Sawyer Redstone

The PL Python Module

The Prolog programming language offers powerful logic programming capabilities applicable to a wide variety of purposes, including natural language processing, theorem proving, and other AI systems. However, despite its unique capabilities, Prolog lacks many of the capabilities of modern object oriented languages such as Python. PL[1] is a Python module designed to offer Python programmers the best of both worlds. By importing the PL module into their code with the instruction "**`from PL import *`**", Python programmers can use Prolog logic programming operations directly from Python without first learning an entirely new language. PL allows users to create Prolog style predicates, variables, and constants representing numbers and atoms, and also offers its users Prolog-like recursion, computation, and output capabilities.

Variables and Constants

In Prolog, variables and constants are distinguished by their capitalization, but in PL they are explicitly defined by the class that created them, i.e. **`Var()`** and **`Const()`**, respectively. A PL user can instantiate a variable, say, `X`, through the instruction `Var("X")`. Similarly, a user can declare constants, such as **`dog`** or the number **`2`**, through instructions like `Const("dog")` or `Const(2)`. Note that in PL, atoms and variable names are written as strings.

Predicates

To set up Prolog logic code in PL, the user begins by defining all the predicates they will be using. This is done by setting a Python variable equal to **`Predicate(name)`**. For example, to create a `parent/2` predicate, a user would add the statement `parent = Predicate("parent")` to their code. Note that this predicate shows no indication of its arity. That is because this same statement can be used for all predicates that share a name, regardless of arity.

Let's walk through some code in Prolog and show how the same can be implemented in PL.

Adding to the Knowledge Base[2]

A user begins by filling the program's knowledge base with facts. For example, suppose they wish to provide information on the gender and parent/child relationships of a group of individuals. In Prolog, this code may look as follows:

```
male(sawyer).
male(michael).
female(rachel).
```

---

[1] PL was originally called PythoLogical, but focus groups found the name "disturbing".

[2] The code examples in this guide can be found in PL_Example.py and Prolog_Example.pl in the project's Github repository.

```
female(naomi).
child(sawyer, michael).
child(sawyer, rachel).
child(rachel, naomi).
```

To create a similar set of facts In PL, the user begins by defining the predicates that will be used:

```
male = Predicate("male")
female = Predicate("female")
child = Predicate("child")
```

The user then adds facts to these predicates through use of PL's **`add()`** method:

```
male.add([Const("Sawyer")])
male.add([Const("Michael")])
female.add([Const("Rachel")])
female.add([Const("Naomi")])
child.add([Const("Sawyer"), Const("Michael")])
child.add([Const("Sawyer"), Const("Rachel")])
child.add([Const("Rachel"), Const("Naomi")])
```

Of course, a logic program with only facts isn't very interesting, and the user will likely desire to add some rules to the program, such as defining what it means to be a parent. In Prolog, this can be done with an instruction like the following:

```
parent(A, B) :- child(B, A).
```

To achieve a similar effect in PL, a user may begin by adding `parent = Predicate("parent")` to their list of predicates, then may add a rule as follows:

```
parent.add([Var("A"), Var("B")], [Goal(child, Var("B"), Var("A"))])
```

This syntax is somewhat unwieldy, so in order to simplify rules such as this, PL users may choose to pre-declare variables such as `A = Var("A")`, then use the variable name as a shorthand for all future variables with that name. For example, the rule above can be written more clearly with the following set of instructions:

```
A = Var("A")
B = Var("B")
parent.add([A, B], [Goal(child, B, A)])
```

There, isn't that easier to read?

In this case, the `add()` method takes two arguments: a list of the terms found in the head of the clause and a list of the goals found in the body. For a slightly more complex case, the user may wish to add the rule that a mother is someone who is both female and a parent. Here is how a rule like that might look in Prolog:

```
mother(A, B) :- female(A), parent(A, B).
```

To implement this in PL, the user first adds `mother = Predicate("mother")` to the list of predicates, then adds this line to their code:

```
mother.add([A, B], [Goal(female, A), Goal(parent, A, B)])
```

Notice how there was no need to re-declare variables `A` and `B`, as they were already defined earlier in the code.

PL can also be used for recursive rules. Suppose the user wishes to write a predicate that succeeds when someone is an ancestor of someone else. This predicate might be written In Prolog as follows:

```
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :- parent(A, C), ancestor(C, B).
```

Implementing a recursive predicate in PL is no different from adding any other predicate. The user begins by adding `ancestor = Predicate("ancestor")`. Since the rule will use the variable `C`, the user must next add `C = Var("C")` to the list of variables. Once that is done, all that is left to add is the following:

```
ancestor.add([A, B], [Goal(parent, A, B)])
ancestor.add([A, B], [Goal(parent, A, C), Goal(ancestor, C, B)])
```

PL can also evaluate arithmetic expressions. Prolog has a special built-in predicate for this called `is/2`. The Prolog expression `X is 2 + 3` evaluates `2 + 3` and binds the resulting value to `X`. A user can combine this with recursion to make a predicate count/2 that can infinitely increment its first argument, as shown in the Prolog code below:

```
count(A, A).
count(A, C) :- B is A + 1, count(B, C).
```

A user of PL can use the built-in predicate `equals` to produce the same effect. Unlike its Prolog equivalent `is/2`, `equals` in PL works no differently than any other PL predicate. First the user must add the line `count = Predicate("count")` to their code. Once they have done this, they may add this to their code:

```
count.add([A, A])
count.add([A, C], [Goal(equals, B, A + Const(1)), Goal(count, B,
C)])
```

Another built in predicate available in both Prolog and PL is the `write/1` predicate. Say a user would like to make a calculation and then write its results to the screen. They may try it in Prolog like so:

```
write_var(A) :- A is 6 + 2, write(A).
```

The PL equivalent is creating the predicate `write_var = Predicate("write_var")` and then adding this line of code:

```
write_var.add([A], [Goal(equals, A, Const(6)+Const(2)),
Goal(write, A)])
```

Unfortunately, in both scenarios, the user would find this has the undesired effect of not only outputting 8 to the screen, but also unnecessarily displaying the variable that has been assigned to the value 8 beneath it. To avoid this, PL provides Prolog's `fail` feature, which ensures that a rule will fail and prevents instantiated variables from displaying. An example of this in Prolog would be the following:

```
write_var2(A) :- A is 6 + 2, write(A), fail.
```

A similar effect can be achieved in PL with this:

```
write_var2.add([A], [Goal(equals, A, Const(6)+Const(2)),
Goal(write, A), fail])
```

Lastly, to create a predicate with an arity of 0, the `add` method may take an empty list as its first parameter. Take this line of Prolog code:

```
always_true.
```

The same can be written in PL by setting `always_true = Predicate("always_true")` and then adding this line of code:

```
always_true.add([])
```

Issuing Queries

Predicates are not very useful if they cannot be queried. Once the user has added facts and/or rules to their knowledge base, they must then be able to search through solutions to a query. In Prolog, a user is not limited to finding one solution to their query, and can instead continuously search for as long as the query is successful. In order to emulate this feature in Python, PL uses

a generator function. This generator, **`tryGoal()`**, takes in the goal that is being queried as an argument.

To demonstrate how this may be used, we will use the knowledge base built earlier. To find all people that are male, a Prolog user would make the query `?- male(X).` Prolog would respond with the result `X = sawyer,` at which point the user may choose to press the semicolon to produce a second result, `X = michael`.

To make this query in PL, a user can begin by adding `X = Var("X")` to the list of defined variables. Next, the user may add this line to the code.

```
success = tryGoal(Goal(male, X))
```

This makes `success` into a generator, which can be used in the following ways:

Searching Through All Solutions

The following code will output each result on a new line until no more results can be found, at which point `False` will be displayed.

```
for s in success:
    print(s)
```

Here are the results for this query:

```
['X = Sawyer']
['X = Michael']
False
```

Searching Select Number of Solutions

For situations when the user only wishes to see a limited number of solutions, for example 5, they may instead use the following code:

```
for _ in range(5):
     print(next(success))
```

The number 5 can be replaced by the number of solutions the user wishes to display. However, the current version of the program will issue a StopIteration error if an instruction is searching for more solutions than exist.

Finding a Single Solution

Lastly, if the user is looking for a single solution, they may use `print(next(success))`.

Next Steps

The features described in the document provide glimpses of some ways in which PL can offer Prolog functionality directly from within Python, but there are still many areas for improvement. To name just a few, future enhancements could include the addition of Prolog's cut (!) functionality and list processing capabilities, and the addition of better parsing capabilities so that users could add commands more succinctly.