PySmith: A Random Python Code Generator

Hugo Wood

March 20, 2012

Where we left off

The first version of PySmith, developed during the Fall quarter, had one main drawback: it borrowed the principles of CSmith and blindly applied them to Python. This process resulted in the generator being severely limited by its need to track types. While this is completly fine (and necessary) when generating code in statically-typed languages, it is crippling when producing code in dynamically-typed languages such as Python. It applies large additional constraints to the generated code, decreasing its variety, and disables a whole chunk of language features based on dynamic typing. An example of such a feature is the possibility of declaring a variable inside a branch or a loop, and have this variable reachable outside the scope of that branch or loop.

Embracing Dynamic Typing

The solution to this problem is to stop keeping track of types. However, this gives the generator too much freedom. The generated code has calls to non-existing members, uses operators with invalid types, calls functions with completly random parameter etc. Notice that this type of code has no undefined behavior and will just raise exceptions, consequently it can still be use to test interpreters. Yet, generating 200 lines code so that the first one raises an exception is not very interesting.

So how can the generator know if it can output `a+b` without knowing what are the types of `a` and `b`? Well, there is only one way. The code has to be run. If an error is thrown, then the statement is not valid. This is a native feature of Python, exposed as the `exec` statement. This statement has to be supplied with code (as a string) and two mappings from strings to objects representing the global and local variables respectively. The code is interpreted within the context of these mappings, which get updated accordingly. If the code raise an error, it bubbles up to the caller.

This solution was implemented. Every statement that gets generated is run in a context (separate from the one the generator itself is running in). Depending on a user setting, an erroneous statement can be dropped or kept. Obviously, the generation process is a lot slower, but it is also much more "Pythonic" and can reach more into the language. Also, the code is signigicantly simpler and smaller.

The problem now is that a majority of statements gets dropped. In order to solve this, a number of "guards" were implemented. For example, when generating a division, instead of picking two variables at random, the program queries the current context for two variables that expose the methods `__div__` and `__rdiv__` respectively [1], and cheks that the second one is not 0. Such "guards" prevents the generation of erroneous statements ahead of their execution, hence reducing those from nearly every statement to just a few.

Refactoring

The project went through extensive refactoring. An emphasis was put on simplicity. The model classes of language constructs are now delegated to `ast` module, part of Python's standard library. All user-definable rules are stored in the `rules` module.

New Constructs and Features

- Augmented assignments.
- Variadic functions. Both classic and keyword extra arguments are supported.
- While loops. Termination is guaranteed through an additional exit condition of the form `x < A` where A is integer constant and x is always incremented in the last statement of the loop. Moreover, the `continue` keyword is not allowed.
- Classes. Single and multiple inheritance are supported. Old-style and new-style classes are supported.
- String, tuple, list and dict literals.
- Built-ins read and write. All built-in types and functions can be used and redefined in generated programs.

Results

Unfortunately (fortunately?), no bugs were found in the 4 major Python interpreters.

[1] I have learnt since that this is not exactly the way to find two compatible operands, but this technique works quite well.