Analysis of Student Misconceptions using Python as an Introductory Programming Language

Fionnuala Johnson
University of Glasgow
Glasgow, UK
fionnualajohnson7@gmail.com

Stephen McQuistin
University of Glasgow
Glasgow, UK
sm@smcquistin.uk

John O'Donnell
University of Glasgow
Glasgow, UK
john.t.odonnell9@gmail.com

ABSTRACT

Python has become a popular language for the delivery of introductory programming courses. Two reasons for this are Python's convenience and syntactic simplicity, giving a low entry barrier for beginners and the ability to solve complex problems with short snippets of code. However, students exhibit widespread misconceptions about the meaning of basic language constructs, inhibiting their ability to solve problems and damaging their understanding of fundamental concepts. In this paper, we document our observations of level 1 university students over several years, as well as surveys probing the nature of their misconceptions. We analyze the misconceptions in relation to a notional machine model for Python, and show that many students form inadequate and brittle mental models of the language. Our results indicate that one of the major sources of misunderstanding is the heavy use of overloading in Python. Overloading hides the complexity of algorithms and data structures, often leading students to write code that involves mutability, sharing, copying, side effects, coroutines, concurrency, and lazy evaluation – and none of those topics are accessible to students who haven't yet mastered basic assignments, conditionals, and looping. We suggest that Python, when taught alone, is insufficient as an introductory language: students can gain a firmer grasp of programming fundamentals when Python is presented alongside a complementary low level language that makes a notional machine clear and explicit.

KEYWORDS

introductory programming course; overloading; misconceptions; notional machine model

1 INTRODUCTION

There is a tension among conflicting goals that must be considered when selecting an appropriate introductory programming language. These include simplicity [4, 8], expressiveness, convenience, and industrial relevance [6]. Simplicity is often taken to be *syntactic simplicity*, avoiding boilerplate and offering a low barrier to entry. Lecturers often want to make programming exercises engaging and entertaining, which requires an expressive language that makes routine programming tasks convenient. Many students want to be taught languages that see use in industry. As a compromise that addresses all of these aims, Python is often chosen for introductory programming courses [7, 12].

CEP 2020, January 9, 2020, Durham, United Kingdom
2020. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of Computing Education Practice 2020, January 9, 2020.*

A suitable programming language should help students to understand a *notional machine* and to form suitable *mental models* [14]. These should be constructed easily from the language syntax. Without such mental models, students develop only a vague and inaccurate understanding of basic language constructs and fundamental programming techniques. This leads to poor performance in more advanced study and poor quality programming.

In this paper, we provide evidence that Python's syntactic simplicity hides a complex notional machine [2], allowing students to form inadequate and brittle mental models [10]. Our evidence comprises anectodal observations over several years of several hundred first-year students at a large university, analysis of some of the features of Python, and a series of surveys asking students probing questions about the meanings of basic Python constructs.

Our results indicate that teaching Python alone (*i.e.*, without reference to an explicit notional machine) leads to the formation of misconceptions and misunderstandings by students. We show this through a survey of first year students enrolled in an introductory programming course at a large UK university. This survey highlighted, for example, that in one instance only 4% of respondents can correctly predict the results of a short Python snippet. However, we go on to suggest that Python can be effective when taught alongside languages that share its notional machine, and that are sufficiently low-level that they expose this machine. To achieve this, we describe a visual language that shares Python's notional machine and that aids in the development of a more robust mental model. We also present preliminary findings that support our hypothesis.

Section 2 discusses related work. In Section 3 we analyse some of the features of Python and discuss the difference between syntactic and semantic simplicity, demonstrating the difficulty in inferring a notional machine for Python. Section 4 presents the results of two surveys of students that probe their understanding of the semantics of basic language constructs, confirming that students form incorrect and brittle mental models for Python. In Section 5 we put forward our hypothesis that augmenting the teaching of Python with visual languages that share its notional machine reduces student error, and allows for the formation of more robust mental models. Section 6 concludes.

2 RELATED WORK

Python's selection as a first language has been well studied [3–8, 10–13]. Much of the rationale for selecting Python, rather than other languages, is based upon its simplicity, its industrial relevance, and the ease with which students can "get going quickly on interesting projects" [11]. We acknowledge that, arguably, these are desirable attributes for an introductory programming language.

However, it is common for *syntactic* simplicity to be conflated with *semantic* simplicity. Python's syntax has been developed for its use as a scripting language, and therefore, by design, it is syntactically simple, while semantically complex. We explore this distinction further in Section 3. The suitability of introducing students to programming with a scripting language is strongly dependent on the learning objectives of the course of study. If the goal is to provide students with a medium in which to develop techniques that allow them to solve simple, immediate problems [3], then Python and similar languages are appropriate: a deep understanding of the semantics of the language is less important. However, if the objective is to provide a firm basis for in-depth study into the fundamentals of program design, architecture, performance, and accuracy, as is the case for computing science degrees, then we argue that Python alone is not a suitable introductory language.

Literature that discusses the long-term impact of selecting Python as an introductory language is limited. Oldham [11] reflects upon the adoption of Python in a CS1 course. While confirming that Python was a good choice as an introductory language, the author identifies a number of issues that have arisen as a result. At a high-level, it is noted that "there is more to Python than we care to teach". However, the author identifies more salient issues, including Python's use of dynamic typing, and a number of "concepts and constructs" quirks. In summary, the author states that having taught Python in CS1, students are left requiring a better understanding of compilation, pointers, and parameter passing, noting that these issues are addressed by follow-up teaching in C. This supports our hypothesis that teaching Python alone is insufficient to meet introductory programming teaching objectives.

Finally, we set introductory programming teaching within the context of *notional machines* [2, 10]. We argue that Python's syntax makes it difficult for a notional machine to be constructed by implication, and that one must be explicitly taught. We posit that teaching Python alongside visual languages that share its notional machine allows students to form more robust mental models.

3 SYNTACTIC VS. SEMANTIC SIMPLICITY

Python contains many rich features with complex semantics, but the semantic complexity is often masked by simple syntax. This approach is convenient for advanced programmers who understand the semantics, but it confuses beginners.

In the sections that follow, we describe the overloading of Python's + and += operators. There are many other examples, such as Python's control flow statements that are not covered here, owing to space constraints.

3.1 The + operator

Python simplifies the syntax through extensive use of *overloading*: using the same operator for several different operations. The particular meaning of the operator is determined by the types of the operands. Like many languages, Python overloads the + operator to mean addition of integers and also addition of floating point numbers. This is usually benign, but Python goes much farther, using + also to indicate appending lists and concatenating strings:

```
2 + 3  =  5
[5,9] + [7,3]  =  [5,9,7,3]
"abc" + "def"  =  "abcdef"
```

This simplifies the syntax slightly by reducing the number of operators, but it complicates the semantics. The programmer still needs to be aware of all the underlying operations, and needs to know in addition how to determine the operation from the type.

A particular problem arises when overloading hints at incorrect algebraic properties. The algebraic properties of + over integers and floating point numbers are similar, though not identical, but Python's heavy overloading can be misleading. The associative law holds for +:

```
(a+b) + c = a + (b+c)
([2,3] + [9,8]) + [6,7]  =  [2,3] + ([9,8] + [6,7])
("abc" + "def") + "ghi" = "abc" + ("def" + "ghi")
```

The commutative law holds for integers but fails for lists and strings:

```
[6,7] + [4,5] ≠ [4,5] + [6,7]
"abc" + "def" ≠ "def" + "abc"
```

Python goes even farther, allowing multiplication of lists and strings. Multiplication is treated as repeated addition:

```
3 * 4 = 12
3 * [7,5] = [7,5,7,5,7,5]
3 * "qwe" = "qweqweqwe"
```

The distributive law states an algebraic property of multiplication combined with with addition:

$$x \times (a + b) = x \times a + x \times b$$

The distributive law holds in Python for numbers

```
a = 3
b = 4
c = 5
a * (b + c) = 27
a * b + a * c = 27
```

But it fails for lists and strings:

```
a = 3
b = [4,5]
c = [6,7]
a * (b + c) = [4,5,6,7,5,4,5,6,7,4,5,6,7]
a * b + a * c = [4,5,4,5,4,5,6,7,6,7,6,7]

3 * ("b" + "c") = "bcbcbc"
3 * "b" + 3 * "c" = "bbccc"
```

These are not trivial issues. The whole point of overloading is to use the same syntax for analogous situations, inviting the programmer to apply intuitions from one type to operations over other types. Traversing and copying a list, terminating the list with a pointer to a different list, is fundamentally different from adding two integers; yet Python uses the + operator for both.

3.2 The += operator

A common idiom in programming is to increment a variable (i = i + 1) or to add one variable to another (a = a + b). Python (as well as many other languages) provides the += operator that simplifies the syntax (a += b).

What does a += b mean? It is commonly stated that a += b means a = a + b. (Students often look to Stack Overflow to answer their questions, and many postings on Stack Overflow claim that a += b means a = a + b.) If this were true, it would mean that you can replace a = a + b by a += b, or vice versa, without changing the meaning of a program. But, in Python, a += b *does not* mean the same as a = a + b.

```
a = [1,2]
b = a
a = a + [3,4]
print a  ⇒ [1,2,3,4]
print b  ⇒ [1,2]

a = [1,2]
b = a
a += [3,4]
print a  ⇒ [1,2,3,4]
print b  ⇒ [1,2,3,4]
```

This issue is particularly confusing for beginners because the *value* of a is the same after either `a = a + [3,4]` or `a += [3,4]`, but the latter performs a side effect that can change *another* variable. In the first example above, the variable b is pointing to the list `[1,2]`, and the statement `a = a + [3,4]` calculates a new list and makes a point to the result. Thus a is now pointing to a different memory location, but b still points to the original location. In the second example, the list that a points to is modified in place, and both variables a and b point to the same node they did before.

In order to understand what the += operator means, it is necessary to understand the data representations of lists, as well as the concepts of sharing, copying, and mutability. Yet students will encounter the += operator before they hear of or understand these relatively advanced topics.

4 PYTHON STUDENT SURVEY

As described in the previous section, many years of teaching Python to a large cohort of first year university students have enabled us to identify common misconceptions. In order to build evidence that our anecdotal findings are more widespread, we carried out a survey in Spring 2018. This survey was designed to test the robustness of the mental models of students that had been taught Python. This initial survey focused heavily on the Python language, rather than on more abstract concepts, such as overloading.

Based on the results of this initial survey, we refined our hypotheses, and shifted our focus from obscure parts of the Python language. We developed a new survey, which we carried out in Spring 2019; it is the findings from that survey that we discuss here. The focus of the questions was on material that had been covered in the Python course that students had taken. Each question posed a Python code fragment, and asked what output would be produced, or what the final values of variables would be. In addition, each question asked for the respondent's degree of confidence in their answer, and for any comments or observations that they had.

The survey attracted 42 responses, largely from students participating in a first year Python programming course. Broadly, we found that the confidence of respondents matched their correctness: where students were less confident, they were less likely to be correct. This is a promising result: it can be dangerous for program quality if a programmer is confident in their incorrect knowledge.

In this paper, we limit our discussion of the survey to those misconceptions identified in the previous section. We provide the full survey and anonymised results at http://dx.doi.org/10.5525/gla.researchdata.917.

We asked participants about three different list manipulation operations, where a and b are lists:

(i) `a = a + b`
(ii) `a.append(b)`
(iii) `a += b`

The survey was completed by 42 students. Of these, 17 (40%) answered (i) correctly; 4 (10%) answered (ii) correctly; and 17 (40%) answered (iii) correctly. Overall, performance was poor: none of the three questions was answered correctly by more than 40% of students. Further, only one student (2%) answered all three of the questions correctly. These results are surprising: lists are an important data structure in Python, and the ways in which they are manipulated are covered at length in our first year teaching.

We make two other interesting observations. First, we consider the results for questions (i) and (iii). While these operations are covered in the first year course, they are difficult to teach, given the issues described in the previous section. As a result, teaching favours the `.append()` idiom, and so we believe that students are intuiting the results of the + and += operators. On the face of it, reusing mathematical operators that have clear, well-understood semantics, and applying them to lists, may appear to be successful. However, only 4 respondents correctly answer *both* (i) *and* (iii). This is likely because students believe that these statements are equivalent, when, in reality, they have different semantics. As discussed in the previous section, this is likely perpetuated by Python's overloading of the + and += operators: for other types, such as integers, these statements would be equivalent.

Second, we consider the result for question (ii). This result is surprising because, in our teaching, `.append()` is the preferred idiom for adding to a list. We identify two reasons for why students answer this question incorrectly. First, `.append()` is a method on a list object: our first year Python course does not cover the object-oriented programming paradigm, and so students are unlikely to have a robust mental model of its use. Second, we find that most students are incorrect because they do not understand Python's use of references. In the code snippet provided to students, we update the list b before asking for the final result. Most students do not understand that this also updates the list a. The use of references is a threshold concept [9]: our results support this.

5 AUGMENTING PYTHON TEACHING WITH VISUAL LANGUAGES

In the previous section, we highlighted that misconceptions about Python are widespread. These misconceptions can be attributed to Python's scripting language syntax, where overloading is used to hide semantic complexity. This makes Python a poor choice as an introductory programming language, unless its teaching is augmented with other languages that make the same notional machine explicit, and allow students to build robust mental models.

In this section, we describe one such complementary languages: a visual language, that combines Python's syntax with block and flow diagrams that make the manipulation of data structures and control flow explicit. Importantly, both languages – Python and the visual language – share a notional machine.

In addition, we provide preliminary observations from a survey that evaluates the technique we describe. This survey was made

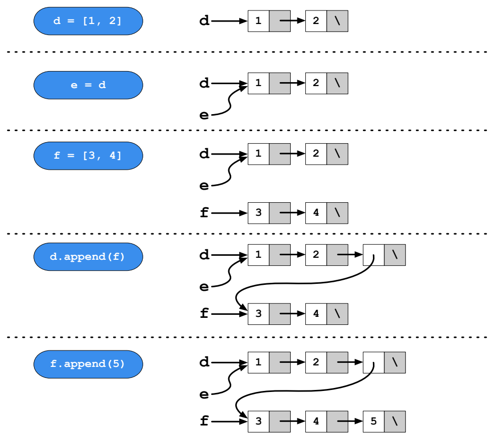

Figure 1: Illustrating Python's `.append()` method

available to the same cohort of students as described previously. However, 15 students responded to this survey, which is lower than the 42 that undertook the first. Both surveys were anonymised, preventing conclusions about progression between the surveys.

Figure 1 gives an example of how this language can be used to describe Python's (`.append()` operation, as discussed in Section 4). Python statements are shown on the left side, with a pictorial representation given on the right. In this way, the visual language creates an explicit notional machine for Python, showing how each list is manipulated by a given statement. In this example, two threshold concepts – references and objects – are exposed, where they would otherwise be hidden by Python's syntax.

The purpose of the visual language is to make data structure manipulation and program control flow explicit in a way that cannot be achieved solely through textual languages. This is particularly important for those students that have not programmed before, and for whom sequential execution, branching, and data structure manipulation may be challenging concepts.

Our preliminary results are promising. Looking at the question from our first Python-based survey that was least well answered – the use of `.append()`, discussed in Section 4 – we find that 10 (out of 15) respondents (67%) correctly answer the question when posed using the visual language, an increase from the 4 (out of 42) respondents (10%) who correctly identified the semantics of the Python statement alone. Similarly, for other questions posed using the visual language, we see an improvement in correctness.

6 CONCLUSIONS

The choice of programming language is central to meeting the objectives of an introductory programming language. The choice of programming *paradigm* is fundamental: whether to use a procedural, object oriented, function, or scripting language. Scripting languages – including Python – aim to make programming fast and convenient. These are desirable features for advanced users, but they introduce complex statements and data structures that are prone to misunderstanding by beginners.

In this paper, we have demonstrated that many students lack a clear understanding of many of Python's core language constructs. We hypothesise that while teaching Python alone is insufficient, teaching it alongside visual languages that share a common notional machine improves students' understanding. We described one such complementary language – a visual language using flow diagrams – and described preliminary observations that show promise for our approach. Other complementary languages (`e.g.`, a low-level, `goto`-based language) may also be beneficial; we omit further discussion of this approach, owing to space constraints.

Teaching additional languages is likely to require additional teaching hours. However, this approach satisfies the conflicting goals of introductory programming courses: students learn an easy-to-use, industrially-relevant language in Python, whilst also gaining a deeper understanding of its semantics through a low-level language. This allows students to develop robust mental models that support further study beyond their introductory course.

REFERENCES

[1] Edsger W Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.
[2] Benedict Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.
[3] Hans Fangohr. A comparison of c, matlab, and python as teaching languages in engineering. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, pages 1210–1217, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
[4] Linda Grandell, Mia Peltomäki, Ralph-Johan Back, and Tapio Salakoski. Why complicate things?: introducing programming in high school using python. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 71–80. Australian Computer Society, Inc., 2006.
[5] Alexander Holt, Sarah Rauchas, and Ian Sanders. Introducing python into the first year curriculum at wits. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITICSE '06, pages 335–335, New York, NY, USA, 2006. ACM.
[6] Tony Jenkins. The first language-a case for python?, 2004.
[7] Vambola Leping, Marina Lepp, Margus Niitsoo, Eno Tõnisson, Varmo Vene, and Anne Villems. Python prevails. In *Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, page 87. ACM, 2009.
[8] Linda Mannila, Mia Peltomäki, and Tapio Salakoski. What about a simple language? analyzing the difficulties in learning to program. *Computer Science Education*, 16(3):211–227, 2006.
[9] Jan Meyer and Ray Land. *Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practising within the disciplines.* University of Edinburgh, 2003.
[10] Donald A Norman. Some observations on mental models. In *Mental models*, pages 15–22. Psychology Press, 2014.
[11] Joseph D. Oldham. What happens after python in cs1? *J. Comput. Sci. Coll.*, 20(6):7–13, June 2005.
[12] Atanas Radenski. Python first: A lab-based digital introduction to computer science. In *ACM SIGCSE Bulletin*, volume 38, pages 197–201. ACM, 2006.
[13] Ian D Sanders and Sasha Langford. Students' perceptions of python as a first programming language at wits. *ACM SIGCSE Bulletin*, 40(3):365–365, 2008.
[14] Juha Sorva. Notional machines and introductory programming education. *ACM Transactions on Computing Education (TOCE)*, 13(2):8, 2013.
[15] Vesa Vainio and Jorma Sajaniemi. Factors in novice programmers' poor tracing skills. In *ACM SIGCSE Bulletin*, volume 39, pages 236–240. ACM, 2007.