toy by the commercial and academic world.

1.2 Python

1.2.1 What is Python?

The “What is Python?” page on http://www.python.org describes Python as:

> “an *interpreted*, *interactive*, *object-oriented* programming language. It is often compared to Tcl, Perl, Scheme or Java.” [their emphasis]

By *interpreted* it is meant each time a program is run the *interpreter* checks through the code for errors and the interprets the instructions into machine-readable “bytecode”. Perl is perhaps the best known example of an interpreted language.

This is different to a compiled language (such as C) which is compiled only once and produces a binary executable which can then be run again and again, even on different physical systems of the same architecture[2]. This means programs written in Python generally run more slowly than programs written in C. Run-time speed reductions ranging from factors of two to an order of magnitude or more are typical. However, the compile-debug cycle and prototyping of algorithms is much quicker an interpreted language such as Python. In general programs may be written in less time in Python but will take longer to run.

Python is *interactive*. In one mode of operation Python is used in much the same way as the Unix command line; a Python statement is entered, followed by the `Return` key and, if appropriate, the result is printed on the screen. This is particularly advantageous in the debugging process, and is the natural way of working in many mathematical programming and visualisation environments used in the physical sciences such as Mathematica, Matlab and IDL.

Python is *object-oriented* by design. Object orientation provides a powerful way of abstracting the data structures and flow of programs. It is not yet used as much in the physical sciences (and may never be), which is partly due to the naturally procedural way of thinking about algorithms, and partly to a culturally entrenched way of programming, originating in the days before OOP.

C and Fortran are perhaps the most popular procedural languages in academia. C++, Perl and Java are the most widely used OO languages. However, it is worth noting that, of these, only Java was explicitly designed as an OO language. The OO components of both C++ and Perl take the form of extensions to a underlying procedural structure. Sometimes this leads to a confusing implementation, particularly in C++, which is often dismissed as too baroque for non-experts.

Python is *portable*. The Python interpreter can be installed on any architecture for which there exists a C compiler. In particular it can run on the Sparc-based SunRay system in the laboratory, and on Intel-based PCs running both

[2] However, each time the program is changed it must be recompiled.

Language	Search results returned
C	4,930,000
Java	2,670,000
C++	1,800,000
Perl	1,690,000
Python	838,000
Pascal	652,000
LISP/Scheme	629,000
Fortran	602,000

Table 1.1: Searches for language name AND either "code" OR "language" OR "program" in order to get meaningful results from searches for "C" and "Python", both of which have other meanings. http://www.google.com accessed 3/6/2002.

Microsoft Windows and Linux. Furthermore, because it is interpreted, programs written on one architecture will generally run on another[3]. This allows students to write programs on their own PCs[4] and bring them into the department to show to a demonstrator. The interpreter is available pre-compiled, protecting students from the difficulties of building a large application from source code. The Windows binaries are particularly simple to install; they come in the form of a GUI installer and their installation consists of little more than clicking "Next" a few times.

Furthermore, Python is Free. Releases on all platforms are available free of charge (although as with many open source projects, it is possible to pay for a customised commercial distribution) and the source code is open-source.

In an attempt to measure the popularity of various programming languages, I entered their names into the Google search engine, and tabulated the number of results (see Table 1.1). Obviously, this is not a very scientific measurement, but does offer some indication of Python's relative popularity.

1.2.2 Python's design philosophy

There is often a disparity between the *aim* of a piece of software and what it is actually capable of. Nevertheless, it is instructive to consider the aims of Python's author in creating and maintaining it.

Python's original author and its current principal maintainer is Guido van Rossum[5]. Python is based on a teaching language called ABC that was in use in the 1980s. Van Rossum created Python with the aim of removing the restriction that prevented ABC's more widespread use: its inextensibility. ABC was like Pascal: a good teaching language, but relatively useless outside simple classroom applications.

So Python was designed to be suitable for educational use from the ground up, but with more advanced programming constructs, a feature-packed standard library, and the ability to interface easily with programs written in other languages (especially C) so it could be used outside education.

[3]Unless they use the OS-specific libraries such as those which change file permissions and access system variables.
[4]Ownership of computers amongst undergraduates is increasingly common.

It has always been Van Rossum's desire to see Python used for teaching, and he has taken a particular interest in this project. I have received several emails from him offering solutions to specific problems encountered and more often general advice. The Python FAQ entry "Is Python a good language in a class for beginning programmers?"[5] goes to great lengths to justify Python's suitability as a teaching language. It's claims are discussed in Chapter 4, "Discussion".

Python is still relatively little-used in education. However, there is an active SIG mailing list and web page that discusses its use and attempts to coordinate the teaching programs that have begun to appear over the last few years[6].

Python's success is commonly attributed to its standard library; it is often described as coming with "batteries included". The library allows the rapid creation of relatively complex and highly modularised programs. Python is often seen as a direct competitor to Perl, which is most commonly used for systems administration tasks, generating dynamic web pages and other string parsing and manipulation tasks. Because of this Python is sometimes described as a "scripting language", although this term is often used disparagingly, and Python's supporters attempt to avoid it.

Python programs are syntactically extremely clean and consistent. The syntactic structure of the language was designed with great care. Contrast this with Perl, which is notoriously difficult to read.

For most experienced programmers, the striking visual difference between Python and almost any other language is the use of indentation. Code blocks such as those following `if` statements are delimited by their indentation, and not by, e.g. `BEGIN` and `END` statements or braces:

```
print "This program prints the integers from one to ten..."
while i < 10:
    print i
print "Done"
```

1.2.3 Project rationale

In my second year as an undergraduate I chose to do some computing problems to contribute towards the required laboratory work. With my thoughts turning towards what I would be doing after I had finished my degree, I decided to learn a language that would be more useful on a CV than Pascal. The natural choice was C, the *lingua franca* of computing. I eventually completed a 250 line program to solve the Korteweg-de Vries simultaneous differential equations that describe solitons[5].

Learning C and writing the program was an unpleasant experience. I had to get to grips with the syntactic strictness

[5] The non-linear waves seen ahead of canal barges.

and unfamiliarity of the language, the compile-run-debug cycle, and a relatively tortuous method of generating plots of my results. When the program was finally working I took it into the department to run on the SunRay system, only to find the program crashed when run for reasons still unknown.

It was as I was considering how to tackle another problem that I first heard about Python. Having read some of the hype surrounding it on the web I decided to attempt to use Python to write a program to solve the Lorenz equations of three-dimensional fluid flow using the Runge-Kutta method. I found its syntactic simplicity a breath of fresh air. I learnt as much as I needed of the language in much less time than had taken for C. Its dynamic typing and the lack of true declaration of variables speeded up the process of writing the program. Having completed a program of equivalent complexity in a fraction of the time and only one-hundred lines, I became convinced Python was suitable as the principal teaching language in the department.

With help from Charles Wiles, I designed a replacement for the current Pascal course as partial credit for a Masters degree. Working from the original handbook, written by Clive Rodgers[2], I wrote a Python equivalent. A trial was then conducted to find out whether:

1. The use of Python as the principal teaching language of the department of Physics was *feasible* and;
2. Python's design and implementation made its use *preferable* to other languages, in particular Pascal and C.

In order for Python's use to be considered feasible it must satisfy several requirements:

- It must be capable of running on the Solaris-based SunRay system.
- It must be possible to teach enough of the language in just one day that students are able to implement algorithms of the complexity seen in the current problems CO11 to C016.
- It must be possible for demonstrators unfamiliar with the language to quickly achieve such a level that they can help students debug their programs.
- There must be a simple, ideally graphical, IDE from which it is possible to write, run, and debug programs.
- It must be possible to teach generic programming concepts, rather than skills peculiar to that language.

In order for it to be considered the preferable alternative, there are other criteria it is desirable for a proposed language to fulfil:

- It should be possible for students to write (and understand) basic programs quickly. This would provide them with immediate feedback on their progress and enthuse their interest in computer programming.

- Although the course is designed to teach programming, rather than a particular language, that does not mean the language should be inapplicable outside an artificial education environment. Ideally a language should be usable in many fields, particularly Physics. This would provide students with experience in a language they could use later in their careers.
- As well as it being *possible* for a language to be used in real-world applications, the language should already be in *actual* use.
- It should be Free (as in both cost and source code) and available on platforms other than Solaris (in particular Windows and Linux).
- It should not have any unnecessarily complicated constructs whose use is required to undertake common programming tasks.

CHAPTER

TWO

The Python trial

2.1 The course handbook

The current undergraduate programming course, as discussed in Chapter 1, is implemented in Pascal. A successful handbook for the Pascal course already exists. It takes students from the basics of variables to the complexities of procedural programming and arrays. Although Python is more modern in design and syntactically very different, it proved possible to re-implement much of the Pascal course's structure with few problems.

It was decided that the handbook's usefulness to students would be increased by a move from a terse, reference manual to a more verbose textbook-like document. To this end, the amount of explanation in the handbook was increased. The Python handbook used during the trials was of the order of 10,000 words[1].

Due to time limitations, and the previous success of the exercises scattered through the Pascal handbook, there was little change to the exercises within the handbook. Happily, this allowed a determination of whether these extremely simple but common programs could be implemented in Python by students.

The document was formatted using a set of shell and Python scripts maintained by Fred L. Drake, core documentation maintainer for the Python project. These allow a document to be written in LaTeX using a special set of Python-specific markups and to be exported conveniently to either Postscript or PDF for printing, HTML for distribution on the web, perhaps with a view to moving the handbook to an online (rather than printed) resource, and also to plain text.

[1] The Pascal handbook was around 6,000 words.

2.2 The trial

A trial was run in order to determine whether Python's use as the principal undergraduate teaching language was both feasible and preferable to other languages.

The format of the trial was exactly the same as the first-year programming course. The first day was spent going through the handbook, learning the language and environment, and completing the exercises. The second was spent on one of the more substantial problems CO11 to CO16. In practice, many students took less than a day to complete the handbook and move on to the exercise.

From an initial group of seventy-four who expressed an interest in taking part, thirty-three completed the trial.

Several other demonstrators were present to assist me on the busier days. As well as the statistical information produced by the questionnaire, a great deal of anecdotal evidence was provided by talking to the students during the trial and helping them with their problems. These experiences are discussed in Chapter 4, "Discussion".

2.3 Software

2.3.1 Python

The trial was conducted using Python 2.2.1 which, at the time, was the latest stable release. This was built from source for Solaris 5.7 by Charles Wiles.

At the time of writing, Python's division behaviour is changing. The current default is C-like: `1/3` returns `0`, whereas `1.0/3` returns a floating-point approximation of one-third (i.e. 0.3333333`...`). However when the Python version number reaches 3.0 it is its maintainer's intention to change the division operator to behave more intuitively from a mathematical point of view: `1/3` will return the floating-point approximation. The old behaviour will still be available using the `//` operator.

The new division behaviour is clearly more intuitive to physics students, although it does hide some of the details of the internal representation of numbers. It was decided that, as this was going to become the default behaviour anyway and would certainly cause less confusion, the new behaviour would be the default behaviour on the trial system. All programs run through IDLEfork (see below) are passed the `-Qnew` flag, which implements the new division operator.

2.3.2 IDLEfork

The IDE that comes with the Python source distribution is called IDLE. It's feature set is relatively cautiously developed, and is not considered suitable for this environment. Instead we used IDLEfork[7], which has more functionality, and is better layed out. The version used was 0.8.1.

At the time of writing IDLEfork is in active development. It proved impossible to run more than one instance of the program on the same physical machine because of its currently immature implementation of a client-server model. This was a particularly problematic on the SunRay environment used in the department of Physics because, although there are many terminals, there is only one physical machine, making it impossible for more than one student to work at a time.

The IDLEfork developers suggested a workaround that involved dynamically generating a number for IDLEfork to use as its communication port based on process ID (in version 0.8.1 this port was a constant). The port used was then guaranteed to be unique. See Appendix B for further details.

2.3.3 Numeric Python

Python contains many very-high level data types, such as lists and dictionaries. However, it does not contain an array type that behaves in the same way as matrices in mathematics, i.e. operations take place element-wise[2].

The Numeric Python library adds this feature. It is implemented in C for speed and is highly optimised. It can be used as a drop-in replacement for IDL and Matlab in scientific research[8].

The version of Numeric Python used during the trial was 21.0.

2.3.4 Gnuplot

Most of the first-year problems call for some data visualisation. The standard UNIX package for this is Gnuplot. However, it is not particularly user friendly. The Gnuplot.py package was installed to allow students to manipulate Gnuplot from within their Python programs, simplifying the procedure enormously. Gnuplot.py allows the user to plot arrays trivially, or to issue arbitrary and complex Gnuplot commands if required. It offers simplicity to undergraduate students, but could be used for large, scripted data-plotting[9].

The version of Gnuplot used during the trial was 3.7.1. The version of Gnuplot.py was 1.5.

[2] Neither do C or Pascal, but the alternative to using Numeric Python was to use Python's *lists* which are an extremely powerful and flexible multi-element data type. Their elements may be of multiple, arbitrary types (including further lists). It was felt this had the potential to lead to extremely subtle bugs in student programs, and reduced understanding of the principles of arrays as a generic programming concept.

2.4 Questionnaire

The students were asked to complete a questionnaire after they had completed the trial to allow the presentation of quantitative measurements of their opinions of Python.

They were asked to indicate how strongly they agreed or disagreed with a series of contentious statements about Python. Their responses were not anonymous but the questionnaire was not completed in the presence of a demonstrator. The results were cross-referenced with their initial emails indicating their interest in participation, in which they were asked to rate their programming ability from one to ten.

Results

3.1 Questionnaire results: All students

The mean self-assessed programming ability was 4.1 (on a scale of one to ten), with a standard deviation of 2.3.

The results of the questionnaire for the entire sample, regardless of ability constitute figures 3.1 to 3.11.

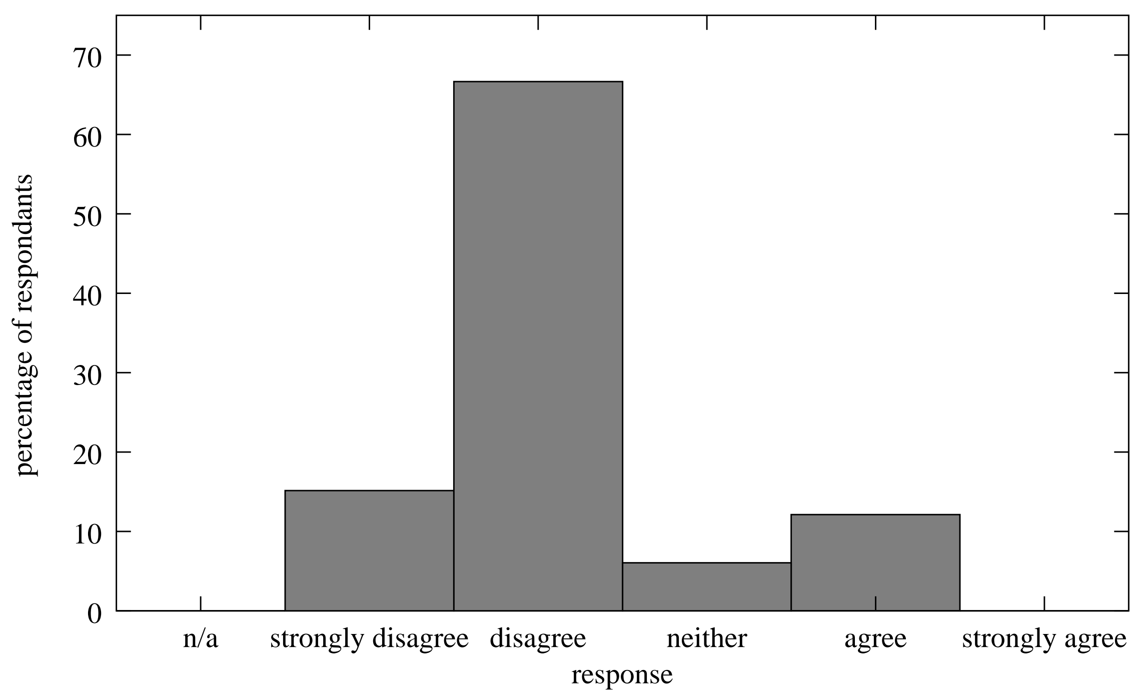

Figure 3.1: Students were asked to say how strongly they agreed or disagreed with the following statements:“I thought the handbook was too long”

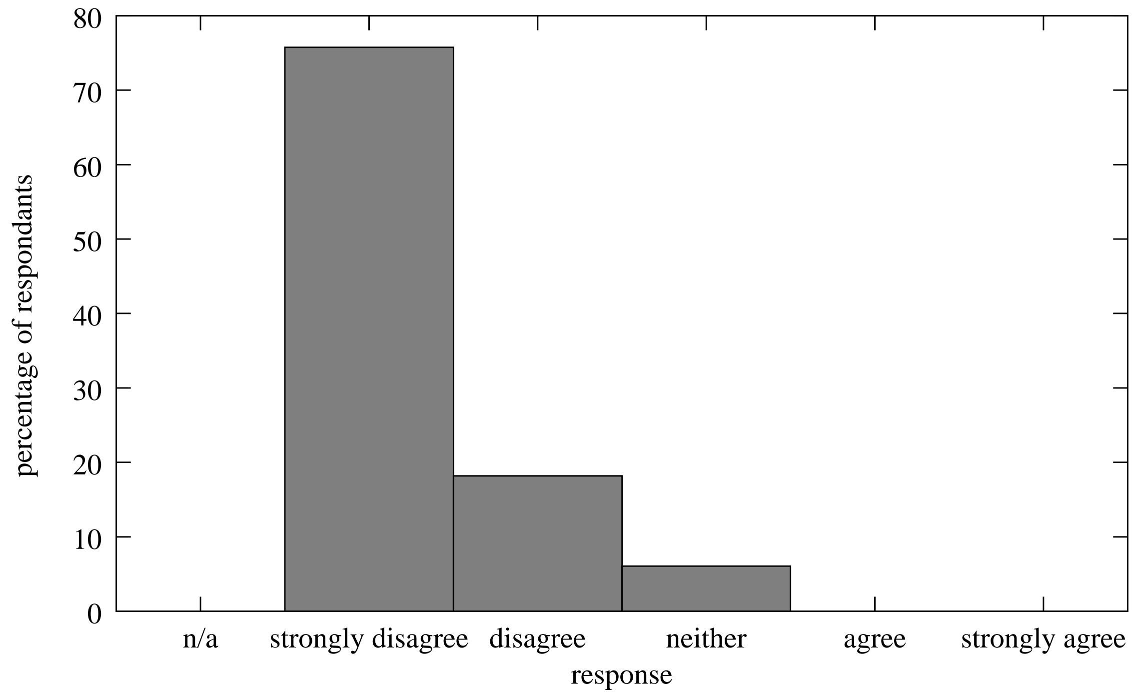

Figure 3.2: “I had trouble indenting my program correctly”

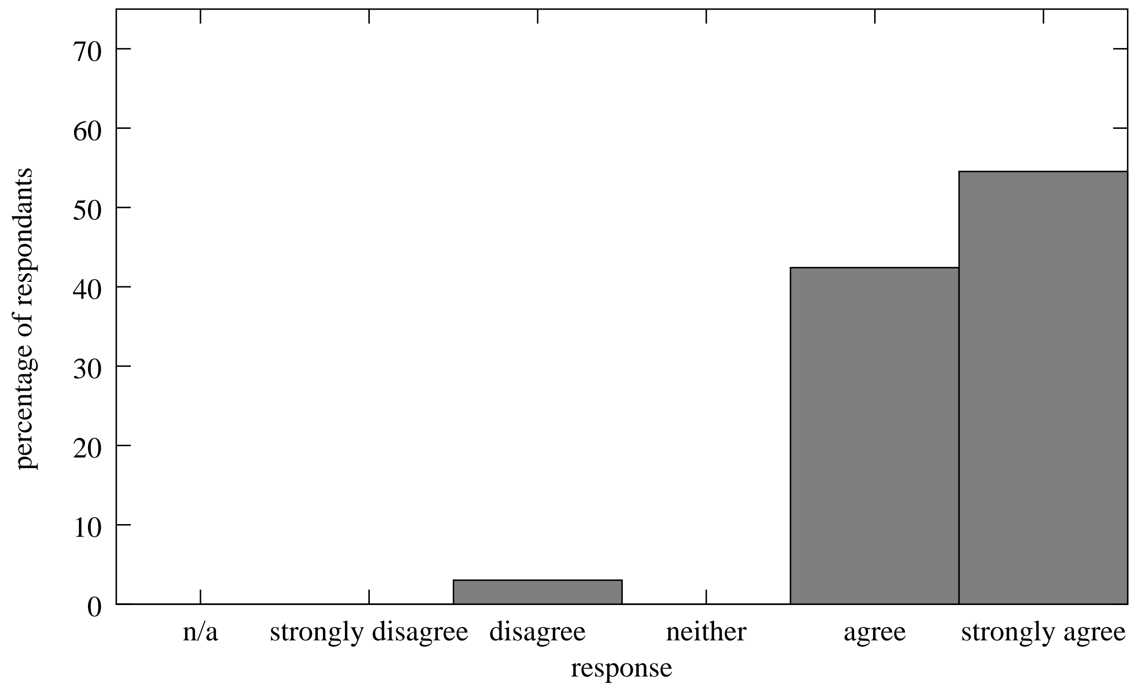

Figure 3.3: “I understood the significance of indentation”

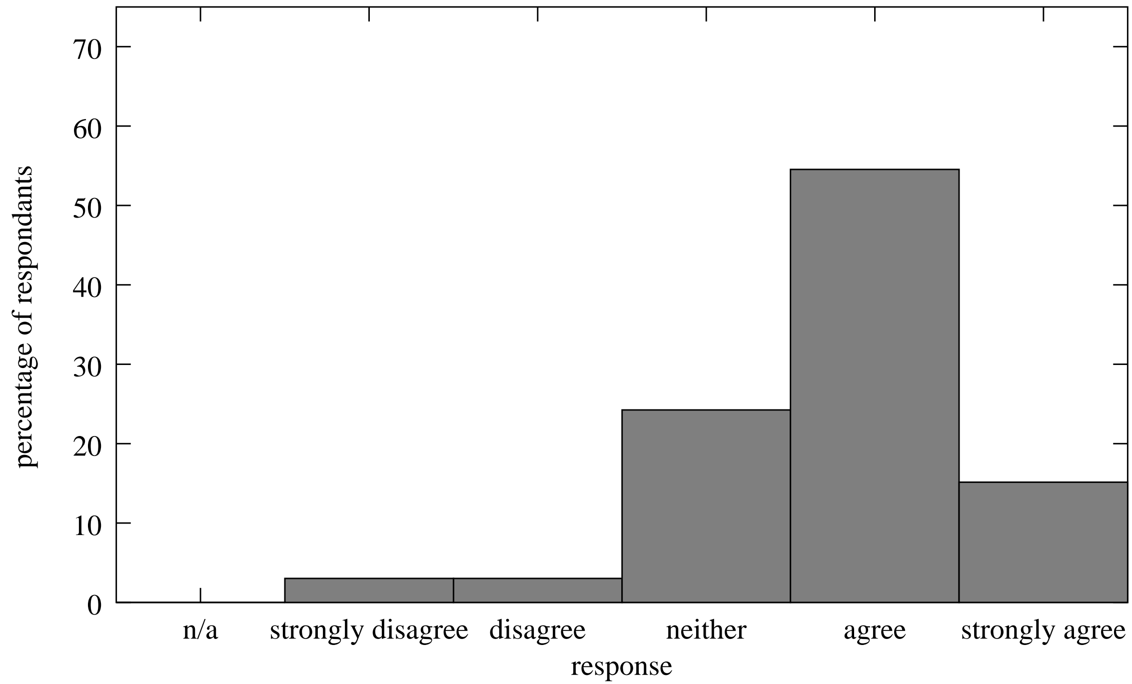

Figure 3.4: “I found the error message sufficiently helpful such that when they arose I could quickly see the problem”

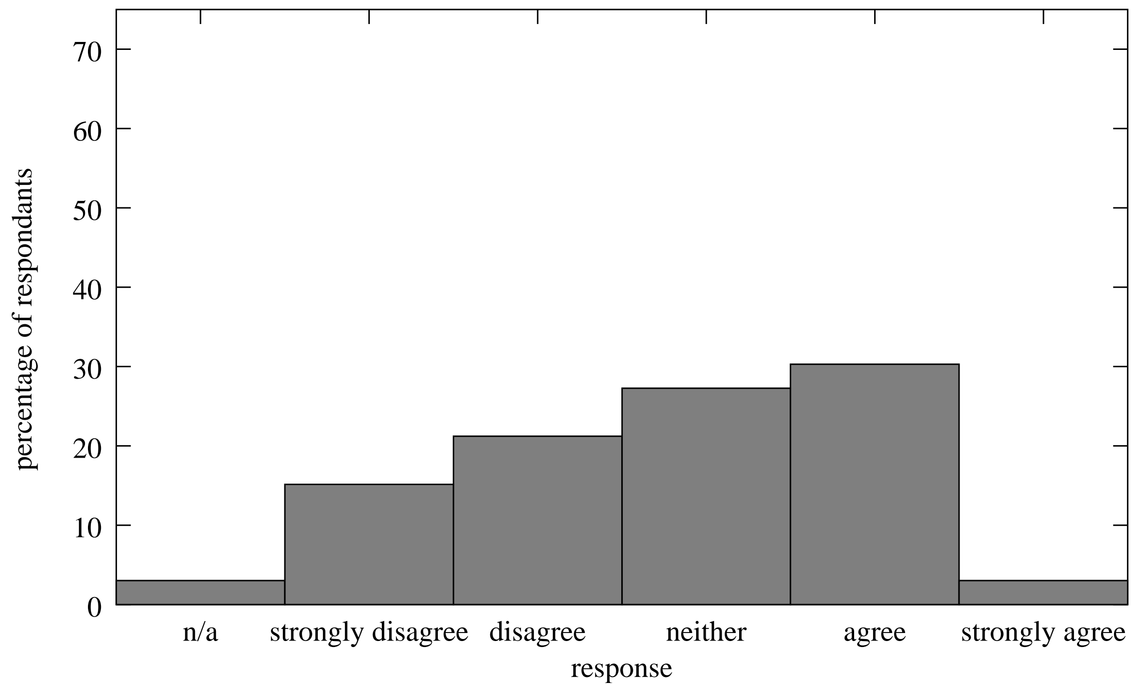

Figure 3.5: “I often made syntax errors like missing colons and brackets”

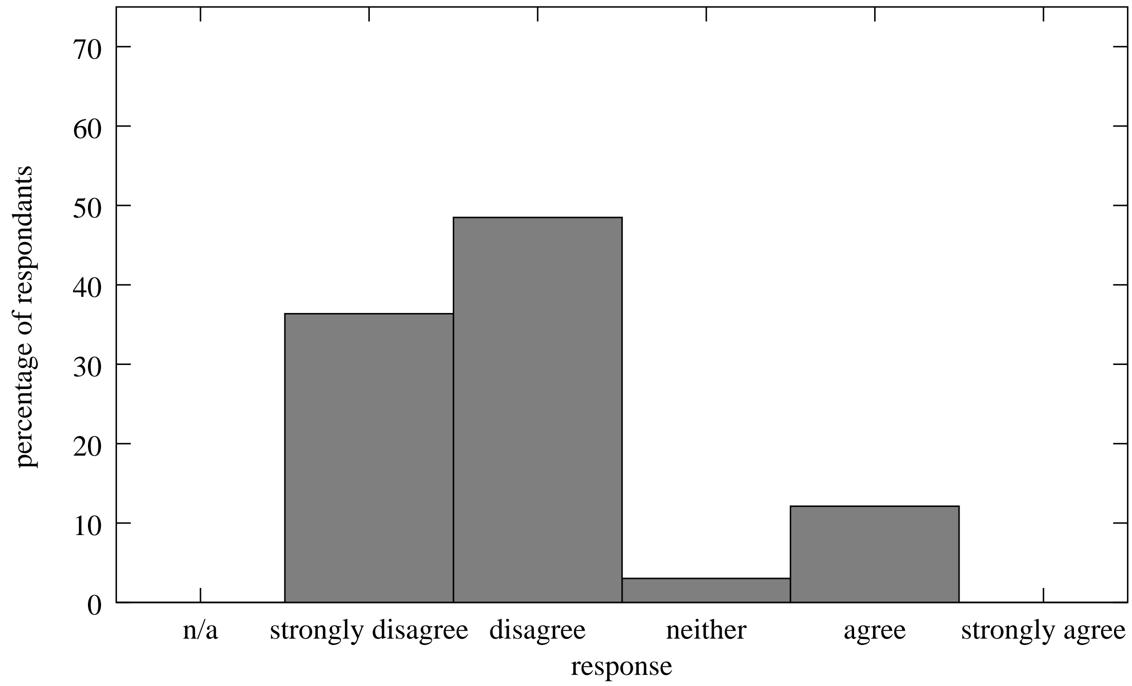

Figure 3.6: “I did not understand the explanation of arrays”

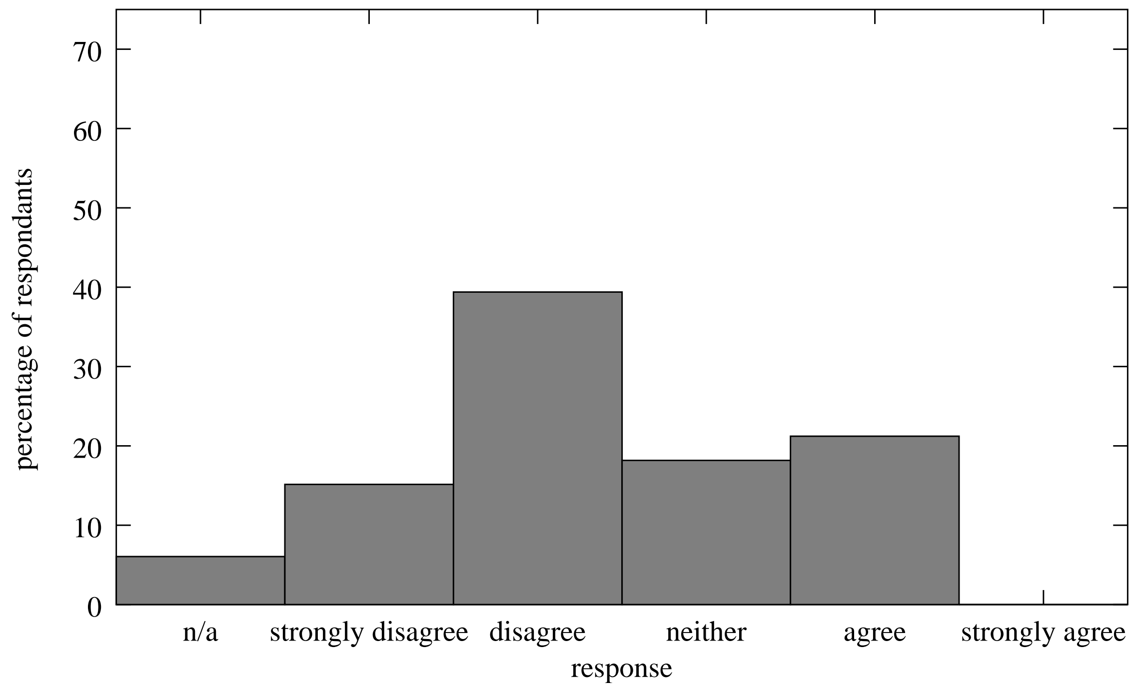

Figure 3.7: “I had trouble *reading from* files”

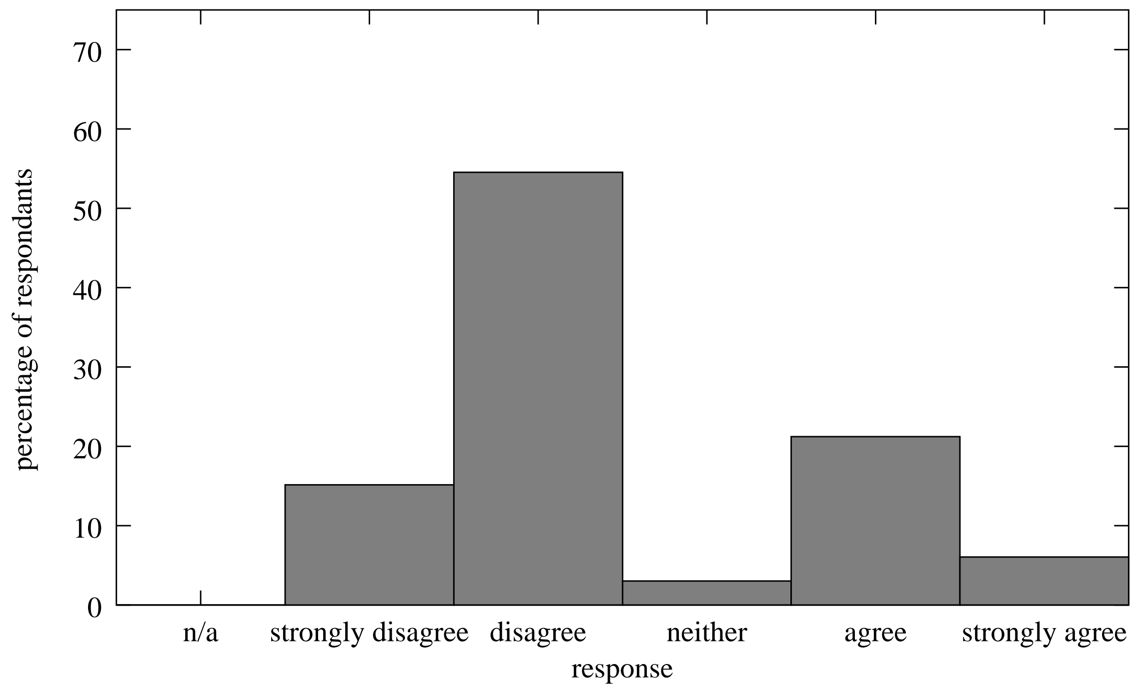

Figure 3.8: “I had trouble *writing* to files”

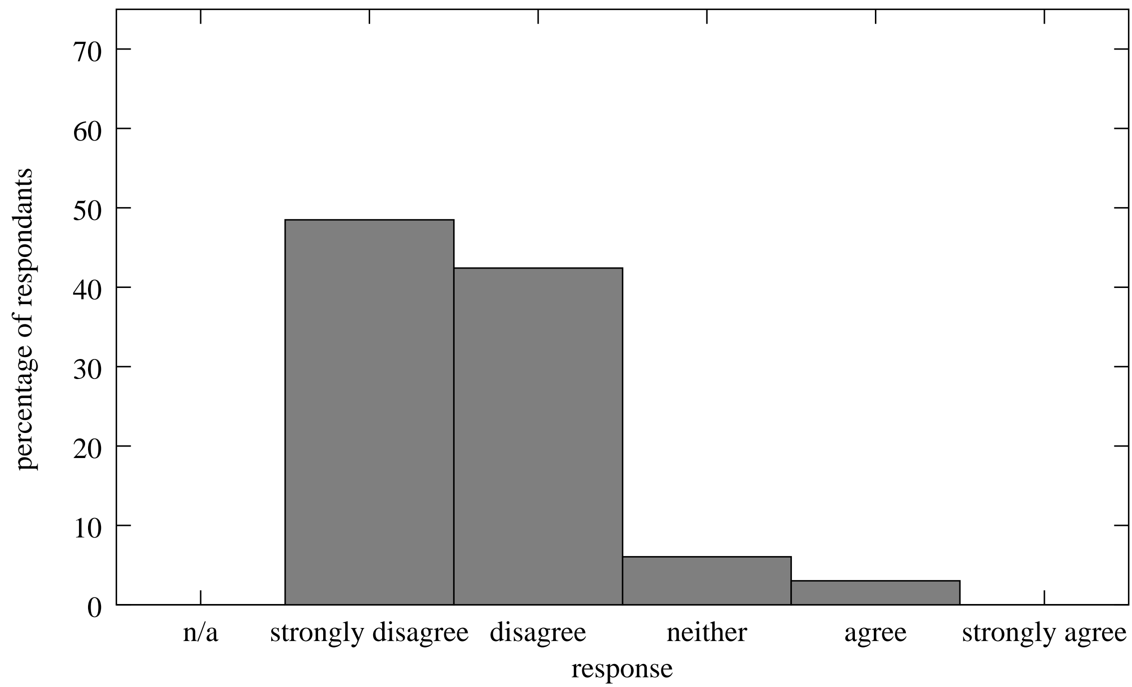

Figure 3.9: “I had trouble with the limits of the range() function”; the range() function is a function that generates lists starting at zero (rather than one) which is a common cause of confusion.

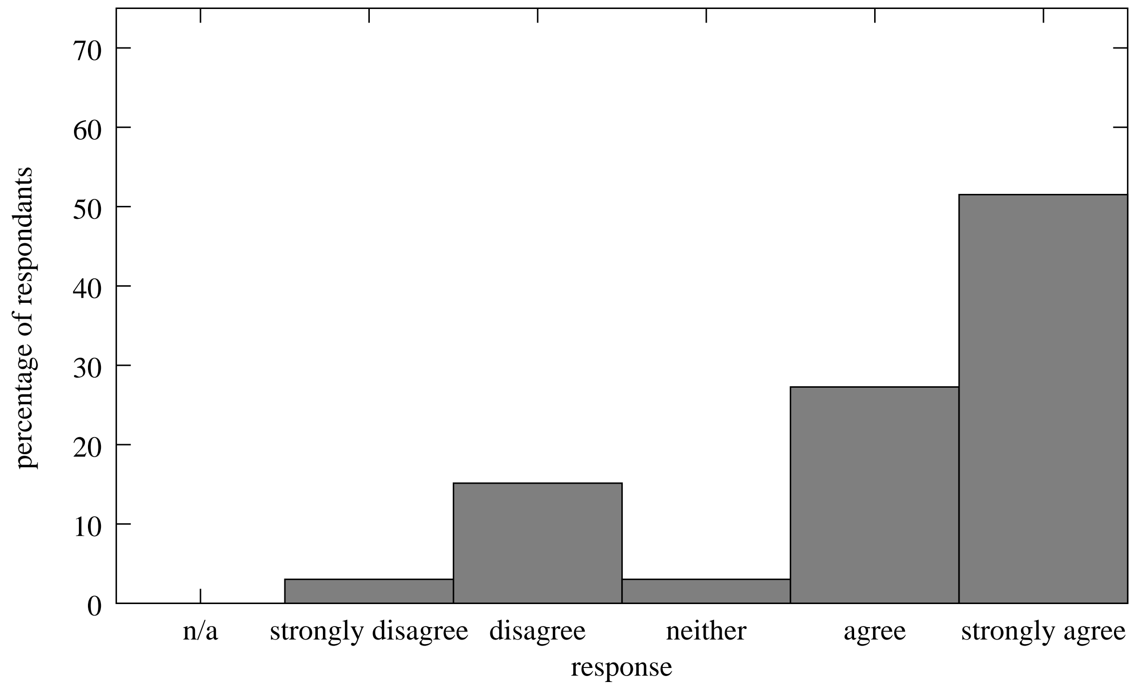

Figure 3.10: “I did not use the interactive interpreter much after the first couple of exercises”

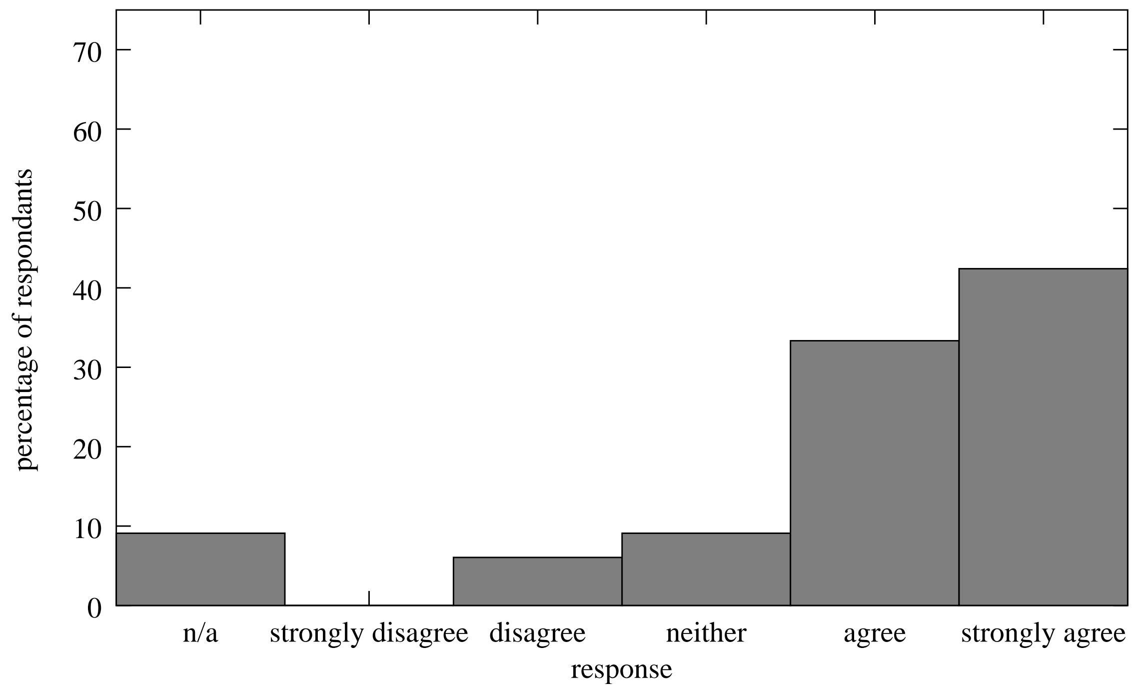

Figure 3.11: “I would rather write the program for the problem I solved in Python than Pascal (put n/a if you can’t remember Pascal!)”

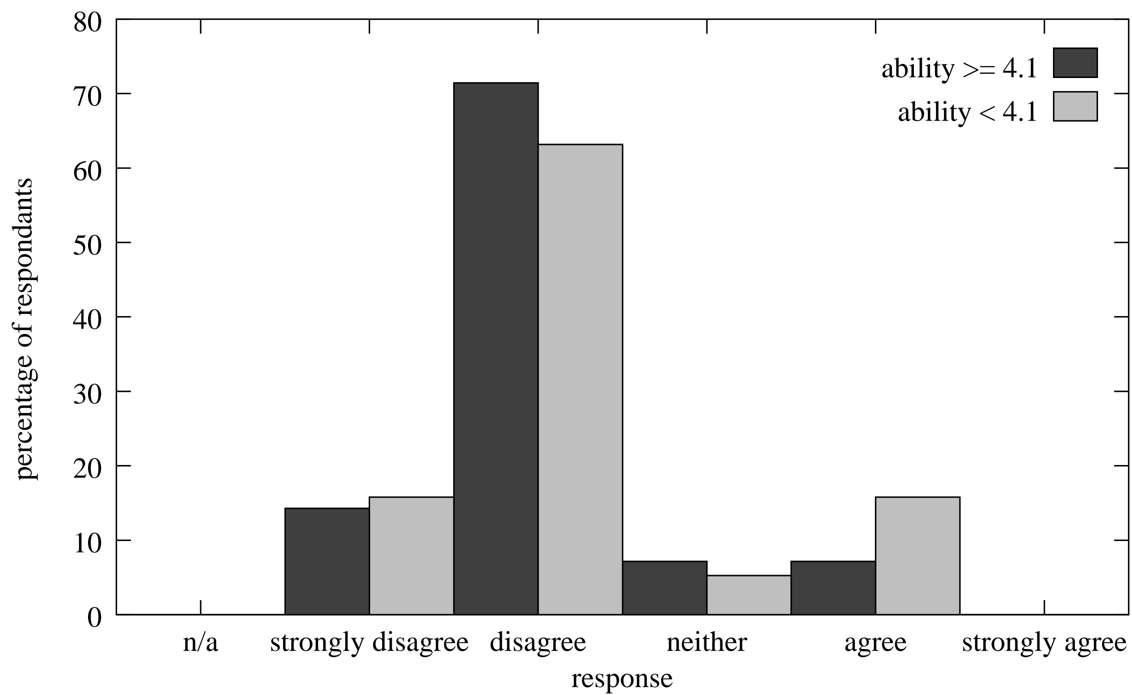

Figure 3.12: Students were asked to say how strongly they agreed or disagreed with the following statements:“I thought the handbook was too long” (low and high ability comparison)

3.2 Questionnaire results: Categorised by ability

The results for the subsets of trial participants with self-assessed abilities of greater than and less than the mean constitute figures 3.12 to 3.22.

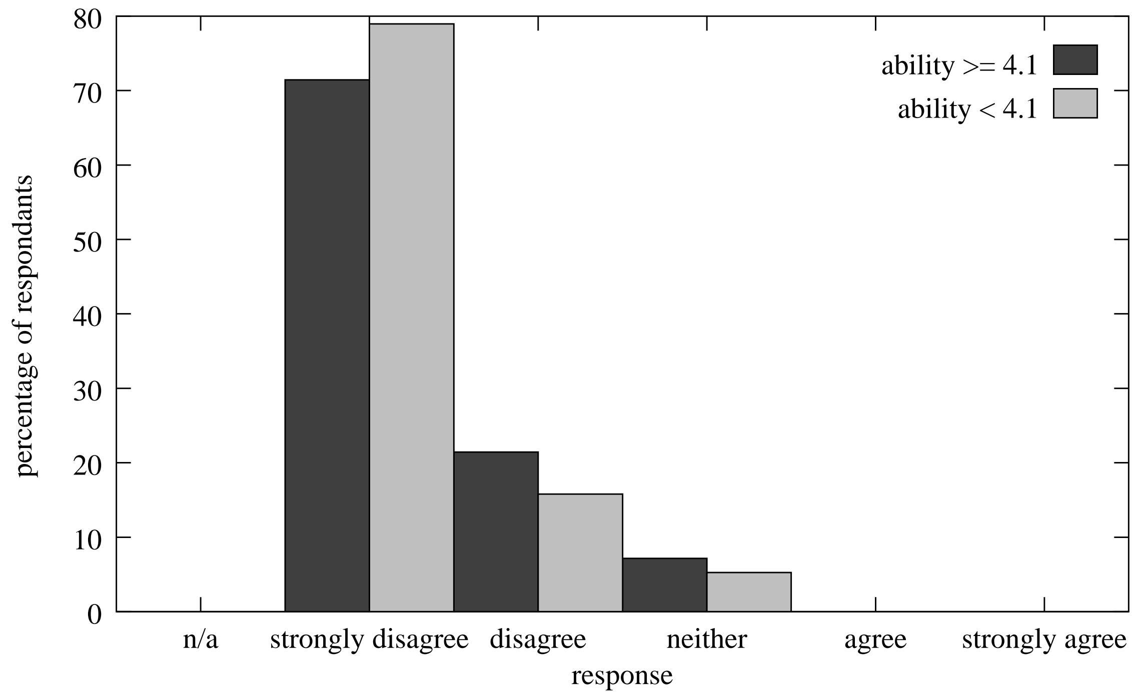

Figure 3.13: “I had trouble indenting my program correctly” (low and high ability comparison)

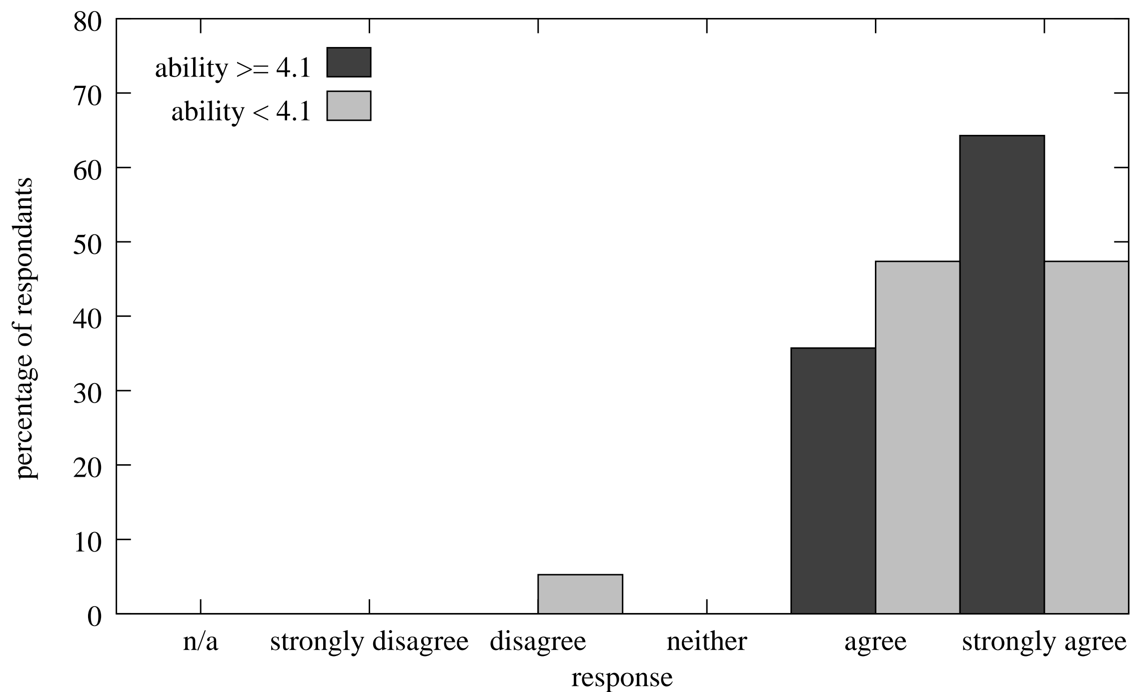

Figure 3.14: “I understood the significance of indentation” (low and high ability comparison)

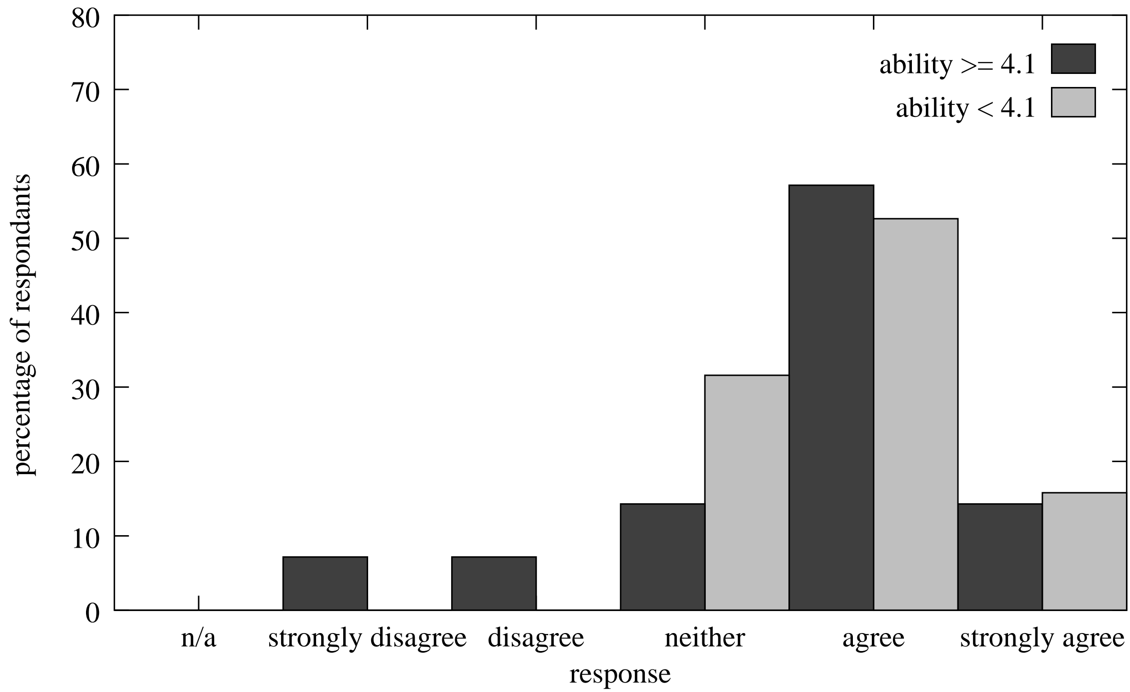

Figure 3.15: “I found the error message sufficiently helpful such that when they arose I could quickly see the problem” (low and high ability comparison)

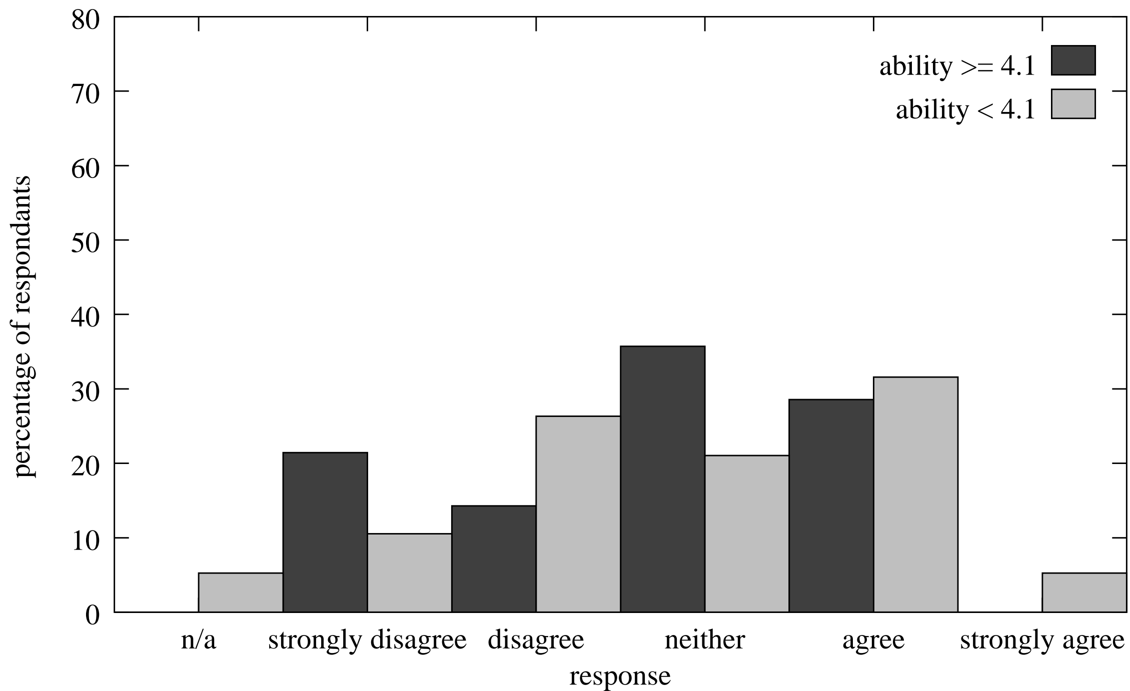

Figure 3.16: “I often made syntax errors like missing colons and brackets” (low and high ability comparison)

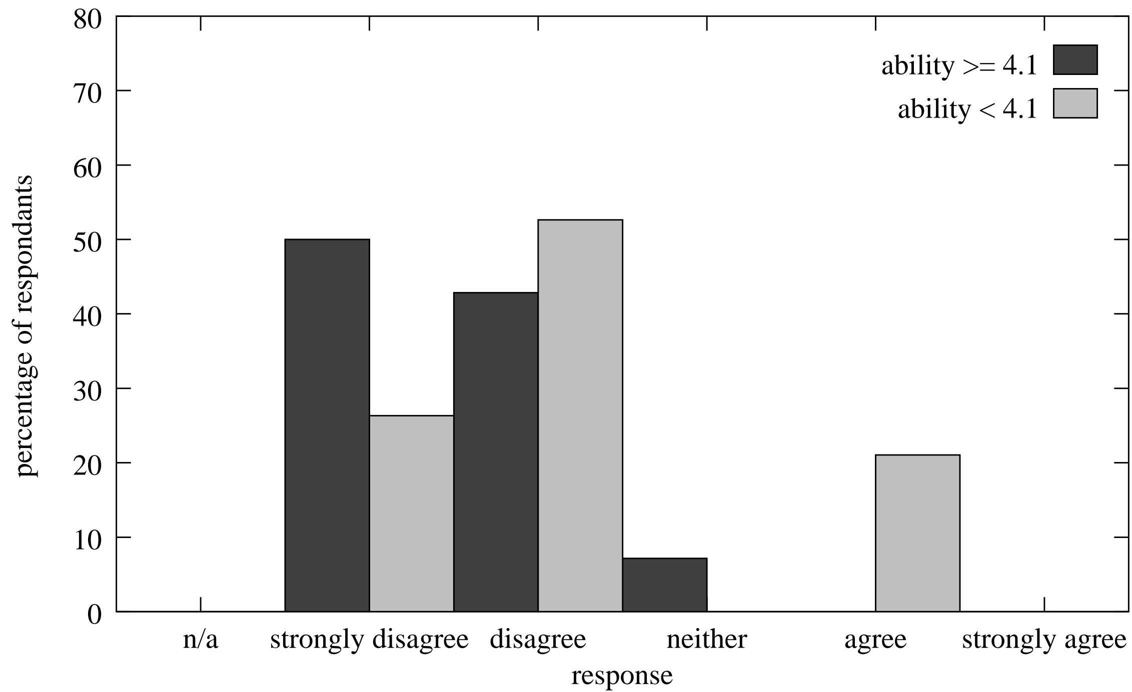

Figure 3.17: “I did not understand the explanation of arrays” (low and high ability comparison)

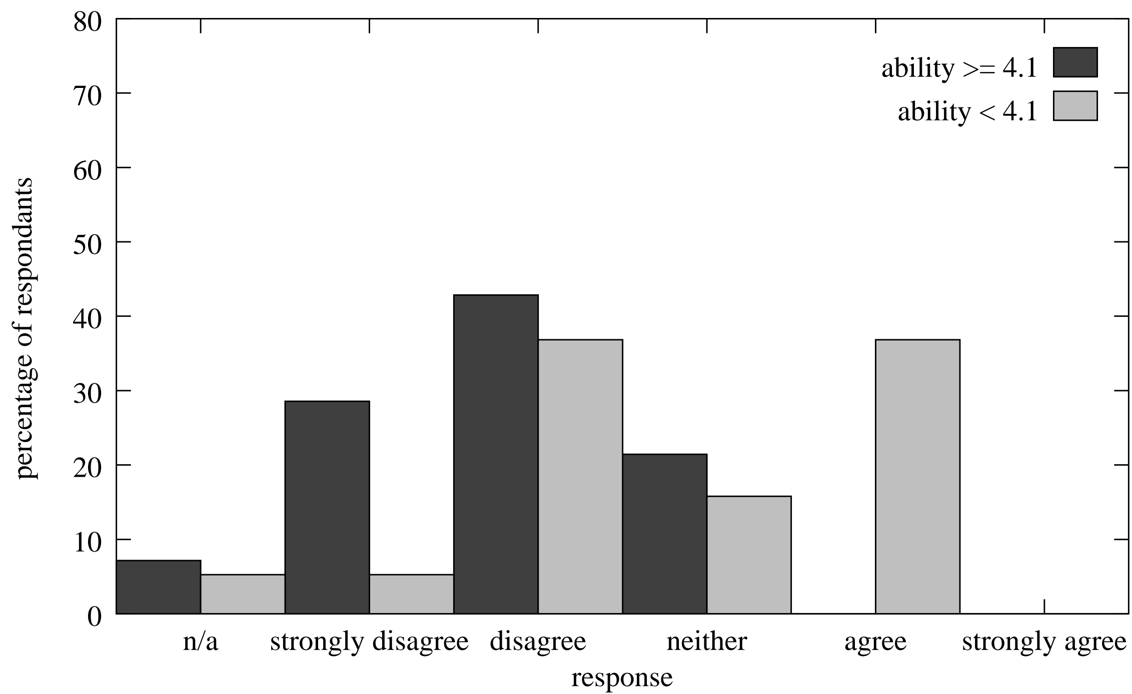

Figure 3.18: “I had trouble *reading from* files” (low and high ability comparison)

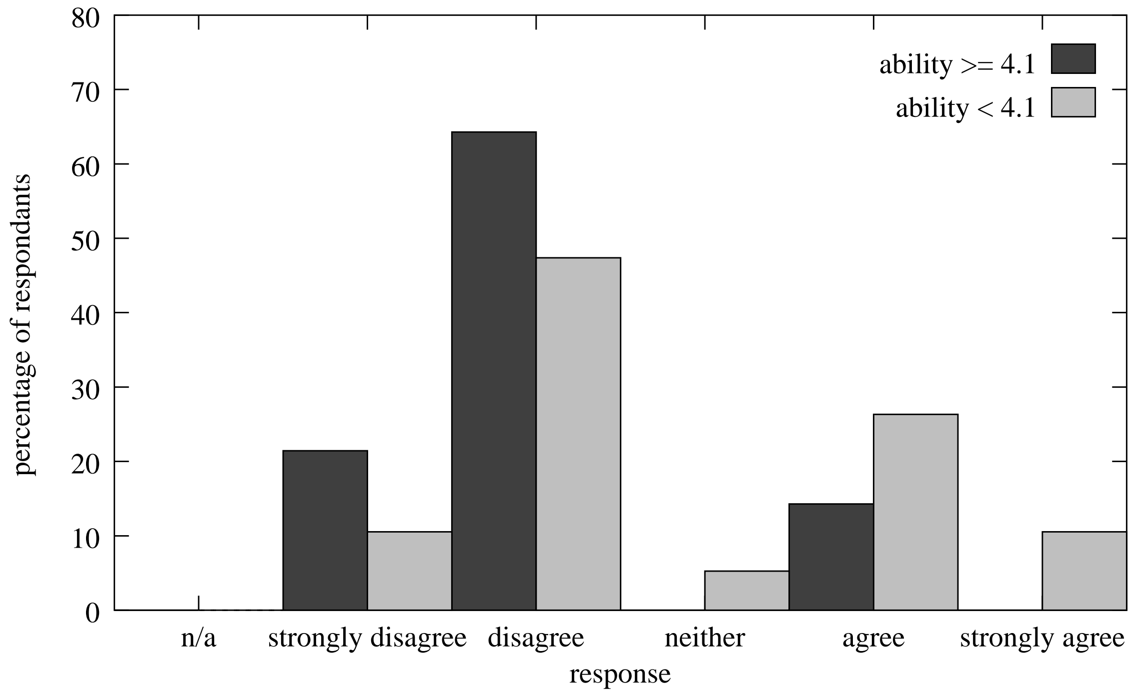

Figure 3.19: "I had trouble *writing* to files" (low and high ability comparison)

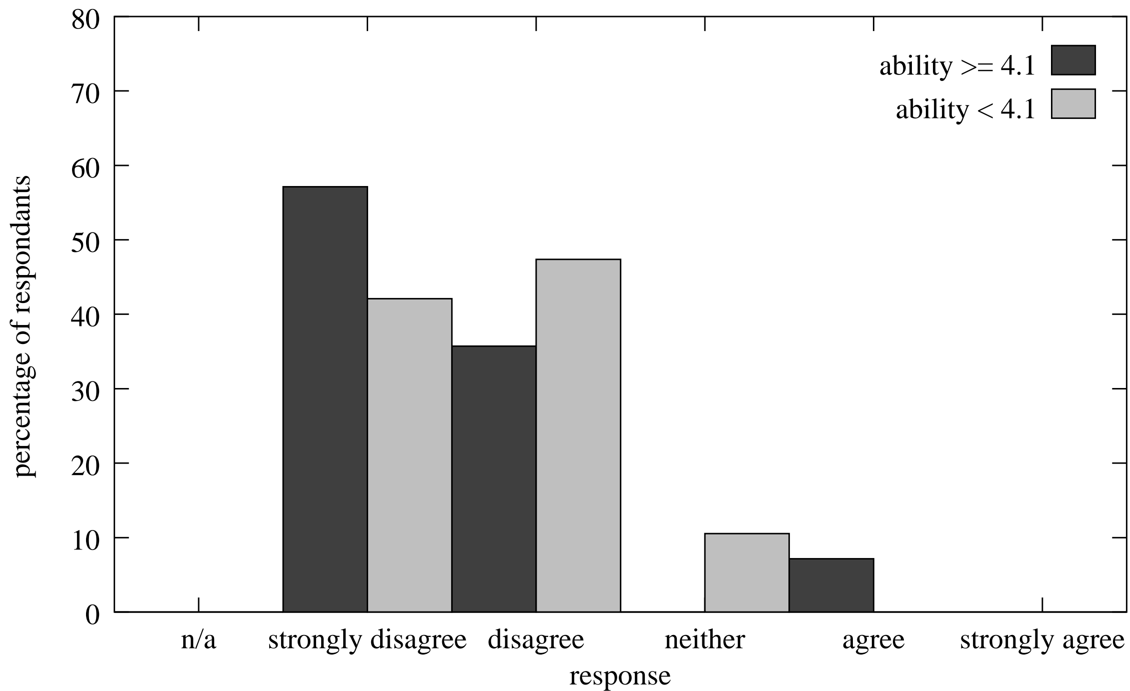

Figure 3.20: "I had trouble with the limits of the range() function"; the range() function is a function that generates lists starting at zero (rather than one) which is a common cause of confusion. (low and high ability comparison)

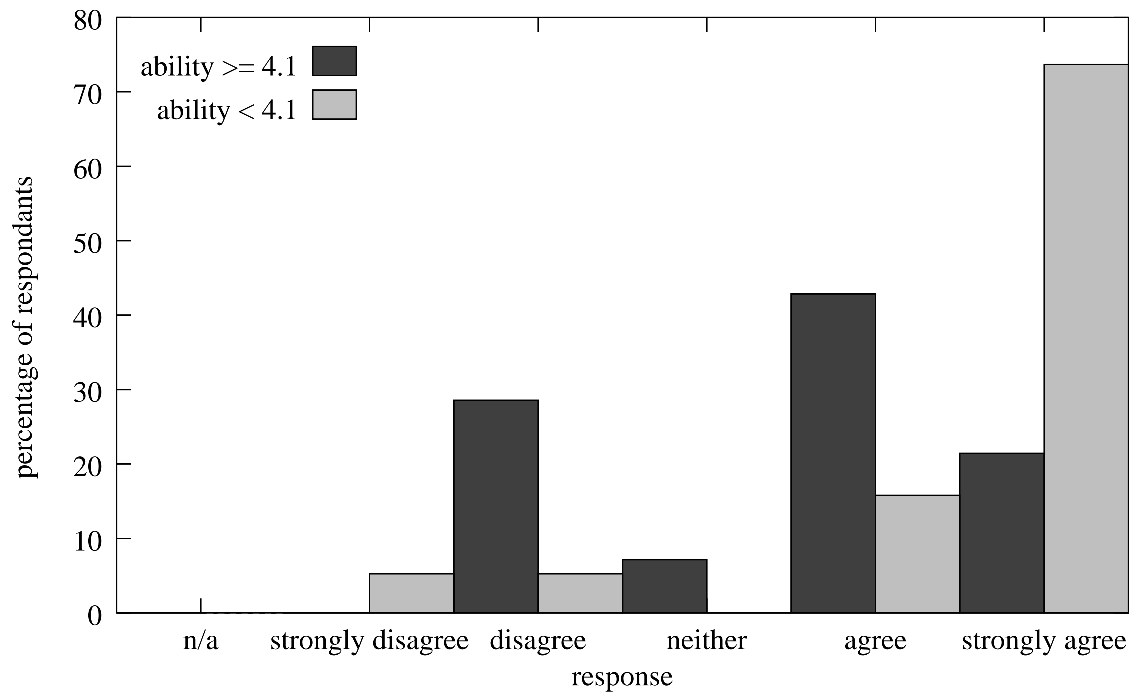

Figure 3.21: “I did not use the interactive interpreter much after the first couple of exercises” (low and high ability comparison)

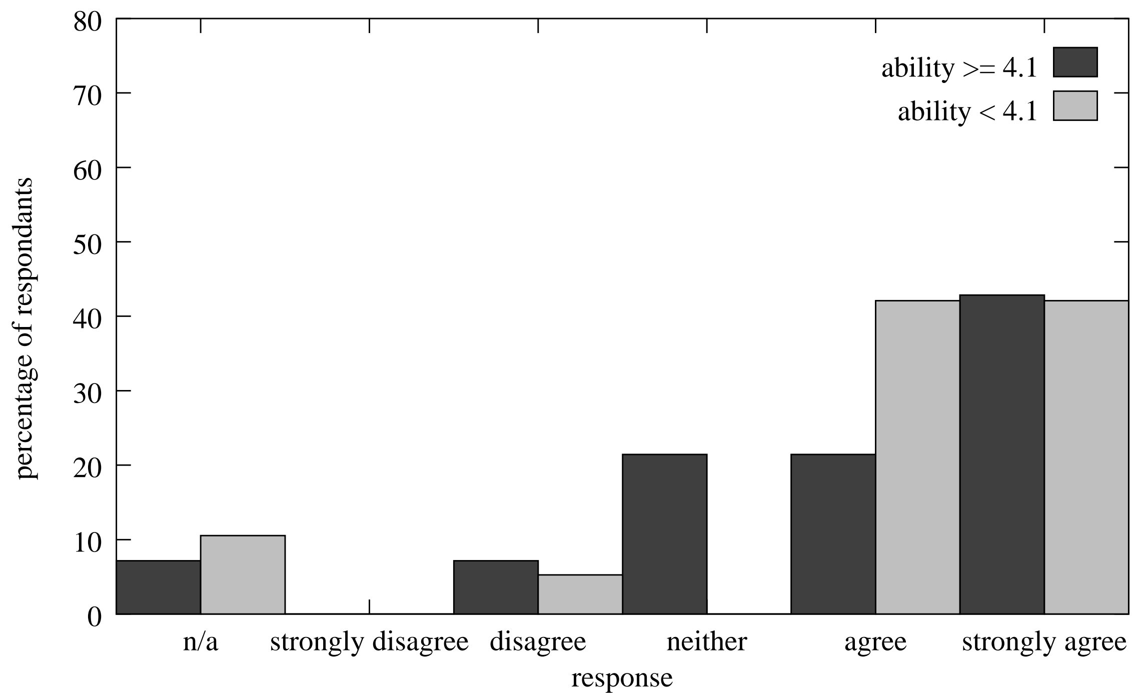

Figure 3.22: “I would rather write the program for the problem I solved in Python than Pascal (put n/a if you can’t remember Pascal!)” (low and high ability comparison)

CHAPTER

FOUR

Discussion

4.1 Aims

The aim of this project was to answer the two questions:

1. whether the use of Python as the principal teaching language of the department of Physics was *feasible* and;
2. Python's design and implementation made its use *preferable* to other languages, in particular Pascal and C.

This was done by:

- Producing a handbook from which students could use to learn Python, and;
- Running a trial of the proposed course.

4.2 Methodology

Because the nominal MPhys project period (in which the trial had to take place) conflicted with first-year examinations it proved impossible to involve them in the trial. However, second-years were relatively free at the time so were offered the opportunity to take part in the trial in return for practical course credit.

This meant all (except one) of the subjects had programming experience in Pascal. Furthermore, they had undergone nearly two years of scientific education which had improved their logical thought-processes. These factors meant that the sample's success or failure in the use of Python could not be considered as entirely representative of the experience first-years would have when using the course. However, it did allow them to make subjective comparisons of Python with at least one other language.

The number of students who took part in the trial was small (although the level of interest was much higher than had been anticipated). Thirty-three students completed the course (from a year group of around 200). Although this is a small fraction of the cohort, their abilities were generally representative. Self-assessed programming ability was gratifyingly close to the theoretical average (4.1 compared to 5) and there was a large range (from 1 to 9). More able students seemed to be taking part out of curiosity. The large number of relatively inexperienced programmers probably took part because of the offer of practical course credit.

The students were very good a spotting typographical errors, inconsistencies and confused explanations. As a result the handbook was changed on an almost daily basis. The version included with this report is the “final” version; the version whose contents reflects the changes made during and after the trial.

All students were required to complete the questionnaire, which was presented in the form of a web page. The questionnaire is included as Appendix A. The statements were designed to assess the problems (or lack of problems) due to some of the most obvious differences between Python and languages such as Pascal and C, and also to ascertain the handbook’s suitability.

A comparable C course was not available at the time of the trial. It is hoped that next year a more detailed comparison of the two languages can be made. This will perhaps involve a longer questionnaire allowing the derivation of more substantial quantitative results.

4.3 Is Python’s use feasible?

4.3.1 Portability

It must be capable of running on the Solaris-based SunRay system

There were no problems compiling Python (which is written entirely in C) on Solaris.

4.3.2 Speed with which it can be taught

It must be possible to teach enough of the language in just one day…

All the students managed to write a program to solve one of the problems. Students are introduced to the same requisite programming concepts.

4.3.3 Demonstrators

It must be possible for demonstrators unfamiliar with the language to quickly achieve such a level that they can help students debug their programs.

I was the course's principal demonstrator and am a relatively experienced Python programmer. However I was assisted by five others, only two of which had encountered Python before. By reading the handbook and being directed to http://www.python.org the demonstrators who were new to Python were quickly (usually within the first day) sufficiently competent that they could solve student problems. Obviously a significant fraction of these problems were not Python-specific (algorithmic misunderstandings, etc.), but those that were (mainly syntax errors) were always quickly solved by the demonstrators (with occasional reference to the handbook). The important conclusion is that demonstrators were quickly able to *read* Python programs.

4.3.4 Integrated Development Environment

There must be a simple, ideally graphical IDE...

There were some problems with the IDE used (IDLEfork). The client-server modification discussed in Section 2.3.2 was perhaps the most worrying, but several small problems were encountered during the trial:

- Files Python writes for which a full path is not specified are saved in `/tmp` until the program is saved (from then on the paths are considered as relative to the program file). This is generally not the expected behaviour (which should probably be to save to the user's home directory or the directory from which IDELfork was run).
- If a program was saved without a `.py` extension (which was a common error, in spite of its importance being emphasised) syntax highlighting is stopped (although code written before the save remains highlighted). This is strange. The editor should stop highlighting completely and in an educational context it would be useful for it to point out that you are no longer writing a Python program.
- IDLEfork is still quite unstable. On several occasions the editor stopped responding to requests to run programs. The solution was to exit IDLEfork altogether. The cause of this was never determined.

However, IDLEfork is in active development and all these problems have been discussed on the mailing list and solutions suggested.

In spite of these problems the environment provided by IDLEfork was at least satisfactory. Syntax highlighting was particularly useful, and the editor has several useful advanced features such as commenting out regions that some students found on their own.

4.3.5 Teaching of generic programming concepts

It must be possible to teach generic programming concepts . . .

This was possible, as can be seen from the relatively unchanged structure of the handbook. Python has all the control constructs of a typical procedural language[1]. Arrays are implemented in a fairly standard way (although there are some convenient simplifications to make their behaviour more mathematically intuitive). Lists (a multi-element data type) are outside the scope of the course and no harm is done by ignoring them. Although it does not appear in the handbook, formatted output in Python is syntactically almost exactly the same as formatted output in C.

The most profound semantic (rather than syntactic) difference between Python and its alternatives is the implicit declaration of variables and dynamic typing.

Implicit declaration means that memory is allocated by the interpreter the first time a variable is assigned to, without the need for a list of variables at the start of the program. Dynamic typing allows a variable to change type during the program. Students still need to know about the concept of "type", as can be seen from the handbook.

Python's interpreter *overloads* operators, which could be thought of as hiding too much of the real programming from students. For example, `4 * "Hello"` returns `"HelloHelloHelloHello"`. If the expression must return a result this is the only sensible answer. However, in this particular case, it would be preferable for Python to raise an error as there is little need for this kind of string manipulation in an introduction to scientific programming.

As well as being designed with the aim of being a clear language suitable for educational use, it is commonly employed as a scripting language. In that context being extremely high level is indisputably desirable, and it is this that has lead it to gain plaudits as a language suitable for rapid prototyping of large and complex applications. However, if it is desired to teach students the fundamentals of programming the two groups of user can have conflicting interests. The argument that the fundamentals should be taught can be reduced *ad absurdum* to a claim that programming should be taught in assembler[2]. The course is only two days and a balance must be struck.

I believe that the course finds this balance and A little emphasis in the handbook forces some more general skills to be used. Most of the time-saving scripting constructs are not of use in scientific programming (except arrays' matrix like behaviour). In any case, it is possible to recreate the difficulties or low-level nature of other languages using Python (for example, the operator overloading example above could be rewritten using a `for` loop).

The real danger of Python's extreme high level nature is the introduction of subtle bugs; the interpreter might do something it sees as the right thing, rather than raising an error. By regularly printing variables these bugs are often trivially soluble.

[1] With the exception of `REPEAT...UNTIL`, which can be replicated with `while` in all but the most obscure circumstances.
[2] Machine code.

Variables are implicitly typed (you need not specify a type on creation). Function definitions do not require the specification of the type or size of the actual parameters passed. This could be another source of error, but it was not one encountered during the trial. In fact, the lack of a requirement of the specification of array size allowed students to write more general programs.

From talking to students and questionnaire responses, the lack of variable declaration seemed to be one of the things they liked the most about Python. Dynamic typing and implicit variable declaration are indubitably desirable paradigms for a modern VHLL (like Matlab, IDL, Perl, Python, etc.) to follow. Whether student's programming education is best-served by the use of a language with such features is less clear.

The answer to the first question the trial was designed to address is "yes"; It fulfils all the criteria listed in Section 1.2.3. Python has problems that will be discussed in more detail later, but there are no fundamental barriers to its use as the principal teaching language; its use is *feasible*.

4.4 Is Python preferable?

Now I consider whether Python is *preferable* to Pascal and C. I discuss some of the desirable criteria mentioned in Section 1.2.3 and make more general considerations of Python's suitability as a teaching language.

4.4.1 Readability and the speed with which Python can be learnt

It should be possible for students to quickly write (and understand) basic programs...

This is perhaps the strongest point in Python's favour. It's syntax is very cleanly designed and intuitive. At times it looks like pseudo-code, and to experienced programmers with no knowledge of the language simple programs are usually immediately comprehensible. This suggests that students would find it easy to understand, which was found to be the case during the trial.

As mentioned in Section 1.2.3, this is desirable as a student's enjoyment and interest in programming is increased if they can quickly write a functional and useful program *and understand what they've done*. This is one of the dilemmas faced by someone attempting to introduce a student to Pascal or C; the traditional "Hello world" program is relatively long and contains a number of constructs which the student must either "take on trust" as magic every program requires, or the author of the course must explain. Both alternatives are likely to deter the novice programmer. Python's "Hello world" program is:

```
print "Hello world"
```

Programs which are functionally more complex still retain this readability and conciseness. As well as being more rewarding for students, it could allow more content to be squeezed into the limited time allowed for computing. If Python were used as the principal teaching language it might allow the setting of longer, more interesting problems. The standard library and large ammount of community-support aid this; a wealth of libraries already exist to do common tasks such as image manipulation.

Python (and scripting languages in general) are commonly used for projects which require rapid production of a usable application. This is because their conciseness of expression makes their use simpler. One quantitative measure of this is the number of lines of (non-comment) code in a program.

Studies of program length have been undertaken (generally as a component of a more wide-reaching language comparison). These invariably conclude Python programs are amongst the shortest in spite of their readability (which might reasonably be expected to make them more verbose)[10][11].

The perceived problem of its lack of rigorous declaration, which means that students do not become familiar with a common programming skill, does have advantages. It is one less thing that students either have to explain early on and seemed to accelerate their development through the language.

It is clear that the desire for students to quickly learn a language implies that language must have:

- clean and consistent syntax, which makes programs readable and,
- dynamic typing and implicit declaration, which makes programs short.

4.4.2 Python's use outside Education

a language should be usable—and ideally used *in many fields*.

The cursory analysis of the number of mentions of various languages found by the Google search engine showed that, although Python is a popular language, it is not as popular as C. However, it is perhaps more common than Fortran, which remains widely used in academia.

Python is undoubtedly less widely used than C but, in principle, Python could be used to do everything C can with the exception of very low level applications such as operating systems. You might not *choose* to use it for applications which require speed as it is interpreted, but in general it is possible.

The lack of speed with which Python programs execute might hinder its growing use in the scientific community were

it not for its extensibility. A carefully designed API makes it possible to implement the most time consuming parts of an algorithm in a compiled language such as C or C++, whilst retaining Python's clean design for the "glue" of the program. This has allowed the creation of the popular Numerical Python library, which implements, amongst other things, a matrix-like array data type which can be manipulated very quickly.

It is difficult to quantify what appears to be Python's growing acceptance in academia. However, it *is* being used. In our own Physics department Arzhang Ardavan and others are using it to model crystalline structures as part of their research in condensed matter. There is a web page promoting Python's use in astronomy[12]. A web search reveals it is also being used in scientific disciplines from Molecular Biology at Grenoble[13] to Meteorology at the Swedish Meteorological and Hydrological Institute[14]. It is also used in non-academic (but still "real-world", non-educational fields)[15].

Furthermore, Python is multi-paradigm; it is possible to write both procedural and object-oriented programs.[3]. This makes it a language that students could use if they came to write object-oriented programs later in their careers (unlike C).

In conclusion, Python is used outside education (unlike Pascal) and it is a worthwhile specific skill for students to have (but not as useful as C).

4.4.3 Python's peculiarities

It should not have any unnecessarily complicated constructs whose use is required to undertake common programming tasks...

In general, the trial demonstrated that Python's way of doing common tasks was suited to educational use; there are few complicated constructs.

Perhaps the most clumsy thing in Python is formatted input, such as reading in a data file with many entries on the same line. In Pascal this is done using `readln(x, y)` (or `readln(fin, x, y)` where `fin` is a file pointer). `x` and `y` are then the first two whitespace separated numbers on the line. The program will fail at runtime if it reads in something other than numerical values. It is not much more complicated in C, although it does involve pointers, which is a topic a one-day introductory course would probably avoid:

```
scanf("%d %d\n", &x, &y);         /* from stdin */
fscanf(fin, "%d %d\n", &x, &y);   /* from file  */
```

[3] And, less usefully from the point of view of this investigation, functional programs which LISP is often used for.

In Python the same task is achieved by doing:

```
linestring = raw_input()                  # Read in a line of input
                                          # from stdin.
linelist = string.split(linestring)       # Split the line on
                                          # whitespace, returning a
                                          # list of strings.
x = int(linelist[0])                      # Assign the coerced strings
y = int(linelist[1])                      # to the required variables.
```

If input is to be taken from file the first line is replaced with:

```
linestring = fin.readline()
```

Here `fin` is a file object. This method is objectively convoluted in comparison to Pascal and C. It did not appear to be a problem for many of the trial students though. This was required by the easiest problem (CO11) which was attempted by the least able students. Encouragingly, few of them found it a problem. Those that did had usually incorrectly coerced the strings returned by the `split` function which is not a problem in Pascal as it is not dynamically typed. The questionnaire suggested a large minority found reading from files a problem (21%), but it is worth bearing in mind that reading and writing files is always one of the things students struggle with.

4.5 Trial experiences and questionnaire results

The students' reaction to Python after completing the trial was favourable. Fears that Python idiosyncrasies would be confusing generally proved unfounded.

4.5.1 Indentation a block-delimiter

No students claimed to have trouble indenting and most said they understood its significance (see Figures 3.2 and 3.3). It was feared that Python's unique way of delimiting code blocks would be problematic. In fact many students expressed an appreciation of it, perhaps due to its intuitiveness; it is used in much the same way as a document written

in English might be indented. Perhaps rather more infinite loops were encountered than usual, but not very many. In his report on using Python as an introductory language Jeff Elkner suggests a plausible explanation:

> The idea of several statements acting as one takes some getting used to, and it appears the visual cues of a `BEGIN` and an `END` make it easier to understand[16]

However, as Elkner says, this is a price worth paying in return for consistent code layout. One of the most common problems for demonstrators when teaching Pascal (and one that would also be encountered in C) is students either not indenting their programs at all or, even worse, doing so inconsistently. This makes it more time-consuming for them to solve student's problems. Enforcing a consistent style on students from the start eliminated this entirely. It is a useful skill in preparation for other languages where, although not required, consistent and meaningful indentation is considered good practice.

4.5.2 Error messages

Trial participants generally thought the error messages were quite helpful. Sixty-nine percent either agreed or strongly agreed with the statement "I found the error messages sufficiently helpful such that when they occurred I could quickly spot the problem" (see Figure 3.4. Python error messages are perhaps a little verbose for the type of programs the students are asked to write. Here is an example:

```
>>> print a
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'a' is not defined
```

In general, all you need to know for a short program (i.e. one source file) is the line number the interpreter found an error on (always line 1 when running interactively) and the last line of message, which describes the nature of the error. This description is usually quite useful in diagnosing the problem. The handbook had a section on translating these error messages (which are a little esoteric at times) and IDLEfork's ability to jump straight to the suspect line was found extremely useful.

Python does raise one rather confusing error: when there is a syntax error on one line (such as a missing closing bracket) Python claims there is a syntax error on the line that follows. This is because that is the first time it encounters syntactically illegal code. Demonstrators quickly learned to spot this.

As is the case in many other languages, Python's errors are difficult to understand until you know what they mean, but once you do they are very helpful. Hopefully the appendix on errors included in the Handbook will expediate students escape from this chicken-and-egg situation. The appendix was inspired by and derived from the excellent notes included in the Livewires course, a summer camp that, amongst other things, teaches teenage children to program in Python[17]. Discussions with other demonstrators suggested that all of them found the messages at least satisfactory, if not better than other languages.

Unfortunately, at the time of writing, an equivalent course to teach students to program using C has not been sufficiently tested to make detailed comparisons. In the brief trials of the C course the most significant difference seemed to be the nature of the error messages. Because C is a compiled rather than interpreted language there is a qualitative difference in the origin and number of errors encountered at compilation compared to those encountered by the Python interpreter. In general, if a program contains one error the Python interpreter will stop at that error and refuse to continue. However, an equivalent error in a C program may lead to many more error messages. Although these error messages may all have the same origin and be trivially soluble, the sheer number of errors with which the student is presented has the potential to be off-putting.

4.5.3 Syntax errors

Syntax errors were common, as is the case with any language. The commonest response to the statement "I often made syntax errors like missing colons and brackets" was "I agree" (see Figure 3.5. However, Python does have less "punctuation" than Pascal or C so lack of semi-colons at the end of lines seemed to contribute to many student's appreciation of its syntactic clarity. IDLEfork was again helpful: if a colon was missed off from the end of a control statement header (e.g. the first line in a `for` loop or after the condition in an `if`) it would not automatically indent the next line as they expected. This set alarm bells ringing for many students, who specifically mentioned this feature as something they found useful.

4.5.4 Arrays

An encouragingly large proportion of the sample understood the explanation of arrays, which are invariably a tricky topic. This is probably not because of the use of Python but rather an increase in the ammount of explanation of arrays in the handbook over that in the Pascal version.

4.5.5 Reading and writing files

As usual, reading and writing files was slightly problematic. The possible reason for difficulty reading is discussed above (see Section 4.4: the consideration of formatted input). Writing was also problematic but for a different reason. Up until they were asked to write to file students had been using the extremely powerful `print` statement, which takes a lot of the hard work out of printing results to screen. However the `write` method of a file object which is used to write to file cannot take several arguments; it may only take one argument and that must be of type string.

For example one may use `print "Hello", name` to greet a person by printing to screen but separating multiple arguments with commas makes the `write` file object method raise an error:

```
>>> f.write("Hello", name)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function takes exactly 1 argument (2 given)
```

This is confusing to students who had previously been separating many, arbitrarily-typed variables with commas and using the print statement to print them to screen.

The workaround used during the trial was to have students join strings using the + operator (note the addition of a space—the + operator does not insert one):

```
f.write("Hello " + name)
```

Many students did not coerce numbers into strings before attempting to concatenate:

```
>>> x = 5.1
>>> f.write("x is " + x)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'float' objects
```

A better solution would have been to avoid the overloading of + and use the ability of `print` to write to files in much the same way as the Unix command line allows for redirection of the standard output stream which was introduced in

Python 2.0:

```
print >> fout, "x is", x
```

Unfortunately I was not aware of this feature until after the trial, when it was pointed out to me by Guido van Rossum.

Using this method of writing to file partially addresses one of the apparent syntactic inconsistencies of Python, namely it's OOP-derived method syntax. All objects have "methods" which are ways of operating on the objects. For example, if `l` is a list, one can append another value to it using the append method

```
>>> l = [1,2]
>>> l.append(3)
>>> print l
[1, 2, 3]
```

Note the lack of an assignment statement; it is not written `l = l.append(3)`. This is different to the equivalent procedural construct which would generally involve an assignment of a result. This syntax is one of the few things in the handbook which is "hand-waved" away. In the course it is only used for file methods: `fin.readline()`, `fout.write()`, etc. If one of the objects of the course is to "hand-wave" as few things as possible away, the `print >> fout, ...` construct removes the need for the `fout.write` method (although not `fin.readline()`.

4.5.6 The range function and fencepost errors

Python's multi-element objects (lists and arrays as far as the course is concerned) are all zero-offset: the first element is indexed as `[0]`. This is also true of C, but not true of Pascal. The first element of a Pascal array can be an arbitrary index, determined on the array's declaration. Common choices are `[0]` and `[1]`.

Python's `range` function which allows the emulation of `for` loops in other languages (i.e. allows you to step through a code block, incrementing some counter by a constant index) is an example of the "fencepost errors" novice programmers often encounter[18]

In order to generate a list that goes from `0` to `n` inclusive one uses `range(0, n + 1)` (the first argument is optional; code0 is the default). This returns a list with `n + 1` elements. In the handbook student's are told: "this may seem strange but there are reasons". In fact problems with the `range` function itself were surprisingly rare (see Figure 3.9)

A more common error involved arrays. An array can be created using the `zeros` function which takes (at least) on argument: the number of elements in the array. For example, to create an initialised, ten-element array of integers one might do `xx = zeros(10)`, where `xx` is the array. It was then common for students to get confused by fencepost errors and try and step through the elements of the array using a `for` loop as follows:

```
>>> xx = zeros(10)
>>> for i in range(0, 11):
...     print xx[i],       # (the trailing comma suppresses the new line
...                        # automatically included by print)
0 0 0 0 0 0 0 0 0 0
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
IndexError: index out of bounds
```

Students appeared to be confusing the fact that the last number returned by `range(11)` is `10` and the index of the last element of `xx`, which is `9`. It is then that the most able students spotted the reason `range` works like it does; the argument given to the `zeros` function is almost always the same as the argument given to the `range` function. Demonstrators should be aware that, if they see different arguments for `zeros` and `range`, the student is probably confused. Of course in this case the error message is very descriptive.

4.5.7 The interactive interpreter

The interactive interpreter is a very powerful tool for debugging and interactive graphical work. However, it seemed that very few students used it for anything but the first exercise, in which they were explicitly told to do so; 78% agreed with the statement "I did not use the interactive interpreter much after the first couple of exercises"(see Figure 3.10). This may change if the course is run on students who have never programmed before, who have no pre-disposition to typing a list of statements into a text file and having the interpreter run through them one-by-one.

Interestingly, there did seem to be a correlation between programming ability and use of the interactive interpreter; 30% of the most able students disagreed with the statement. This might be because the more experienced students are likely to be comfortable in command line environments, or they more quickly see the possibilities offered by interactive work. One or two even identified it's similarity with the BASIC interpreter, with which they had experience.

Perhaps the handbook should have been a little more directive in its use of the interactive environment; it is very useful

as a debugging tool.

4.5.8 Python compared to Pascal

The general reaction to the course was extremely favourable. When asked to compare Pascal and Python 75% of students expressed a preference for Python. This could be due in part to the Solaris environment which, although not as consistently well-designed as the NeXTStep environment, is undoubtedly more modern looking and probably more Windows-like.

However, the comments made by students during the trial and at the end of the questionnaire did explain specific reasons for preferring Python. The most common reason was its syntax. It's lack of semi-colons to end lines was a recurring point made in Python's favour by the students.

Student's appreciation of minor syntactic differences to Pascal like the lack of semi-colons seemed to be a manifestation of a more general increase in understanding offered by Python's improved readability. For many students the computing practical is the part of the course they least understand and, as a result, least enjoy. The relative lack of inexplicable syntax seems to result in increased understanding and enjoyment of the course and meant they could spend more time on problem decomposition and algorithmic details, and less on the details of programming.

4.6 Conclusion

I believe the trial has shown conclusively that it is both possible and desirable to use Python as the principal teaching language:

- it is Free (as in both cost and source code).
- it is trivial to install on a Windows PC allowing students to take their interest further. For many the hurdle of installing a Pascal or C compiler on a Windows machine is either too expensive or too complicated;
- it is a flexible tool that allows both the teaching of traditional procedural programming and modern OOP; It can be used to teach a large number of transferable skills;
- it is a real-world programming language that can be *and is* used in academia and the commercial world;
- it appears to be quicker to learn and, in combination with its many libraries, this offers the possibility of more rapid student development allowing the course to be made more challenging and varied;

- and most importantly, its clean syntax offers increased understanding and enjoyment for students;

Python should be used as the first year teaching language. If used it will be possible to teach students more *programming* and less of the peculiarities of a *particular language*. Teaching a mid-level language like C in just one day is inadvisable. Too much time must be spent teaching C and not enough time teaching generic skills to students with no programming experience.

The use of Python as the first year language is not a dead-end. I have tried to emphasise that Python allows the teaching of widely applicable programming concepts. Its use in no way precludes the use of C in a more advanced course. In fact students who go on to use C in later years will have a better grounding in concepts from their introduction to programming than they might have from a C-based introduction. I believe that more students will go on to advanced programming if introduced using Python because introducing programming using C will frustrate and scare off a large number of students.

In conclusion, Python offers the optimum compromise of teachability and applicability.

BIBLIOGRAPHY

[1] Oxford Physics - Student Handbook 2001. http://www2.physics.ox.ac.uk/front/teaching/handbook01.htm accessed 12/8/2002.

[2] Clive D Rodgers et al. Handbook of the physics computing course.

[3] CO11–CO16. Worksheets describing the computing problems used on the Oxford Physics computing course. Available from the Department of Physics, Oxford University.

[4] Brian W. Kernighan. Why Pascal is not my favorite programming language. Computing Science Technical Report 100, AT&T Bell Laboratories, 1981.

[5] Python FAQ. http://www.python.org/cgi-bin/faqw.py accessed 12/8/2002.

[6] EDU-SIG: Python in Education. http://www.python.org/sigs/edu-sig/ accessed 10/4/2002.

[7] The IDLEfork Project. http://idlefork.sourceforge.net/ accessed 6/3/2002.

[8] Numerical Python. http://www.pfdubois.com/numpy/ accessed 10/4/2002.

[9] Gnuplot.py. http://gnuplot-py.sourceforge.net/ accessed 17/5/2002.

[10] The great computer language shootout. http://www.bagley.org/ doug/shootout/ accessed 11/6/2002.

[11] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical Report 2000-5, Universitä Karlsruhe, Fakultät für Informatik, Germany, March 2000. http://www.ipd.uka.de/ prechelt/Biblio/.

[12] The Astronomical Python home page. http://lheawww.gsfc.nasa.gov/ bridgman/AstroPy/ accessed 22/4/02.

[13] Python for Sciene. Molecular Modelling at the University of Grenoble. http://starship.python.net/crew/hinsen/ accessed 22/4/02.

[14] Swedish Meteorological and Hydrological Institute. http://www.smhi.se/.

[15] Who else uses Python? http://www.python.org/psa/Users.html accessed 22/4/02.

[16] Jeff Elkner. Using Python in a high school computer science program. In *Proceedings of the 8th International Python Conference*, 2000. http://www.python.org/workshops/2000-01/proceedings.html.

[17] Livewires. http://www.livewires.org.uk/python/index.html accessed 28/4/2002.

[18] Eric S. Raymond, editor. *The New Hacker's Dictionary*. The MIT Press, 1996. See entry on "fencepost errors": http://www.tuxedo.org/jargon/html/entry/fencepost-error.html.

APPENDIX

A

The questionnaire

On completion of the trial students were asked to fill in a questionnaire. They were asked to give their name, email address, username, college, and the problem they attempted (CO11 - CO16). If they could program in any other languages than Pascal they were asked to list them. They were then asked to indicate how strongly they agreed with the following statements:

- The hanbook was too long;
- I had trouble indenting my program correctly;
- I understood the significance of indentation;
- I found the error message sufficiently helpful such that, when they arose, I could quickly see the problem;
- I often made syntax errors like missing colons and brackets;
- I did not understand the explanation of arrays;
- I had trouble *reading from* files;
- I had trouble *writing to* files;
- I had trouble with the limits of the `range()` function;
- I did not use the interactive interpreter much after the first couple of exercises;
- I would rather write the program for the problem I solved in Python than Pascal (put n/a if you can't remember Pascal!)

The possible responses were:

- Not applicable/unknown;
- Strongly disagree;
- Disagree;
- Neither agree nor disagree;
- Agree;
- Strongly agree.

They were then given the opportunity to make any general comments on the course.

Fixing IDLEfork

As described in Section 2.3.2, the latest stable version of IDLEfork at the time of writing (0.8.1) is unable to run multiple instances for different users on the same machine. The developers are in the process of fixing this but, until then, the following workaround was implemented:

The file `protocol.py` in the IDLEfork source was edited to allow the `default_port` to be altered by an external environment variable. The original line 322:

```
default_port = 0x1D1E   # "IDlE"
```

becomes:

```
default_port = 0x1D1E   # "IDlE"
#
# Hack added by CRW/MJW to workaround idle mis-design
# idle is now called via a wrapper script which sets the
# ENV variable IDLEPORT to PID+4096 -- we will use this
# value rather than the default of 0x1D1E   -- 2002-05-09
#
    p = os.environ.get("IDLEPORT")
    if p: default_port = int(p)
```

The wrapper script used is the following:

```
#!/usr/local/bin/python
# Wrapper script for idle (idlefork 0.8.1) to allow multiple
# separate instances to be run simultaneously by different users.
# Implemented using a crude hack to the idle (fork) prog itself
# (see idle-0.8.1)
#
#                 CRW & MJW  2002-05-09
import os
thispid = os.getpid()
portnumber = thispid + 4096
os.putenv("IDLEPORT", str(portnumber))
os.system("/usr/local/bin/python -Qnew /usr/local/bin/idle-0.8.1 -i")
```