PyToPseu: Automatic Natural-Language Formulations of Programming Constructs to Avoid Misconceptions

Jean-Philippe Pellet[0000−0001−7559−397X] and
Patrick Wang[0000−0003−3117−8189]

University of Teacher Education, Lausanne, Switzerland
{jean-philippe.pellet,patrick.wang}@hepl.ch

Abstract. Introduction to programming remains one of the first delicate topics in computer science education. In this poster proposal, we describe our efforts to equip beginners with a tool which, from Python code, generates a line-by-line natural-language description of the code. This tool is designed to help students understand programming constructs and avoid common misconceptions. The tool is not based on LLMs, but aims to stay very close to the actual code and is based on a set of rules that map Python constructs to their natural-language interpretations. We believe that this approach can enhance the learning experience for beginners, especially those who may struggle with understanding the syntax and semantics of basic programming constructs.

Keywords: programming education · misconceptions · Python.

1 Introduction & Context

In recent years, computer science courses have become increasingly popular in schools and universities. Programming is part of the main topics in these courses. Even with the rise of LLMs, learning basic programming concepts remains essential—whether one believes that LLMs should be involved in programming courses or not. Indeed, in order to be able to evaluate and use adequately code produced by LLMs, understanding the basic concepts of programming is still crucial.

However, many students struggle with programming concepts, and are subject to recurring misconceptions [1]. Misconceptions can be syntactical or semantical, and can depend on the programming language used; but misconceptions common to whole families of programming languages also exist.

In our context of introductory programming courses in tertiary education, where the curriculum dictates the use of a textual programming language, it is quite common to use Python as the first language. Python has often-cited advantages, such as its readability and simplicity, which make it a popular choice for beginners. Certain teachers skip pseudocode altogether, arguing that Python, most of the time, is close enough to pseudocode to be used directly without loss of readability and with the advantages of lack of ambiguity and executability.

But even with these advantages, we have found out time and again that certain constructs remain systematically problematic for students to use correctly. We found that reading out loud typical faulty code from students, insisting on its semantics by reformulating it in more natural language, helped students find out why it was faulty. This led us to the idea of creating a tool that would automatically generate such natural-language descriptions of Python code, in order to help students understand programming constructs and avoid misconceptions.

Our tool, PyToPseu, automatically generates natural-language descriptions of Python code. PyToPseu is work in progress and, at the time of writing, no systematic data collection or evaluation has been done yet, but the tool is already available online[1] and is being developed as an open-source project[2] with feedback from select students and teachers. We would be looking forward to more feedback from the community based on the poster and the tool itself.

2 Natural-Language Descriptions of Python Code

PyToPseu takes Python code as input and generates a natural-language description of the code. In its current form, it comes in the form of an online code editor (based on CodeMirror[3]) where users can write Python code, together with syntax highlighting and autocompletion. The tool then generates a natural-language description of the code, line by line, which is inserted directly in the code as comments in a non-obstrusive and nicely formatted way. (Currently, the tool produces comments in French, but we would add English support if there is interest in the community.)

```
"""
Code                                    |   Interprétation
-----------------------------------------------------------------------------
"""                                     #|
                                        #|
words = ["i", "love", "programming"]    #|   dans words, stocke une liste avec les éléments "i", "love" et "programming"
                                        #|
total = 0                               #|   dans total, stocke 0
for i, word in enumerate(words):        #|   pour chaque élément de words
                                        #|   |      └ (qu'on va appeler word et numéroter i depuis 0):
  print(i, word)                        #|   |   affiche i et word
  total += len(word)                    #|   |   ajoute la longueur de word à total
                                        #|
print(f"Total length: {total}")         #|   affiche l'expansion de 'Total length: {total}'
```

Fig. 1. Screenshot of PyToPseu, showing the code editor with a Python

At its core, the tool calls Python's parser to create an abstract syntax tree (AST) of the code, and then traverses this AST to generate a natural-language

[1] https://jp.pellet.name/hep/pytopseu/
[2] https://github.com/jppellet/PyToPseu
[3] https://codemirror.net/

description of each construct.[4] In order to make the output natural, it is actually more involved than a straightforward translation of the AST. Some special cases are handled differently to make the output sound more natural, and some constructs are handled differently depending on their context (see below for examples).

PyToPseu is aimed at beginners: its use with more complex constructs such as Python decorators, metaprogramming, or more advanced object-oriented programming in not where its focus lies. The resulting explanations there are less likely to shed light on what is actually going on, since the semantics of the code should, at a certain stage, be understood at a different level and cannot readily be explained line by line. Referring to Schulte's block model [2], PyToPseu is on the "Function/Atom level". As such, we actually rather view it as a tool to help production of code rather than a tool for whole-program comprehension [3].

Here is a non-exhaustive list of the misconceptions or "recurringly problematic constructs" that PyToPseu aims to help students with:

- **Assignment**. We disambiguate the fundamentally asymmetric nature of the `=` operator. `a = 5` is translated as "in `a`, store 5". Type annotation are supported, such that `a: int = "code"` is translated as "in `a`, meant for an whole number, store the string `"code"`", which strongly hints at the type mismatch without explicitly stating it. Augmented assignment operators such as `+=` are translated more specifically; regular assignments of the form `a = a + x` are treated as augmented assignments.
- We handle **subscripts** for strings, lists, and dictionaries differently, when the type information permits, such that `a[i]` is translated as "element number `i` of `a`" for lists, "character `i` of `a`" for strings, and "the value associated with the key `i` in `a`" for dictionaries, which improves the readability.
- **Comparison operations** are translated as expected in an `if`/`while` statement; for instance, `if a == 4` is translated as "if `a` is equal to 4". However, assignement of thusly produced boolean values to variables is frequently seen as problematic, so we translate `cond = a == 4` as "in `cond`, store true/false according to whether `a` is equal to 4". Similary, `if cond` is translated as "if `cond` is true"—students are often confused by the lack of operator.
- **Loops** undergo special treatment because of the common confusion between iterating on indices or on elements, or both with `enumerate`. Here are some of the special cases that are treated separately:
  - `for i in range(x)` is translated as "repeat `x` times, counting with `i`", but `for i in range(a, b)` is translated as "repeat for each item in the range from `a` to `b`, which we'll call `i`".
  - `for i in range(len(a))` is translated as "repeat as often as there are elements in `a`, counting with `i`". This insists that `a` must be some container and that `i` is a index.
  - `for elem in a` is translated as "repeat for each element of `a`, which we'll call `elem`". This insists that `a` must be some container and that `elem` is actually one of its elements.

[4] This unfortunately means that syntactically incorrect code cannot be processed.

  - `for i, elem in enumerate(a)` is translated as "repeat for each element of `a`, which we'll call `elem` and number with `i`".
- **Common functions** such as `sqrt`, `round`, `abs`, etc. are translated, as well as methods on common data types such as `str` (`endswith`, `strip`, etc.) and `list` (`append`, `insert`, `clear`, etc.), Lone expressions that are neither stored nor used later show up as such. For instance, a line whose total content is `math.sqrt(4)` is translated as "the square root of 4". Reading just this, without a print or store instruction, helps indicate that this line does not do anything useful in the program.
- In general, **statements** that are expected to return `None` are translated with a verb of action (e.g, "print", "send", "wait"), whereas expressions are substantives (e.g., "the square root of 4", "the sum of the elements of `a`"). Of course, this is just a heuristic: complex functions may both compute a value and have side effects, for instance, and we have no easy way to determine that from the AST.

One could argue that LLMs, given a faulty program, can produce a natural-language description of the code and suggest improvements as well. However, we precisely believe that an LLM is *too* capable to be used as a tool for beginners and will lead them to simply skip the understanding of the code. PyToPseu, on the other hand, is designed to stay very close to the actual code and to help students understand the constructs they are using, rather than providing a black-box solution that may not be fully understood.

3 Outlook

We are currently working on extending PyToPseu to support more constructs, but our main focus is on the modalities of use in the classroom or during exercise sessions. Our students use Visual Studio Code and one way to integrate our tool would be to create a VS Code extension that annotate code on demand. Of course, feedback from students and peers and structured evaluation of the tool will be crucial to improve it.

References

1. Chiodini, L., Moreno Santos, I., Gallidabino, A., Tafliovich, A., Santos, A.L., Hauswirth, M.: A curated inventory of programming language misconceptions. In: Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1. pp. 380–386 (2021)
2. Schulte, C.: Block model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In: Proceedings of the fourth international workshop on computing education research. pp. 149–160 (2008)
3. Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., Paterson, J.H.: An introduction to program comprehension for computer science educators. Proceedings of the 2010 ITiCSE working group reports pp. 65–86 (2010)