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


Jean-Philippe Pellet and  **Patrick Wang**

University of Teacher Education, Lausanne, Switzerland

jean-philippe.pellet@hepl.ch(✉), patrick.wang@hepl.ch

ISSEP 2025
September 8–11
Trier, Germany

Introduction & Context

Introduction to programming remains one of the first delicate topics in computer science education. We present a tool which, from Python code, generates a line-by-line natural-language description of the code.

Its intended use is first and foremost a **reading and interpretation tool for existing code**, but it can be used to verify the meaning of code being written as well.

Prevent common mistakes

Because they always pop up and we know them well by now

Runs locally or in browser

No server, can do without internet connection

Not an LLM

Not too powerful to replace students' thinking processes

***Work in Progress** as of Summer 2025!*

Works on the AST

… so, syntactically invalid code cannot get processed

Can be integrated in VS Code

Because output is in the form of Python comments "nicely" formatted

Example Outputs

Assignments

```
side = 4              #   in side, store 4
side: int = 4         #   in side, intended for a an integer number, store 4
                      #
area: int             #   prepare area to store an integer number
area = side * side    #   in area, store the product side × side
```

Insist on asymmetry of = Interpret type hints Clarify : vs =

```
side += 2             #   add 2 to side
side = side + 2       #   add 2 to side
```

Desugar op= syntax, make it look like a normal reassignment Use vocabulary to clarify that a variable is being updated

```
val = math.sqrt(area)             #   in val, store the square root of area
length = len(text)                #   in length, store the length of text
```

Interpret common function names

For Loops

```
n = 0                             #   in n, store 0
for i in range(5):                #   repeat 5 times (counting with i from 0):
  n += i                          #   |   add i to n
```

*Use "repeat" and interpret the part after **in** Insist that we are counting from 0*

```
ng = 0                            #   in ng, store 0
for c in text:                    #   repeat for each item of text (which we'll call c):
  if c == "g":                    #   |   if c is equal to "g":
    ng += 1                       #   |   |   add 1 to ng
```

Basic for-each with container Explanation for loop var

```
ng = 0                            #   in ng, store 0
for i in range(len(text)):        #   repeat as many times as there are elements in text
                                  #   |       └ (counting with i from 0):
  if text[i] == "g":              #   |   if the ith element of text is equal to "g":
    ng += 1                       #   |   |   add 1 to ng
```

*Interpret the common **range(len(…))** construct*

```
for i, c in enumerate(text):      #   repeat for each item of text
                                  #   |       └ (which we'll call c and number i from 0):
  print(f"at pos {i}, we have {c}")  #   |   display the expansion of 'at pos {i}, we have {c}'
```

*Interpret **enumerate***

```
for c in len(text):               #   repeat for each item of the length of text (which we'll call c):
  print(c)                        #   |   display c
```

Try to make wrong constructs sound wrong

Functions

```
def add_abs(x, y) -> int:         #   define the function add abs, which accepts 2 arguments,
                                  #   |       └ x and y, and which returns an integer number, like so:
    return abs(x) + abs(y)        #   |   get out of the function returning the sum of ⟨the absolute value of x⟩ and
                                  #   |   |       └ ⟨the absolute value of y⟩
                                  #
def b(x, y, z) -> int:            #   define the function b, which accepts 3 arguments, x, y and
                                  #   |       └ z, and which returns an integer number, like so:
    return x + 1                  #   |   get out of the function returning the sum of x and 1
```

*Spell out function, arguments, return type Insist that **return** terminates the function*

Conditions, If Statements (and while statements)

```
if age ⩾ 18:                      #   if age is greater than or equal to 18:
  print("you can drive")          #   |   display "you can drive"
```

Provide textual versions of symbolic comparison operators

```
if value is None:                 #   if value is empty:
  print("missing")                #   |   display "missing"
```

Interpret some common conditions with more intuitive names

Common values for the % operator are explained

```
if len(text) % 2 == 0:                          #   if the length of text is an even number:
    print("text has an even number of characters")  #   |   display "text has an even number of characters"
else:                                           #   else:
    print("text has an odd number of characters")   #   |   display "text has an odd number of characters"
```

```
if 0 ⩽ index < 10:                #   if index is between 0 (inclusive) and 10 (exclusive):
  print("index is valid")         #   |   display "index is valid"
```

Support for multiple comparisons

```
text: str = "programming"         #   in text, intended for a a string, store "programming"
                                  #
cond1 = "gr" in text              #   in cond1, store true/false according to if "gr" is in text
cond2 = text == text.upper()      #   in cond2, store true/false according to if text is equal to
                                  #           └ an uppercase copy of text
                                  #
if cond1:                         #   if cond1 is true:
  if not cond2:                   #   |   if cond2 is false:
    print("cond1 && !cond2")      #   |   |   display "cond1 && !cond2"
```

Explanations for storing booleans in variables and testing conditions without comparison operators

Lists (and tuples, sets, and dicts)

```
l: list[int] = []                 #   in l, intended for a list of integer numbers, store an empty list
l = [1, 2, 3]                     #   in l, store a list with items 1, 2 and 3
ll: list[list[int]] = []          #   in ll, intended for a list of lists of integer numbers,
                                  #   |       └ store an empty list
                                  #
l: int = list()                   #   in l, intended for an integer number, store an empty list
```

List/tuple/set/dict literals and constructor functions

Type mismatch: this hopefully sounds wrong

```
l.append(42)          #   at the end of l, append 42
l.extend([33, b])     #   at the end of l, append elements 33 and b
l2 = l.copy()         #   in l2, store a copy of l
l.index(42)           #   the position in l of 42
l.count(30)           #   the number of occurrences in l of 30
l.sort()              #   sort l
l.reverse()           #   reverse the order of the items of l
l.pop()               #   from l, remove the last item
```

Make a difference between methods/functions with side effects (use a verb) and computing a value (use a noun)

*Difficult with those that do both, e.g. **pop**.*

```
a = lambda: 42                    #   in a, store an anonymous function without any arguments
                                  #   |       └ which returns 42
a = lambda x: x ** 4              #   in a, store an anonymous function which accepts one argument, x, and
                                  #   |       └ which returns x to the power of 4
```

Lambda functions are also supported (albeit without type annotations)

Future Work:

- *Usefulness tests in classrooms*
- *Iteration with students on best phrasing*
- *More precise phrasing with type info from mypy/pylint*
- *If useful: better technical integration with IDEs*

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] Qian, Y., Lehman, J.: *Students' misconceptions and other difficulties in introductory programming: A literature review*. In: ACM Transactions on Computing Education (TOCE) 18(1), 1–24 (2017)

[3] 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)

[4] Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., Paterson, J.H.: *An introduction to program comprehension for computer science educators*. In: Proceedings of the 2010 ITiCSE working group reports pp. 65–86 (2010)

Relevance of reference in the context of this poster:

[1] List of misconceptions classified by programming language
[2] Literature review of misconceptions
[3] Introduction of the block model and the various levels of understanding of a program
[4] Comparison of models for program comprehension