The Python Papers Monograph, Vol. 1 (2009)
Available online at http://ojs.pythonpapers.org/index.php/tppm

Introductory Programming with Python

BRENDAN MCCANE
Department of Computer Science
University of Otago
Dunedin, New Zealand

This paper describes a new course introduced at Otago University in 2009, called "Practical Programming in Python". The course is intended as a first course in programming and uses the excellent programming language Python. Python was chosen because at an introductory level it is simple, has very few unexplainable concepts, and immediate feedback is possible. Despite these attributes, the language and language environment is as useful for practical programming tasks as any other language. We describe the principles used for designing the course, the curriculum and structure of the course, and the student outcomes.

1 Introduction

With apologies to Douglas Adams [1]:

> Programming is hard. You just won't believe how vastly, hugely, mind-bogglingly hard it is. I mean, you may think making a decent sandwich is hard, but that's just peanuts to programming.

Here's what Donald Knuth says [6]:

> In fact, my main conclusion after spending ten years of my life working on the TEX project is that software is hard.

In case you are still not convinced, [2] report that the average failure rate across CS1 courses they surveyed was 33%, but with a very high variance (between 0 and 60% fail rates depending on the course).

Universities typically attempt to teach industry relevant programming languages such as Java, C# and C++ in their first programming course, often claiming that object-oriented programming (OOP) is more "natural" than procedural programming. The evidence seems to suggest that OOP is mostly just more complex than procedural [8]. Although learning the particular syntax of a language is only a small part of learning to program, in such a difficult field, it seems foolish to erect more learning barriers than is necessary.

Because of these issues and the perceived problems with attempting to teach Java as a first programming language, we wanted to introduce a course that used a language with the following principles:

- as simple as possible
- as few magical incantations as possible
- immediate feedback
- a practical and modern language.

Those principles led us immediately to modern scripting languages and very quickly to Python.

This paper is a report on our experiences with the first time the course was offered. In Section 2 I describe the curriculum of the course, in Section 3 the structure of the course is outlined, in Section 4 the student outcomes and comments are discussed, and last words are given in Section 5.

2 Curriculum

We chose an open source textbook [4] as the basis for the course and subsequently modified it (sometimes quite heavily) to suit our needs [5]. Both of these books can be downloaded from the web. Our textbook [5] is roughly divided into two parts. The first part deals largely with the essential components of computation and programming (up to the in class test), while the second half deals with slightly more advanced and interesting topics. The lecture outline is as follows:

1. Introduction, what is a program, debugging and environment
2. Variables, expressions and statements
3. Python built-ins (batteries included)
4. Functions part 1: definitions, flow of execution, parameters and arguments
5. Functions part 2: locality, stack diagrams, comments
6. Conditionals: Booleans, operators, chained and nested conditionals
7. Fruitful functions: return values, program development, the function type
8. Test driven development: modules and files, triple quoted strings, doctest

9. Files and modules: files, file processing, directories, creating modules, namespaces, attributes and the dot operator
10. Iteration part 1: multiple assignment, updating variables, while statement, tracing a program
11. Iteration part 2: nested iteration, encapsulation and generalisation, algorithms
12. In class test
13. GUI programming: event driven, TkInter, callbacks
14. Case study: Catch part 1
15. Case study: Catch part 2
16. Strings part 1: length, for loop, slices, comparison, `in` operator, looping and counting
17. Strings part 2: `str` methods, string formatting
18. Lists part 1: accessing elements, length, membership, operations, slices, range function
19. Lists part 2: for loops, parameters, pure functions and modifiers, nested lists, matrices
20. Tuples: mutability, assignment, as return values, sets
21. Dictionaries: operations, methods, aliasing and copying
22. System programming: `sys` and `argv`, `os` and `glob`, case-study
23. Classes and objects: OOP, attributes, methods
24. Case Study 2: encrypting a file.

The order of the lectures has been determined by two guiding principles. We wanted to take a very structured approach, but also to introduce the Python environment and the use of Python as early as possible. Hence the early chapter on Python built-ins which might otherwise look out of place. The idea of introducing functions early was a result of the order taken by [4], although we do not think it makes that much difference - students are already familiar with the notion of functions from their use of calculators. The lecture on test driven development was introduced as early as possible because we wanted to embed the idea of testing code as a process that was part of programming, and not something in addition to programming. Files were introduced relatively

early for practical rather than idealogical reasons - we wanted to be able to use files for more interesting laboratory exercises rather than always relying on keyboard input.

The lectures up until the class test were focused largely on what we consider to be the fundamentals of programming: sequence, selection, iteration and encapsulation. The latter half of the course was in some ways optional extras, with a very practical bent and focused on the data structures supplied with Python and their use. The GUI programming and case study lectures were included partly for motivational purposes, and partly because of the practical aspects of GUI programming. I'll discuss more regarding this part of the course in Section 4 below. System programming was introduced to open students' eyes to the power of scripting, especially for doing repetitive tasks — something students don't normally get in a traditional computer science course. The lecture on OOP was a very brief introduction and was meant to lead into the semester two course which uses Java. Finally, the last content lecture was a further case study with cryptography being chosen because I thought students may be interested, but also as an introduction to Computer Science and the sorts of algorithms that Computer Scientists think about.

3 Structure of the Course

The course structure was reasonably traditional with a few twists. For each lecture, there was a corresponding laboratory with a set of programming exercises — both basic and extension exercises for the more adventurous. There were a total of 21 laboratories and students had to submit work for 18 out of the 21. The submissions were not marked, but we wanted to force students to attend lab sessions as there is very strong evidence that those who don't attend labs, don't pass. Although the reverse appears to be largely true (those who do attend, pass), this is not necessarily a causative factor and there is some evidence to indicate that attendance at laboratories is self-selecting — students who are coping with the course tend to keep coming, while those not coping, don't [7]. Nevertheless, we wanted to encourage lab attendance as much as possible.

The lecture format was quite different to any CS paper I had previously been involved with. Each lecture was essentially a programming demonstration with the lecturer (me) programming live in front of the class covering the content of the lecture material. There were two main reasons for this approach. Firstly, in most courses, students never get to see an actual programmer programming. They only get to see the result and not the process. Imagine being taught how to build a table where the lecturer made use of explanation and diagrams, but the student never got to see someone actually build a table. Programming live let's students see the process warts and all. Typos, spelling

mistakes, syntax errors, semantic errors were all part of the lecture content. The idea being that students won't get discouraged when they come across similar problems. Secondly, programming live forces the course to follow a fairly slow pace. The only code I show during a lecture is code that I had to type during the lecture. This provided a useful upper bound on the amount of material that could be covered in a single lecture. This approach wouldn't be applicable for more advanced courses, but for a beginning course, the pace seemed just right.

Another interesting strategy we used to "encourage" students to maintain a reasonable understanding of the material was the use of mastery tests. There were two mastery tests during the course, each worth 10% of the final grade. The tests involved students solving programming problems in the laboratory and tests were deemed to be passed if the doctests passed. The contents of the tests were published well before the actual tests, so students could, if they desired, practice the actual test as much as they liked prior to the real thing. Although they knew the contents of the test beforehand, they were not allowed to bring any material with them into the test. Here's an example of one of the more difficult problems in the first mastery test which happened in lab 9:

```
def score(numbers):
    """
    give the average of the numbers excluding the biggest
    and smallest one
    >>> score([2, 7, 9, 10, 13, 1, 5, 12])
    7.5
    >>> score([3, 7, 2.5, -4])
    2.75
    """
```

The major form of assessments were the mid-semester class test and the final exam worth 20% and 60% respectively. Both were fully multi-choice answer exams, which is a slightly unusual choice for Computer Science exams. There were three major reasons for this. Firstly, I wanted to remove the problems of mistaken syntax from the exam setting. Many first Computer Science courses focus the exam marks on questions of syntax without the advantage of a compiler or interpreter at hand. This was perhaps a valid form of examination when batch compilation was the norm, but seems completely unnatural today [1]. Also, the mastery tests had already examined the ability of students to type correct syntax. Secondly, it is my belief that examining understanding of code (although not perhaps code generation) can just as easily be done in a multi-choice format. In a beginning course, examining

[1] Leaving aside the unnaturalness of any final examination.

understanding in preference to generation seems perfectly appropriate. Finally, the lecturer for the course is inherently lazy and hates marking exams. Here's an example of one of the more difficult exam questions used in the final:

What is the output of the following code?

```
def matrix_to_sparse(in_matrix):
    sparse = {}
    for row_index,row in enumerate(in_matrix):
        for col_index,val in enumerate(row):
            if  val != 0:
                sparse[(row_index,col_index)] = val
    return sparse

matrix = [[0,0,1], [0,2,0], [3,0,0]]
sparse = matrix_to_sparse(matrix)
print sparse[(2,0)], sparse[(1,1)], sparse[(0,2)]
```

(A)

```
1 2 3
```

(B)

```
1 3 2
```

(C)

```
3 1 2
```

(D)

```
2 1 3
```

(E) None of the above.

4 Outcomes

In this section, I'll outline the results of the course, both the student outcomes and the results of a course evaluation we handed out to students towards the end of the course, but before the final exam. The course started with 180 students with 172 of those attending the first lab which is a good indication of those who intend to at least attempt the course (there are always a proportion of students who enrol, but never show up). Thirty three students submitted fewer than 18 labs and were hence not allowed to sit the final exam. A total of 27% of the students failed the course (including those who did not sit the final). There were 28% A's, 23% B's, and 22% C's. A remarkably flat distribution and one that is quite unusual for a CS1 course which typically has a bi-modal distribution [3, 7].

The results of the mastery tests are rather illuminating. These tests are divided into 4 sections with each section gaining a pass/fail mark and each section worth 2.5%. Figure 1 shows the results for the two mastery tests. Remember that the students knew the contents of the test well before sitting it and they could practice the test beforehand as much as they liked. The first test results (Test1Try1) were quite disappointing with a significant number of students failing (approx 40%). These results also follow a strong bimodal distribution as can be seen from the figure. Since the tests are mastery tests, we decided to let the students have a second chance (Test1Try2) if they wished. These results seem to indicate that not only are students unable to complete some tasks, they're not even aware that they are unable.

Figure 2 shows the results of some of the evaluation questions asked of students towards the end of the course. Of particular interest is the mastery test question which asked "How effective were the mastery tests in encouraging you to learn important skills?" Clearly these tests were extremely effective despite the fact that students had ample opportunity to gain these skills outside of test conditions, the test seems to have forced skill attainment that otherwise may not have occurred (as evidenced by the difference in distributions for Test1Try1 and Test1Try2). Also of interest is the rating of the coursebook — students seemed to really appreciate the very tight integration between text book, lectures and lab exercises — something that was only possible because of the open-source nature of the textbook.

Students also had a chance to answer some open-ended questions in the course evaluation. Notable answers include (number of responses in brackets):

Best aspect: the labs (10)

I would like to change: marked labs (4), "a move away from extensive use of programming" (1).

Easy topics: start / first half of the book (37)

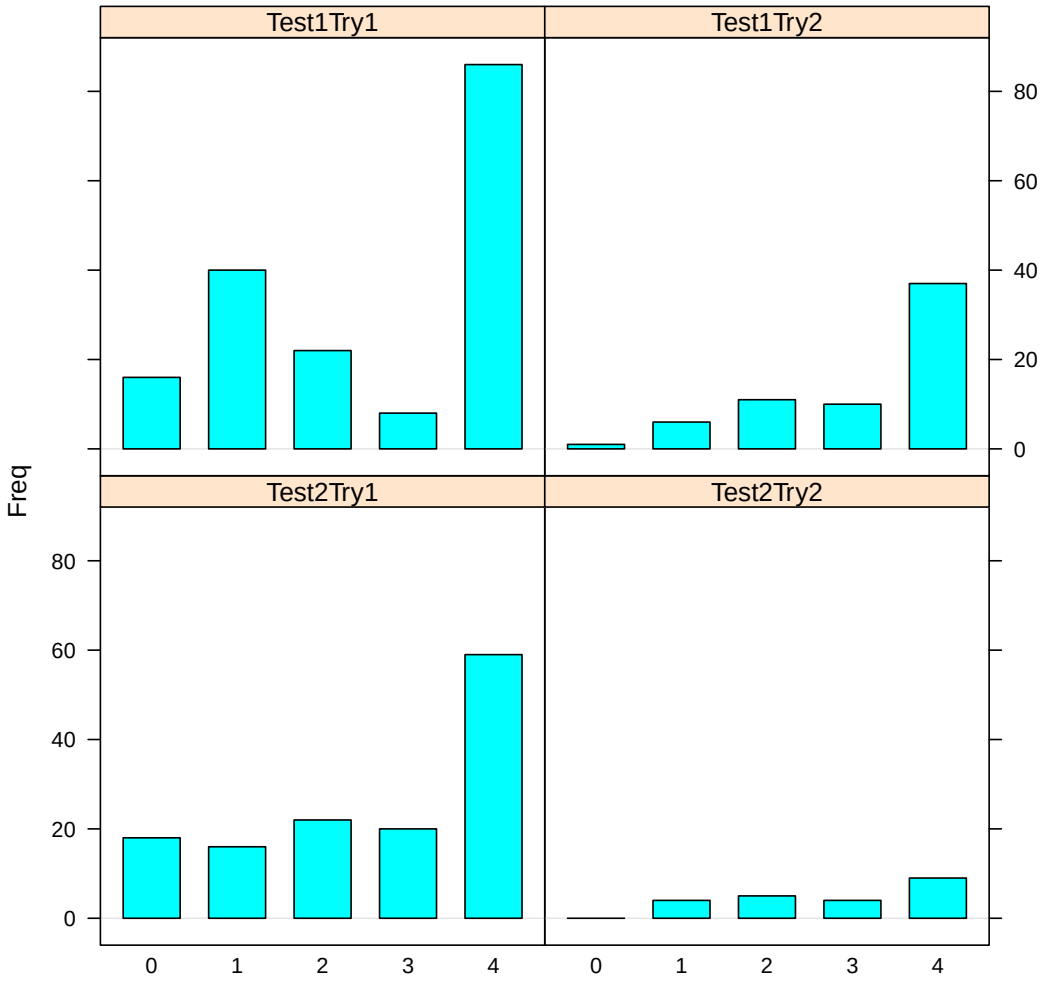

Figure 1: Results for the mastery tests. There are two tests (Test1 and Test2), and for each test students were allowed to resit if they chose (Try1, Try2). The bars represent the number of students with 0, 1, 2, 3 or 4 sections correct.

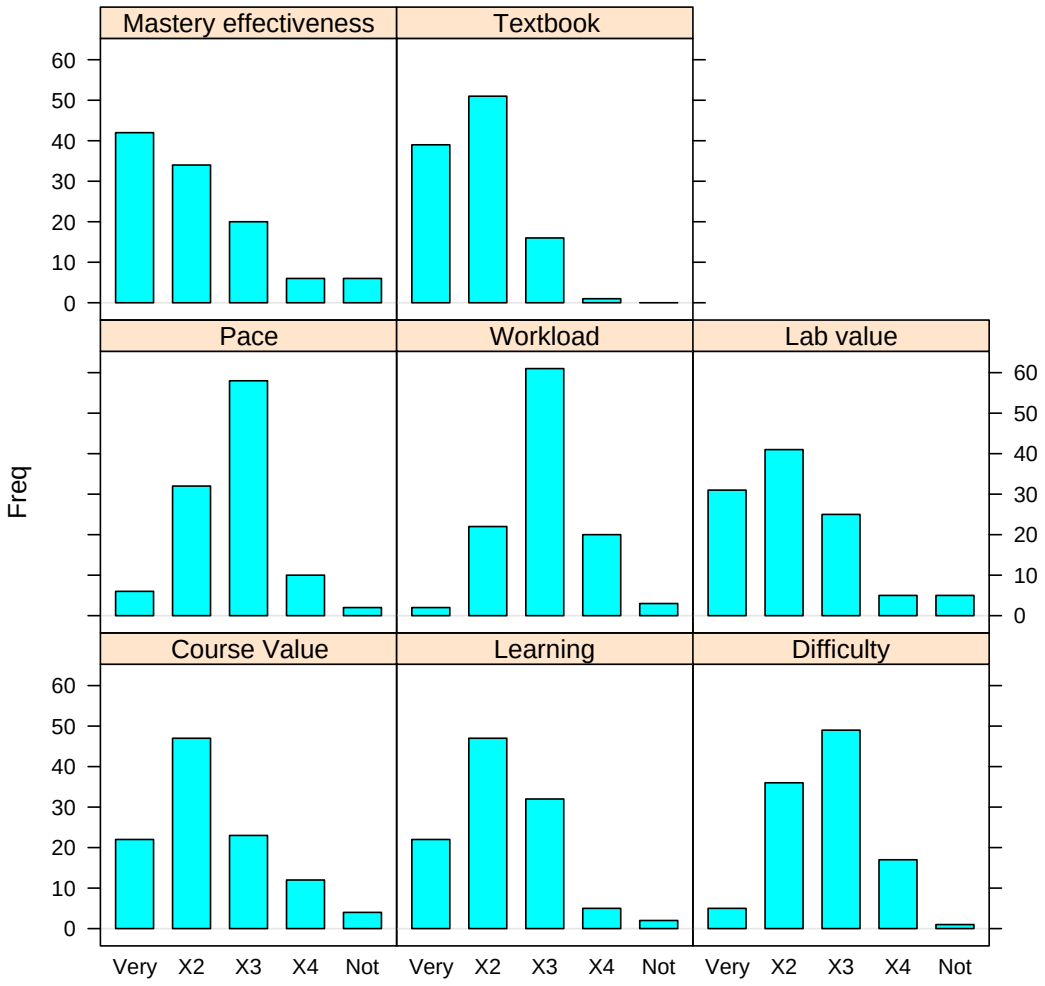

Figure 2: Some of the results for the evaluation questionnaire. Students answered each question on a scale of 1 to 5, where 1 represented "very interesting", "very demanding", "too hard", etc, and 5 represented the opposite. For some questions, 1 is a good outcome, for others, 3 is a good outcome.

	Num. Students	Mid-Sem (50)	Labs (20)
Python	111	30.2	14.6
Not Python	105	23.6	13.6

Table 1: Preliminary results for the second semester Java course comparing students who enrolled in the first semester Python course versus those who did not. Maximum possible marks are indicated in brackets.

Hard topics: end / second half (20)

Any other comments: "Fu@% matrices" (1)

Table 1 shows preliminary results for the second semester Java course with separate results for those who enrolled in the Python course versus those who didn't. Performing a two-sample t-test on the results indicate that the mid-semester marks for the students who took the Python course are significantly better than for other students at the $p = 0.0003$ level. The laboratory marks do not indicate a significant difference. Nevertheless, this is strong evidence that the Python course was very good preparation for the second semester Java course. Informal feedback in our class representative meeting also indicates that the Python course was helpful preparation for the Java course. However, care needs to be taken in attributing the improvement in performance to learning Python per se. It is more plausible that learning some programming language first is good preparation for learning a second programming language. It is also important to note that these findings are preliminary. We hope to have a clearer picture of the outcomes when the results and the evaluation of the Java course are completed.

5 Conclusion

Overall, the introduction of Python as a first programming language has been successful, and preliminary quantitative results indicate that learning Python is a good pre-cursor for learning other languages such as Java [2]. Unlike Java, Python is a joy to teach because it is simple, has few magical incantations, provides immediate feedback, and is also practically useful. The two most successful aspects of the course were the ability to tailor an open source textbook to be very closely aligned with the course, and the use of mastery tests to force students to learn important programming concepts.

[2]Of course, learning Python is also a useful end in itself.

References

[1] Douglas Adams. *Hitchhiker's Guide to the Galaxy*. Harmony Books, 1979.

[2] Jens Bennedsen and Michael E. Caspersen. Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2):32–36, 2007.

[3] S. Dehnadi and R. Bornat. The camel has two humps. *Little PPIG*, 2006. http://www.cs.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf.

[4] Jeffrey Elkner, Allen B. Downey, and Chris Meyers. *How to Think Like a Computer Scientist, Learning with Python*. Number 2nd Edition. http://www.openbookproject.net/thinkcs/python/english2e/, 2008.

[5] Jeffrey Elkner, Allen B. Downey, Chris Meyers, Brendan McCane, Iain Hewson, and Nick Meek. *Practical Programming in Python*. http://www.cs.otago.ac.nz/staffpriv/mccane/Downloads/PracticalProgramming.pdf, 2009.

[6] Donald Knuth. All questions answered. *Notices of the American Mathematical Society*, 49(3):318–324, 2002.

[7] A. Robins. Learning edge momentum: A new account of outcomes in cs1. *Computer Science Education*, 2010. In Press.

[8] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.