Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.S083, Computation and Linguistics

Final Project: Artificial Python Code Generation

Jake Barnwell (with some help from Abdi Dirie)

1 Introduction

Inspired by the success of ngram-based artificial language generators, such as the now-well-known Math [3] and Computer Science [4] paper generators, we decided that we wanted to try to generate artificial *code*. We liked the idea of generating code because its structure is much more rigid than that of natural language, and hence generating it based on its hierarchical structure should yield reasonably "correct"-looking code. We wanted to use CFGs and PCFGs to try to generate such code, even though technically programming languages are context-sensitive and not context-free.

Our choice of language was Python, since it has very little overhead, which makes context-sensitive errors like bad class-names and scoping less obvious to a human reading the generated code. However, Python is dynamically typed, making it much harder to infer the correct context for certain variables and functions. For example, if a function takes an `int` and a `string`, it is easy to enforce that statically and hierarchically in Java or another statically-typed language, but much harder to enforce when generating Python. Though this particular job would be harder with Python, we decided that we weren't necessarily looking to generate perfect Python code. Instead, we were interested in the methods used to make "reasonable-looking" code artificially.

Since the formal grammar for Python (or any programming language) is exact, and is always exactly followed (unlike a natural language), generation of code using the structure rules will always be syntactically correct.[1] From this point of view, it is far easier to generate reasonable code than reasonable natural language. Unfortunately, because variables, function names, etc. are often chosen and used on a per-developer basis, no contextual information about certain "words" (or names, or string literals, and so on) can be gathered on a large scale to help make semantically meaningful code; this is not so for natural languages, which have common sets of words used by everyone and hence is amicable to such analysis, which improves language generation and recognition.

In this paper we will explore the design of our Python-generating system, as well as some of the process behind creating the system.

2 Design

Generation of Python code happens in a series of steps. Below is a high-level description of each of those steps, which are discussed in more detail later on. Listed next to each step is the relevant section that discusses that step.

1. **Generate the Corpus (3.1)**: Generate the corpus of Python training files.
2. **Parse Data (3.2)**: Use the Python Parser module to parse files into AST representations.
3. **Infer CFG Rules (3.3)**: Walk the parsed AST trees, inferring CFG rules and other relevant data.
4. **Induce PCFG Model (3.4)**: Induce a PCFG model based on the CFG rules and their relative frequencies.

[1] Additionally, if context-sensitive issues (such as variable names, scoping, etc.) are appropriately resolved, then the resulting code should also be semantically meaningful. However, this was not our primary focus.

5. **Generate Raw Artificial Code (3.5)**: Using the PCFG model as well as other data, stochastically generate artificial Python code.

6. **Post-Process Artificial Code (3.6)**: Apply various tweaks to the generated code to make it more human-readable and believable.

3 Implementation

We discuss the implementation details of our Python-generation code, as outlined in Section 2.

3.1 Generating the Corpus

To acquire a corpus of data files, we query the SearchCode API [1] for .py files, using the query term "`import`", since the API requires at least one search term and the `import` keyword is one of the most ubiquitous statements in any Python file.

We originally wanted a corpus size of 10000 Python files. However, limitations with the API meant that we were only able to acquire 1000 Python files. However, we have found that this is sufficient to generate believable Python code (indeed, using only a few hundred Python files as training files was acceptable).

Each Python file that is gathered for the corpus is saved with its native file name so that, in the future, we can utilize that information during generation. To prevent conflicts among files with the same names, each file is put in a different folder with a unique identifier.

3.2 Parsing the Data

We used Python's in-house Parser module to do the parsing for us, instead of having to write our own custom code or use an external package like NLTK.

We took the corpus of training files and had the parser parse each one, storing the file's parse tree into an AST node (also referred to as an AST object, or just AST).

The format of an AST node is relatively intuitive. The top-level node of the AST is the root: in our case, the root node is always a "`Module`" object node which covers the entire input code. Each node (sometimes referred to as a `head` in our code and in this paper) has zero or more children, representing constituent elements or *fields* of that node.[2] The leaves of the AST are the primitive objects that have no children, such as string literals or number literals.

3.3 Inferring CFG Rules

To infer CFG rules, we execute a standard tree walk on each of the parsed ASTs. At each node, we mark that node's class name (e.g. "`Module`" or "`ClassDef`") as the *head*, mark that node's children as ordered *constituents*, and log the nominal rule based at that node as

```
head -> constituents...
```
(1)

If that same rule has already been logged, we increment a frequency counter for that rule.

With this method, we calculate the nominal frequency count for each and every re-write rule that is used in the corpus.

---

[2]The root `Module` node, for example, has the field `body` which points to an ordered list of constituent top-level entities. One such possible `Module` rule is

```
Module(body=[Import(...), ImportFrom(...), Assign(...), ClassDef(...)])
```

In addition to counting rules using the AST structures, we also store *primitives* into a dictionary for later use. We recognize and use seven primitives in parsing and generation:

$$\texttt{primitives} = \{\texttt{int}, \texttt{long}, \texttt{float}, \texttt{complex}, \texttt{string}, \texttt{unicode}, \texttt{bool}\} \tag{2}$$

In our code, we often refer to `int` and `long` as *inty* or *int-like*, and we refer to `string` and `unicode` as *stringy* or *string-like*.

Semantically, any token in a Python file that is not a reserved keyword or pre-defined control sequence is considered a primitive: this includes variable names, function names, attribute names, module names, value literals, and so on. Technically, primitives are exactly the elements whose class-names are one of the seven listed in Equation 2. In practice, primitives are the leaves of the ASTs.

3.4 Inducing the PCFG

We induce a PCFG model from the CFG rules inferred in Section 3.3.

To do so, we turn the nominal frequency counts of the rules into probabilities by normalizing the counts *by head*. Supposing that $R(\texttt{h})$ is the set of all rules with `h` as the head,

$$\forall\ \texttt{h} \in \texttt{heads}, \sum_{\texttt{rule} \in R(\texttt{h})} P[\texttt{rule}] = 1 \tag{3}$$

Hence, each unique rule ends up with an assigned probability normalized by head-type.

Similarly, the dictionary of primitive counts is normalized so that the assigned probabilities for each primitive is normalized for each primitive class-type. If $V(\texttt{p})$ represents the set of stored possible literal values of primitive class-type `p` (where $\texttt{p} \in \texttt{primitives}$ in Equation 2), then

$$\forall\ \texttt{p} \in \texttt{primitives}, \sum_{\texttt{val} \in V(\texttt{p})} P[\texttt{val}] = 1 \tag{4}$$

3.4.1 Smoothing

We do not do any smoothing on the rules frequency counts. One reason for this was that we found that it didn't improve the quality of the generated Python code by any respectable amount. Another problem was that it just needlessly complicated our code. We do admit, regardless, that it wouldn't have hurt to include it.

One possible issue with not smoothing our rules data is that discrete intermediate rules (that aren't found in training) will never be invoked. For example, since the children of the root `Module` node are exactly the top-level expressions of the file (basically, every line of code that is flush to the left margin), those children are considered the constituents in the rewrite rule. This means that there are likely hundreds of rewrite rules with `Module` as the head, each a distinct rule since there is a different number of import statements, or assignment statements, or class definition statements, or whatever. If we have a rule for 10 function definitions and a rule for 12 function definitions, but no rule for 11 function definitions, no Python file with 11 function definitions will ever be written. Smoothing to allow the possibility of un-represented rules to be invoked would help alleviate this issue technically, but it is just not something that would typically be noticed by a human reading the code.[3]

We did, however, smooth primitives frequencies data. We thought this was a very important change. Before smoothing, certain strings (coming from both string literals and what we call *name-like* entities, which

[3]Note that if we were to somehow use function currying with our generation, this entire issue may have been avoidable, since we can recursively curry some arbitrary number of times.

include function, variable, attribute, and argument names) had extremely high frequencies that absolutely dominated the others. Examples of such strings are "`self`", "`value`", "`None`", and "`len`".[4]

To prevent the generator from using these strings/names stiflingly often, we wanted to remove some weight from these high-frequency words and add more weight to the lower-weighted strings. We implemented a relatively naive smoothing algorithm, but we have empirically found that it is more than acceptable.

Smoothing was done on a per-primitive-classname basis. For each primitive class, we computed the average of the raw frequency count within that class. Values that had a higher-than-average count were decreased by 10% of the difference, and value counts that were lower-than-average (or equal to the average) were increased by 1 plus 10% of the difference.

Supposing that $\texttt{mean}(\texttt{p})$ is the average count within primitive class $\texttt{p}$, $c(\texttt{val}, \texttt{p})$ is the frequency count of $\texttt{val}$ within primitive class $\texttt{p}$, and $c'(\texttt{val}, \texttt{p})$ is the smoothed count, we have:

$$c'(\texttt{val}, \texttt{p}) = \begin{cases} c(\texttt{val}, \texttt{p}) - \frac{c(\texttt{val},\texttt{p}) - \texttt{mean}(\texttt{p})}{10} & \text{if } c(\texttt{val}, \texttt{p}) > \texttt{mean}(\texttt{p}) \\ c(\texttt{val}, \texttt{p}) + 1 + \frac{\texttt{mean}(\texttt{p}) - c(\texttt{val},\texttt{p})}{10} & \text{if } c(\texttt{val}, \texttt{p}) \leq \texttt{mean}(\texttt{p}) \end{cases} \tag{5}$$

In practice, the mean was usually under 10, so adding 1 plus 10% of the difference to low-count values was a significant increase. Typically, for these values, the 10% was floored to zero and so just 1 was added.

Executing this weight-shifting usually caused the mean of the set to increase by roughly 1, which is not surprising. Note that we did not care too much about preserving weight, since everything would be normalized later anyway.

It is possible that even stronger weight-shifting could work, but we worried that if we shifted too much weight from the common words, a human reading the generated code would start to wonder why they didn't see any of what they believed to be a popular name.

3.5 Generating Raw Code

Before generating anything, we first do some pre-processing on the primitives dictionary. After this pre-processing is complete, we can actually generate the ASTs and then the raw code.

3.5.1 Pre-Processing the Primitives Dictionary

Recall that we mentioned "name-like" entities in Section 3.4. Name-like entities are tokens in Python files that are *named*: such entities include function names, variable names, attribute names, argument names, and module names. Each of these names are, at the lowest level, string-literals *without the enclosing quotation marks*, but these string-literals must satisfy certain constraints.

For example, Python variables must begin with an underscore or letter; after that, the variable may have any number of digits, underscores, or letters. For simplicity we have extended this rule to cover all *name-like* entities.[5]

Because of these restrictions for name-like entities, we have to differentiate between primitive strings that are requested to populate a string literal and those requested for a named entity. To simplify random polling of such a primitive string, we build separate dictionaries for the two cases (one dictionary contains

[4]Recall that these primitives were loaded from basically anything that wasn't a strict keyword. `len` is included because it's a common function name. `value` is a relatively common variable or attribute name. One might consider `self` a keyword, but the Python parser likely treats it as just another variable name with lexing and parsing. Similarly, we would consider `None` as a strict special word, but again the parser just interprets it as a literal or variable name.

[5]We believe this is an exact and accurate restriction for all name-like restrictions, although we are not positive that module names (in `import`-like statements) follow the exact same rules as variable names. We think they do (ignoring dots, which are dealt with in a separate place in code) but we were not able to find sufficient documentation.

all strings; the other, only name-like-applicable strings). This is the pre-processing that is done on the primitives dictionary.[6]

3.5.2 Generation

We generate an artificial AST by starting with a `Module` node and recursively filling the entire tree by populating its constituent children. The choice of children with which to populate a node is made by choosing from the set of rules that have the current node type as the head. The rule is chosen by the probability distribution of rules in the PCFG model.

We deal with name-like entities as explained in Section 3.5.1 by carrying around a "context stack" when generating the AST. This context stack contains information about the current node's ancestors, including their class names and field types. This allows us to know what "type" of primitive string is being requested: in our case, if it is a primitive string that will be used to fill in a name-like entity, or if it will just be used to fill in a string literal token. The astute reader will notice that this makes this a non-context-free language: see Section 3.5.3 for more discussion on that.

To turn the AST into human-readable code, we utilize the `Unparser` class supplied as a demo file in the Python Subversion repository [2]. The `Unparser` class takes an AST and outputs human-readable code into a specified file.

3.5.3 Context-Sensitivity of Python

An unfortunate caveat to our code generation is that Python (and programming languages in general) is not actually a context-free language. It is somewhat context-sensitive, which may be obvious: as a simple example, consider that a variable must be defined before it can be legally read. This itself already guarantees that Python is context-sensitive.

One of the biggest downsides to our artificial Python generation is that, since it is generated in a context-free way (except for the name-like entities, which are kind of hacked in to act correctly), there is very little *cohesion* within the output code file. There are rarely two references to the same variable (or function, or attribute, etc.) since we do not do any smart analysis of "already-used-names," and names are chosen randomly. This is definitely a big (and arguably *the* big) region we can improve on.

3.6 Post-Processing the Generated Code

We do some post-processing on the generated Python file in order to improve readability and make the code seem more realistic. We currently have three post-processing filters:

- Remove lines that only contain a string literal. We have found that, for some reason, about 5% of lines generated contain just a single string literal with nothing else. This contributes exactly nothing to the code, and is something one would never see in code written by a reasonable human being.
- Remove trailing comma in certain function- and class-definitions arguments lists. Although the trailing comma is technically allowed, it's something we rarely see written by human programmers. Note that having a trailing comma is somewhat more standard in tuple literals (especially in single-element tuples, since it is required), so we do not remove the commas in those cases.
- Add random line breaks. The `Unparser` module deterministically adds line breaks given the context of each line. We feel that a human programmer would sometimes add empty lines to delimit certain parts of a block of lines. As such, if three consecutive lines have no empty lines between them, a new empty line is added with probability 0.15.

[6] Technically, we have to do similar pre-processing with int-like primitives as well, but the explanation is far more boring. See the code for details.

We feel that it would be more linguistically and intellectually "correct" to incorporate these changes into the code generation step itself (using heuristics based on things such as the context stack, as well as keeping track of heretofore-generated nodes), but we thought it would be nice to have a dedicated step here so that arbitrary post-processing functions could be added on in a simple way. As such, we have left these post-processing filters here and will likely keep at least some of them here in the future.

4 Results

Here is a snippet of code that we generated:

```
...
def auto_now(_meta, os, save=LatLonBox, indexer=[], new_food='x', **BlockTranslateNode):
    raise name(object)

@self
def Result(shutil, hill, pickled, allowed_methods, _add_middleware, null):
    if MOVED:
        layermap = duration(('index' % self))
    else:
        end = super[60:]
    if (not sorted_dirs.request):
        toEmail.toString.privatekeyfile(test_route_url_generation_error, self)
        return self.sort(cls, source_lines)
    return context(_object_cache.slug.CLONEKIND_TO_MODULE)
...
```

In all, it looks pretty good. You can see the prevalence of the `self` keyword still poking through, since the system tries to use that as a function decorator, and also as a string formatter. Nothing else looks *too* suspicious in local regions, however it should be obvious after a few seconds that this code is not globally cohesive at all, and would almost certainly not even run. Also, we can tell that none of the variables or arguments used even care about previously-defined variables, since almost every reference to a name-like entity is an entirely new one.

There are a few other small issues we occasionally have. The first is that sometimes a laughably small file is generated, like this one:

```
import args
print ('vmID' % (print_result, test))
```

Files this size are not *impossible* in the wild, but we would prefer something bigger. On average, the files generated are typically a hundred or so lines, so this is a moderately rare occurrence, but it does happen.

Here we paste the block of import-like statements from another generated Python file:

```
import None, os
import d_list
import USE_EMBEDDED

import pos as FILE_SHARE_READ
from response import worksheet
from .test_ticket14876 import library_manager, lstrip

from value import numpy
from self import smart_str, render_template, self, nodelist as setParent
...
```

These imports are acceptable, although as you can see, we often get crazy import directives such as "`import None`" or "`import self`". This is again due to such high prevalence of these common string-literals.

Lastly, we sometimes use quite fantastic number-type literals, like in this assignment we found in one of our files:

```
messages_replied(None, setup_complete).self = [337837, 3511, 10, 12634, 4368.1721949481]
```

This is in part due to the smoothing we did. We may have been a bit too aggressive with the smoothing in this particular case (that is, for numbers). However, having too little smoothing resulted in a very high prevalence of `1`'s and `0`'s, which are boring in their own way.

5 Conclusion

We feel that we have made significant progress in generating fake, human-readable Python code. There is certainly room for improvement in several directions, the two major ones being smoothing algorithms and context-sensitive generation. However, as a first approximation, we have generated passable code that might even fool a non-programmer.

References

[1] B. Boyter. searchcode — source code search engine, 2015.

[2] M. Dickinson. unparse.py, 2010.

[3] N. Eldredge. Randomly generated mathematics research papers!, 2012.

[4] J. Stribling, M. Krohn, and D. A. Eldredge. Scigen - an automatic cs paper generator, 2005.