# Transforming Pseudocode into Python Code

Kevin Sittser
Texas A&M University
College Station, TX USA
ksittser@tamu.edu

## Abstract

*Generating functioning code from pseudocode is a fascinating functionality that will serve to make coding easier and more accessible to those who may struggle with the precise syntax necessary to create a programming project. A number of projects have made successful attempts at the opposite problem of transforming source code into pseudocode, but I have found rather few that convert in the other direction.*

*This project aims to use machine learning techniques to read a list of lines of natural language pseudocode and modify them into Python code readable by a Python interpreter. The goal is to allow users with different conventions or styles of coding, or those unfamiliar with coding syntax altogether, to reliably produce Python code from English-like input.*

## 1. Introduction

Pseudocode is a ubiquitous method of writing code-like text meant not to be run by a machine but to illustrate a process to a human reader. Pseudocode de-emphasizes syntactic specifics, allowing the writer to write in a human-readable way without concern for the exact phrasing of each command. This keeps the focus on the algorithm or process itself and is also in many ways easier than writing machine-readable code, since the syntax of a particular language does not need to be remembered or followed closely.

This particular benefit of pseudocode is what this project focuses on. The purpose of this project is that a user may be able to write a simple Python program with minimal knowledge of Python's actual syntax; the user needs only to be familiar with the general structure of a program written in a similar language. A user will be able to input commands written in natural-language pseudocode, and the system will produce corresponding Python code that closely reflects the user's input.

This project uses machine translation techniques to perform this transformation from pseudocode into correct Python syntax. Machine translation, a problem that has garnered a great deal of interest since the 1930s [1], has shown itself to be increasingly accurate and useful in software such as Google Translate. The first documented attempt at designing a mechanized translation system was in 1933, when Petr Troyanskii patented an idea for a "translating machine," which would consist of long belts containing dictionaries of words in six different languages. The user would locate words in the target language and the intended grammatical role, and the machine would record them on paper, after which a typist would type the text with the correct inflections [2]. Today's translators are fully automated and can provide instantaneous translations with incredible accuracy with no more input than the user's input sentence and languages.

Machine translation has proven an effective technique for the problem presented here of translating between pseudocode and source code. The majority of projects on this topic consider the opposite problem of converting source code to machine code. A variety of Deep Learning methods have been used, mostly designed originally for natural language machine translation. Alhefdhi et al. use a Neural Machine Translation (NTM) to achieve a code-to-pseudocode system that achieves a 54.78% BLEU score, which measures pseudocode quality [3]. Oda et al. use a Statistical Machine Translation (STM) method to achieve a similar 54.08% BLEU score for pseudocode written in English and 62.88% score for pseudocode written in Japanese [4]. Kulal et al. use an Long Short-Term Memory (LSTM) neural network system to convert fairly long pseudocode programs into C++ and achieve up to 87% accuracy in creating functional lines of code individually, though success in translating entire working programs is minimal [5].

The problem I am working on, of translating pseudocode to Python code, has proven to be a bit less successful. Using neural machine translation techniques, Rahit et al. achieved 74.4% accuracy in translating lines of pseudocode into syntactically correct Python code [6]. My project aims to achieve similar results using techniques from natural language processing (NLP) used widely in translating natural languages such as English and Spanish.

```
def _check_test_runner ( app_configs = None , ** kwargs ) : ...define the function _check_test_runner with app_config set to None and kwargs dictionary as arguments.
  from django . conf import settings ..... from django.conf import settings into default name space.
  weight = 0 ..... weight is an integer 0.
  if not settings . is_overridden ( 'TEST_RUNNER' ) : ..... if call to the settings.is_overridden with string 'TEST_RUNNER' evaluates to boolean False.
    try : ..... try,
      settings . SITE_ID ..... settings.SITE_ID.
      weight += 2 ..... increment weight by integer 2.
    except AttributeError : ..... if AttributeError exception is caught,
      pass ..... do nothing.
    try : ..... try,
      settings . BASE_DIR ..... settings.BASE_DIR.
    except AttributeError : ..... if AttributeError exception is caught,
      weight += 2 ..... increment weight by integer 2.
```

Figure 1: Example data

### 1.1. Dataset

For this project, I use the *Django* dataset developed for Oda et al.'s aforementioned paper on translating Python source code to pseudocode, the opposite problem of my project [4]. This data contains 18,806 lines of Python code which have been translated by hand into corresponding pseudocode. This code is derived from a project written in Python's Django framework, and the pseudocode was written by a single programmer, so it is somewhat lacking in variety in terms of coding purpose and pseudocode style [4]. However, it contains a wide variety of statements ranging from simple assignment statements to imports, control blocks, and generators and will more than suffice to demonstrate the capabilities of my system. Some examples of Python lines and their corresponding pseudocode are shown in Figure 1.

The simplest statements to translate are most likely to be those that are more or less formulaic, such as equations and assignments, whose code and pseudocode ought to match very closely, with each word in the pseudocode simply replaced with a symbol. This will be my primary focus at first, in order to ensure that the most basic of commands can be easily converted before I try anything more complex. I believe it will also be possible to translate statements as complex as control statements such as if-statements and loops, as long as the conditions therein are not themselves excessively verbose. The purpose of my system will be to provide a basic line-by-line translation, without regard to surrounding code; therefore, it will not be able to perform the difficult task of translating entire programs but only individual lines.

## 2. Methods

### 2.1. Data Preparation

In the interest of lowering training time, I eliminated from the dataset any data containing more than 75 characters in the pseudocode or Python code. This lowered the size of my dataset substantially from 18,806 to 11,382 lines of code but resulted in a dramatic decrease in training time. As the long data increased training time by an order of magnitude, this modification was necessary to ensure feasible training times.

While most NLP tasks require preprocessing to omit the effects of meaning-free features like punctuation and conjugation, I do not perform any preprocessing on this data apart from tokenization. Punctuation in the pseudocode is usually kept to a minimum anyway, and the punctuation that is in the pseudocode is usually part of a variable name or important to the meaning of the statement, so it must remain in the data. Furthermore, the precise nature of code requires that alphanumeric words not be stemmed to remove affixes, since this would create a totally new variable and not, as in natural language, simply remove an unneeded affix.

### 2.2. Translation

One of the most common, state-of-the-art models for machine translation is the Sequence to Sequence (seq2seq) model, which was developed by Google for its Google Translate service [7]. In this project, I use two different versions of seq2seq, a character-level model and a word-level model.

seq2seq uses two LSTM networks in an encoder-decoder model, where one LSTM acts as an "encoder" and the other as a "decoder." To train the system, the encoder system converts the input pseudocode excerpt into state vectors and runs them through the LSTM network. The decoder system then initializes to the final state of the encoder LSTM and converts the target Python code into vectors as well. Via the "teacher forcing technique, the output from each time step in the LSTM is the input for the next. This trains the decoder network to generate output based on the input sequence. To translate a new line of pseudocode, the system transforms the text into state vectors and feeds a special START unit into the decoder network to gain the most probable first unit of the Python code (which could be a character or a word,

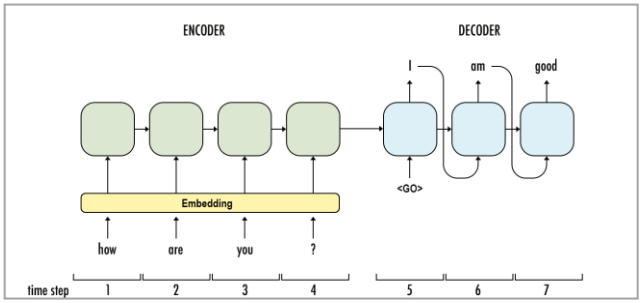


Figure 2: Overview of seq2seq [10]

```
Input sentence: call the function cache.close.
Decoded sentence: cache_suff = cclared_aby

Input sentence: substitute self.default_timeout for timeout.
Decoded sentence: timeout = self . defiglty_equn
```

Figure 3: Nonsensical words generated from the character model

depending on the implementation) via argmax. These two units are fed into the network to produce the second unit, and so forth until a special END unit is generated. This yields the machine's predicted Python code for the pseudocode [8,9]. A visual overview of this process is shown in Figure 2.

My first model is a character-level model based on one created by Keras to translate natural language sentences from English to French [8]. This model generates output-language sequences character by character. The other is based on a model created by Harshall Lamba to translate from English to the Marathi language [9]. I repurpose these models for training on pseudocode rather than natural language and train them on my pseudocode data for varying lengths of 50, 100, and 300 epochs to evaluate the progress of the training.

### 2.3. Evaluation

To evaluate my system, I translate 100 random pseudocode statements from the set. I evaluate the validity of these translations based on how similar the generated Python code is to the pseudocode that it came from. While the purpose of this system is to produce Python code that precisely matches the intention of the pseudocode it came from, naturally my system is not expected to be fully successful. For a generated statement that does not match the code provided in the dataset, I evaluate its validity based on whether it is structurally similar to the intended line of code, and whether it is itself a valid line of Python. I evaluate each generated statement on a three-point scale: 2 for a perfect match to the dataset (or a statement that has the exact same effect), 1 for a statement that has the correct structure and is valid Python (or nearly valid) but is not fully correct, and 0 for a statement that is not similar to the intended code and/or has no semblance of being executable Python. In Figure 4 I report the counts of each score found for varying amounts of training.

## 3. Results

The character-based model performs well but pales in comparison to the word-based model. As can be seen in Figure 3, the character-based model is prone to generating nonsensical words which may or may not seem to relate to a word in the original pseudocode. This is a natural and expected result of generating code character by character but adds an element of randomness that will be too problematic for this project.

The word-based model is much more successful, able to correctly translate 99% of training data and 52% of testing data. While this is well below Rahit et al.'s aforementioned result of 74.4% accuracy, it displays an extremely effective translator. Further, an additional 35% of the testing data in my experiment was fairly close to correct, being structured similarly to the expected code and syntactically valid or nearly so.

I had substantially more success with 300 epochs of training versus 50 or 100, at the expense of long training time. Summaries of the accuracy and loss with increased training are shown in Figure 4 and Figure 5, and sample translations are shown in Figure 6 and Figure 7. It is clear from the data that extensive training is very helpful to this system and yields better results; in fact, it seems likely that training with even more epochs could lead to even better accuracy of translations.

As expected, my system is most capable of translating simple statements; lines containing only one or two words (such as "pass" or "try" and simple "define" or "return" statements) are nearly always translated correctly since they are a very simple formula of one or two words. More complex statements such as control statements ("if" and

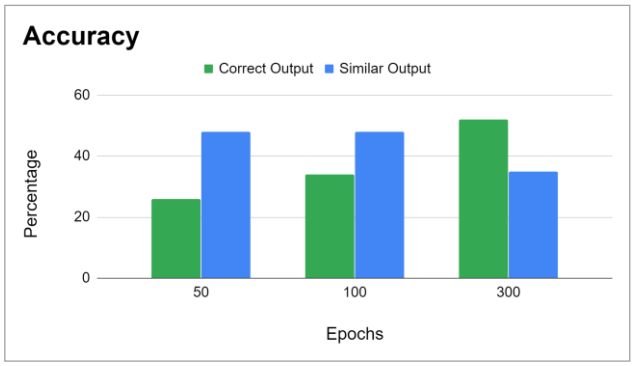


Figure 5: Accuracy with increased training


Figure 4: Loss with increased training

"for" statements), which tend to be longer, are often translated poorly, especially if they contain parentheses. In general, punctuation marks are one of the weakest points of my system. The bad translations in my results frequently have mismatched parentheses and often excessive parentheses where they don't make sense. Since this system is generating word by word, it has no understanding of matching parentheses and that one parenthesis needs to be matched by another later. Strings are another common source of confusion for the system, since the contents of the strings can be absolutely anything and don't have any meaningful relationship with the rest of the line. With that in mind, however, the program is still remarkably capable of producing some statements with correctly matched parentheses.

One other frequent failure of my system is its tendency to produce variable names that were in the dataset but not in the particular statement being translated. Often a statement will be written entirely correctly and match the expected translation, except that a variable name will be replaced with a totally different one. This seems to be a result of the system's skill at pattern matching without real understanding of what it is reading.; it will produce the correct pattern of words and punctuation without actually getting the correct words.

## 4. Conclusions

Machine learning models based on NLP prove to be an effective method of translating pseudocode to source code. In this project I show that the state-of-the-art models using seq2seq LSTM models via encoder-decoder methods are capable of producing real Python code that matches the meaning of the pseudocode it is derived from.

The deficiency of my model in reliably generating proper punctuation in statements shows one of its limitations; all elements of the data are treated as words, so the model is not good at generating correct syntax by, for instance, matching parentheses and brackets, which is not something that words have to do.

My results are a great example of machine learning's affinity for pattern matching; my model was in many cases able to produce syntactically correct Python that exactly matched the format of the intended code but had completely different words. The replacement of one word with another word is considered by this model to be equivalent to confusing punctuation marks, which is another severe limitation of this model. Improvements are needed in the system's ability to gather named entities from the pseudocode.

Extensive testing with more datasets, in particular using pseudocode written by multiple different people with different styles, is needed to further assess the success of the model used in this project. My dataset was relatively small, with only about 11,000 lines of code in my training, so a much larger set would be likely to train more accurately as well as give the system more coding structures and words to recognize. Also, since I omitted long lines of code from my dataset, testing with longer lines of code as well as multiline sections of code is another clear next step in this project's progress.

```
Source Pseudocode:  valid is boolean False.
Expected Python:    valid = False
Generated Python:   request = False

Source Pseudocode:  define the method __init__ with 2 arguments: self and variables.
Expected Python:    def __init__ ( self , variables ) :
Generated Python:   def __init__ ( self , obj ) :

Source Pseudocode:  if ValueError exception is caught,
Expected Python:    except ValueError :
Generated Python:   except ValueError :

Source Pseudocode:  if plan_node is contained in loader.applied_migrations,
Expected Python:    if plan_node in loader . applied_migrations :
Generated Python:   if new_name in config :

Source Pseudocode:  remove dirname from the dirnames.
Expected Python:    dirnames . remove ( dirname )
Generated Python:   os . makedirs ( filename )

Source Pseudocode:  for every receiver, _ and _ in receivers,
Expected Python:    for receiver , _ , _ in receivers :
Generated Python:   for k in enumerate ( self . migrated_apps ) :

Source Pseudocode:  return ZERO.
Expected Python:    return ZERO
Generated Python:   return

Source Pseudocode:  define the function walk_to_end with 2 arguments ch and input_iter.
Expected Python:    def walk_to_end ( ch , input_iter ) :
Generated Python:   def __nonzero__ ( self ) :

Source Pseudocode:  try,
Expected Python:    try :
Generated Python:   try :

Source Pseudocode:  increment self.fixture_object_count by objects_in_fixture.
Expected Python:    self . fixture_object_count += objects_in_fixture
Generated Python:   self . stream . startswith ( '.' )
```

Figure 7: Sample translations with 100 epochs of training (green text indicates a perfect translation by the model, blue a close one with correct structure, and orange a wrong translation)

```
Source Pseudocode:  define the method IE_sanitize with arguments self and filename.
Expected Python:    def IE_sanitize ( self , filename ) :
Generated Python:   def __delitem__ ( self , :

Source Pseudocode:  return new_value.
Expected Python:    return new_value
Generated Python:   return new_value

Source Pseudocode:  _wrapped is None.
Expected Python:    _wrapped = None
Generated Python:   self . error_dict = _urlconfs

Source Pseudocode:  define the method validate_unique with an argument self.
Expected Python:    def validate_unique ( self ) :
Generated Python:   def validate_unique ( self ) :

Source Pseudocode:  if not,
Expected Python:    else :
Generated Python:   else :

Source Pseudocode:  call the method logging.Handler.__init__ with an argument self.
Expected Python:    logging . Handler . __init__ ( self )
Generated Python:   self . close ( )

Source Pseudocode:  substitute stream for self.stream.
Expected Python:    self . stream = stream
Generated Python:   self . stream = callback

Source Pseudocode:  otherwise substitute args for key.   with lock,
Expected Python:    with lock :
Generated Python:   level [ - 1 ]

Source Pseudocode:  define the method inner with argument check.
Expected Python:    def inner ( check ) :
Generated Python:   def default ( self , ) :

Source Pseudocode:  substitute self.deleted_forms for forms_to_delete.
Expected Python:    forms_to_delete = self . deleted_forms
Generated Python:   forms_to_delete = self . deleted_forms
```

Figure 6: Sample translations with 300 epochs of training (green text indicates a perfect translation by the model, blue a close one with correct structure, and orange a wrong translation)

## References


[1] John Hutchins. The history of machine translation in a nutshell, 2005. http://hutchinsweb.me.uk/Nutshell-2005.pdf.

[2] Jeremy Norman. Petr Petrovich Troyanskii of St. Petersburg Invents a Mechanical “Translating Machine”, 2020. http://www.historyofinformation.com/detail.php?id=3887.

[3] Abdulaziz Alhefdhi, Hoa Khanh Dam, Hideaki Hata, and Aditya Ghose. Generating Pseudo-Code from Source Code Using Deep Learning. 2018 25th Australasian Software Engineering Conference (ASWEC), 10.1109/ASWEC.2018.00011, 2018.

[4] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Naka. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation. 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 10.1109/ASE.2015.36, 2015.

[5] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. SPoC: Search-based Pseudocode to Code, arXiv:1906.04908 [cs.LG], 2019.

[6] K.M. Tahsin Hassan Rahit, Rashidul Hasan Nabil, and Md Hasibul Huq. Machine Translation from Natural Language to Code using Long-Short Term Memory, arXiv:1910.11471v1, 2019.

[7] Mani Wadhwa. seq2seq Model in Machine Learning, 2018. https://www.geeksforgeeks.org/seq2seq-model-in-machine-learning/.

[8] Sequence to sequence example in Keras (character-level). https://keras.io/examples/lstm_seq2seq/.

[9] Harshall Lamba. Word Level English to Marathi Neural Machine Translation using Encoder-Decoder Model, 2019. https://towardsdatascience.com/word-level-english-to-marathi-neural-machine-translation-using-seq2seq-encoder-decoder-lstm-model-1a913f2dc4a7.

[10] Manish Chablajni. Sequence to sequence model: Introduction and concepts, 2017. https://towardsdatascience.com/sequence-to-sequence-model-introduction-and-concepts-44d9b41cd42d.