Reintroducing PyLogical

Seth James Nielson
Harbor Labs Research and Development
seth@harborlabs.com

3/26/2012

Abstract

Python is a modern scripting language that has embraced a largely object-oriented framework, but has also supported a number of functional programming constructs. In previous work, we introduced extensions to increase the functional programming capabilities of the language and we also introduced a novel purely-python module that implemented a logic programming style pseudo-syntax. That module was purely academic and was significantly limited in scope and expressiveness. In this paper, we present the newly updated PyLogical module by first reviewing the philosophy behind the mixing of the two paradigms, give a brief overview of the updated pseudo-syntax, and compare this syntax with the Prolog. We note that our new module is capable of expressing almost all Prolog language features including DCG’s with minimal syntactic overhead.

1 Introduction

In an earlier work [10, 9] we explored some possible multiparadigm extensions to well-known object-oriented design patterns. For practical experimentation, we chose to use the Python programming language as it already supported some functional programming concepts and also because Python can be heavily modified at run-time. For our experimentation, we expanded a number of new functional constructs as well as a from-scratch extension for logic programming. Nevertheless, we only implemented a minimal set of functionality as a proof-of-concept and to test our experiments.

We recently revisited the logic programming extension that we called PyLogical to improve the existing features as well as to significantly expand its capabilities. As in our first work, our goal was to create extensions with the following characteristics:

1. A native Python module requiring no recompilation of the Python interpreter
2. A pseudo syntax\(^1\) that mimics the well-known Prolog language
3. All of the functionality of the Prolog language

We have achieved these goals in our newly updated PyLogical experimental release 1 (X1). In this technical report, we first note similar projects and related work in 2 and then discuss the philosophy behind our work in section 3. Next we present many of the basic operations and features of the module in section 4. Then we compare our module to Prolog in terms of functional capabilities, lexical similarities, and limitations in section 5. Finally, we conclude in section 6 by briefly comparing our work to similar projects and discuss future research goals.

## 2 Related Work

There are a number of other python-logical modules that we have identified in the course of our research. We briefly review these projects below.

1. PyLog. The PyLog [4] module is similar in many ways to PyLogical. It was developed some time after our initial disclosure in [9]. However, it does not have the capability to create logical rules of inferences using native Python. Instead, it translates prolog chunks into components that can be used in Python.

2. Prolog in Python. This project [3] translates Prolog into Python, but does not provide a native interface. It is also unclear how DCG is translated.

3. Pythologic. This reference appears to have been disclosed initially in [2]. However, it is unclear how this version would do actual searching. In a later reference [1], it used PyLog to compile its queries. This system also does not support DCG syntax. Furthermore, it has limitations in namespace resolution because of its automated identification of logic variables.

One major difference between these systems and ours is these systems are largely multiple paradigm while ours supports truly multiparadigm programming. Multiple paradigm approaches allow separate paradigms to live side by side interacting through some limited API. Multiparadigm, however, allows the individual paradigms to interact so closely that the result is a hybrid expression with the potential for entirely new linguistic constructs.

In this paper we will not be discussing multiparadigm combinations. Instead, we will focus on the purely logic-programming operations of PyLogical. We feel that it is essential to illustrate that PyLogical is capable of expressing programs using those operations almost exclusively. Furthermore, we believe that increased understanding of a particular paradigm will increase one’s ability to conceive of novel multiparadigm solutions. Nevertheless, what sets PyLogical apart is that it has been designed for future multiparadigm research.

\(^1\)We use the term pseudo-syntax to indicate that, of course, all our constructs are using Python’s unmodified syntax. Nevertheless, we repurpose python to provide an appearance of a different syntax.
3 The Philosophy of Pylogical

Software languages, the tools that convert or interpret the human instructions into something the computer understands, serve two critical purposes. First, they provide conceptual structure to software authors so that their instructions can be organized, expressed, and even explored within a manageable framework. Second, they allow other developers (and even the original authors at a later point in time) to understand what the software does and facilitate maintenance thereon.

In fact, the software language is fundamentally about the developer, not about the computer. “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” [6]. Software languages provide the developer with a framework for conceptual abstractions. These abstractions allow, among other things, the software code to be broken into understandable components with extensive syntactic and semantic clues as to how the component works and what it is supposed to do. These languages generally fit one or more programming language paradigms that also provide abstractions and encapsulations of software ideas.

But while the software language is a tool for the developer, we also believe that it can significantly shape how the developer views and approaches the problems that he or she is solving and the solutions that he or she is creating. When one authors software, the paradigm or paradigms known can literally create the reality of the programming experience as a paradigm constitutes a way of viewing reality [5].

Within linguistic circles, the concept that language influences thought is called linguistic relativity and is often called the Sapir-Whorf hypothesis [8]. How exactly this influence operates and to what degree is the subject of significant debate [7]. Yet, nearly everyone that has attained fluency in a second language has experienced a moment when they realized that the two languages cannot express all things equally. In fact, even when speaking in their native tongue, a polygot may struggle to express some ideas because he or she keeps reaching for a word or expression in the secondary language.

Whether or not language really alters how we think, it permanently and fundamentally alters how we express ourselves to others.

This observation is the driving force behind the development of PyLogical. PyLogical takes the logic programming paradigm of software expression and merges it into the existing and popular object-oriented language of Python. The result is a software linguistic playground where new patterns of thought and expression can be explored and tested. Our core hope is that new ways of expressing software solutions will emerge from the composite OO-Logic hybrid.

It should be noted that Python was not chosen at random. This language is popular partially because of its malleability, its practicality, and expressiveness. And while object orientation is its primary paradigm, it supports many constructs from the functional paradigm as well. Finally, Python, as an interpreted language, allows many introspective features that enable us to modify behavior of existing constructs when necessary.

In choosing one design over another for our syntax and operations, we followed these three guidelines.

1. Support all features of Prolog. Prolog is a logic programming language with widespread use and rich features. We decided to use its language features as our metric for completeness.

2. Remain Pythonic. This is a difficult goal and often contradictory to
supporting Prolog features. Nevertheless, we attempted to maintain Python customs especially when the logic programming syntax intersects the traditional Python elements.

3. Provide seamless integration. Because our goal is actually the hybrid language, and not just the logic programming syntax by itself, integration is an essential design directive.

While we have attempted to craft our PyLogical module to meet these goals. As this technical report will illustrate, we have succeeded in supporting all the features of Prolog with a few minor exceptions. The last two goals are difficult to quantify. While they drove our thinking throughout the process, we will not discuss them further in this paper choosing, instead, to explore them in greater detail in future research.

4 PyLogical: Syntax and Operations

In this section, we briefly introduce the PyLogical module and demonstrate some basic syntax and operations. PyLogical is structured to behave and operate like Prolog. In particular, the basic operation is to define facts or axioms that connect data together in relationships. Subsequently, the relationships can be expanded by rules to define inferred relationships built from the base facts.

Family relationships are the canonical example for illustrating facts and rules in Prolog, and we will use that example here. At a purely conceptual level, suppose that we establish some father-of relationships axiomatically. If we can accept these relationships as true, then we can define a grandfather-of relationship rule that states, “X is the grandfather of Y if X is the father of A and A is the father of Y.”

We will illustrate father-of facts and a grandfather-of rule in PyLogical. Any relationship, whether it uses facts, rules or a combination of both, subclasses PyLogical.Relation. These subclasses will be scanned as they are created for facts and rules. In the code below, for example, we create a Father relationship and declare several facts that assert that “James” is the father of “Seth” and that “Seth” is the father of “Alex.”

```python
class Father(Relation):
    factset = facts(('James', 'Seth'), ('Seth', 'Alex'))
```

The variable name that is assigned the facts is generally unimportant. There are advanced debugging uses for these variables beyond the scope of this document. In normal usage, a Relation subclass needs only one set of facts.

4.1 Unification

Before introducing rules, we briefly diverge here to show how these Relation subclasses are searched. Logic programming syntax is more or less about searching for matching solutions. For PyLogical we provide this feature through the special unify method that returns an iterator. This method requires some special attention.

The unify method takes as input either constraints or free PyLogical variables. A “free” variable is one that is not bound to a value. The method is called unify because it can bind, or unify, the free variables to values that satisfy other constraints. For example, if we were to call Father.unify
with two free variables, those two variables could be bound either to James, Seth or to Seth, Alex. Accordingly, the unify method returns an iterator that searches through all solutions.

However, some values can be constrained. One could, for example, call unify with the string "James" and a free variable. Now, the unify method has to search for data pairs where the first element is "James". The code below shows both of these examples.

```
print "Search father for any matches."
X, Y = makeVars(2)  # makes two logical variables
for result in Father.unify(X, Y):
    print X.value(), Y.value()

Z = Var()
print "Search father for children of James"
for result in Father.unify("James", Z):
    print Z.value()
```

This is the output of the code:

Search father for any matches.
James Seth
Seth Alex

Search father for children of James
Seth

However, for simplicity, it is generally not needed to use free variables for search at this level. In the example below, we show four uses of the Iterator on the Father relation and one use of the iterator in an if statement without using logic variables. The ANY variable is used to indicate a free-variable that is not actually bound to a value.

```
# the X, Y will be the actual values matched
# and not logic variables
print "Unify 1"
for X, Y in Father.unify():
    print X, Y

print "Unify 2"
for X, Y in Father.unify("James", ANY):
    print X, Y

print "Unify 3"
for X, Y in Father.unify("ANY", "Alex"):
    print X, Y

print "Unify 4"
for X, Y in Father.unify("James", "Seth"):
    print X, Y

print "Unify 5"
if Father.unify("James", "Seth"):
    print "James is the father of Seth"
```

This is the output of the code:
4.2 Defining PyLogical Rules

We previously asserted that with father-of relationships accepted axiomatically, we could define a grandfather-of relationship by a rule. We demonstrate how to declare such a rule-based relationship in the code sample below.

class Grandfather(Relation):
    @rule
    def r1(self, Grandfather, Grandson):
        return (Father.unify(Grandfather, free.Son),
                Father.unify(free.Son, Grandson))

    As is hopefully obvious, the rule for this relationship is defined in the method r1. There are three logical variables: Grandfather, Grandson, and free.Son. Conceptually, this rule states that “Grandfather is the grandfather of Grandson if Grandfather is the father of free.Son and free.Son is the father of Grandson.”

    From another point of view, a rule is satisfied when all of the dependent relationships are satisfied. This satisfying of dependent relationships is performed in order. That is, the first Father relationship is searched for a match and then the second. If the first is unified, then the variable Free.Son will be bound to a matching value. Subsequently, the second relation is searched with the newly bound Free.Son for a matching Grandson. If there are multiple solutions, each will be matched. Once all solutions have been exhausted, the system “backtracks” and frees the variable Free.Son and a new solution is searched on the first Father relationship.

    Suppose that Grandfather.unify is called with two unbound variables. The produced iterator generates values based on r1 as shown below. Both of the Father relationships on the two lines are identical, but for clarity, they will be called Father-1 and Father-2.

    1. All three variables start out unbound.
    2. Father-1 unifies Grandfather with “James” and free.Son with “Seth.”
    3. Father-2, with free.Son bound to “Seth”, unifies Grandson with “Alex.”
    4. r1 is satisfied with Grandfather bound to “James” and Grandson bound to “Alex.”
    5. On the next iteration Father-2 backtracks when it cannot find another match.
    6. Father-1 unifies Grandfather with “Seth” and free.Son with “Alex.”
7. Father-2, with free.Son bound to “Alex”, backtracks when it cannot find a match.
8. Father-1 backtracks when it finds no more matches.
9. The iterator stops.

4.3 Variable Unification

All values passed to unify operations are either PyLogical variables or are wrapped into a PyLogical variable. Within the definition of a rule, one can use unification on variables for setting the value of unbound logic variables or for requiring a bound variable to have a specific value. For example, consider the following code.

```python
class SomeRelation(Relation):
    @rule
def r1(self, X):
        return (X.unify(3),
                AnotherRelation.unify(X))
    
@rule
def r2(self, X):
    return (X.unify(4),
            AnotherRelation.unify(X))
```

Calling SomeRelation.unify(V1) will produce an iterator that will attempt to generate solutions based on first rule r1 and then rule r2. In r1 X.unify(3) must be satisfied while in r2 X.unify(4) must be satisfied.

The effect of unifying a variable depends on whether or not that variable is bound when the unification is called. If it is bound, the unification will always succeed and the variable is bound to the value. So, in our example above, if V1 is unbound, then r1 will succeed in unifying it (aliased as X) to 3 and will then attempt to unify X with AnotherRelation. When all possible results produced by r1 have been found, it will try r2. Again, the unbound X will be unified successfully to 4 and will then attempt to unify X with AnotherRelation.

On the other hand, if the variable is already bound, the unification will only be satisfied if the values match. If SomeRelation.unify(4) is called, the X variable will already be bound to 4 when it tries to unify with 3. Because these values are not equal, the unification will not be satisfied and no more results will be generated from the r1 rule. It will, however, satisfy the unification with 4 in r2 and proceed as before.

This fairly straight-forward description explains unification for variables and simple values. It becomes more complex with composite structures such as tuples. When unifying a variable with a tuple, for example, the unification will recur internally across the elements of the tuple including elements that are, themselves, logic variables. For example, consider X.unify("s", ("s", 0)). As before, if X is unbound, the unification will be satisfied and X will now be bound to the new value. Also as before, if X is bound to the tuple ("s", ("s", 0)) it will be satisfied as well.

But X could also be bound to a tuple containing a free variable such as ("s", X1). In this case, the unification of X will succeed and X1 will be bound to ("s", 0). This is how the unification works.

- The unification begins with X.unify("s", ("s", 0)).
• X is already bound to ("s", X1).
• The unification attempts to unify ("s", X1) with ("s", ("s", 0)).
• The first per-element unification succeeds in unifying "s" with "s".
• The second per-element unification succeeds in unifying X1 with ("s", 0).
• X1 is now bound to ("s", 0).
• The unification is satisfied.

4.4 Tuples and Lists

In addition to standard values such as strings, integers, and so forth, PyLogical also has built-in support for tuples and lists in a manner similar to Prolog. These types are special datums that are useful in PyLogical relationships. Tuples allow the Pylogical fact to represent more complex data structures. For example, a “date” structure might hold day, month, and year. So, March 1, 2012 can be represented as ("date", 2012, 3, 1). Facts can be used to construct and extract elements of such structures.

class MakeDate(Relation):
    ctor = facts( free.Year, free.Month, free.Day,
                  ("date", free.Year, free.Month, free.Day) )
class GetYear(Relation):
    getter = facts( ("date", free.Year, ANY, ANY), free.Year)

These types of relationships are designed to do only one thing and do not lend themselves well to iteration. If using a relationship for a single solution, one can use the once method or the Python in construct.

Date = Var()
(2012, 1, 1, Date) in MakeDate
# Date is now bound to ("date", 2012, 1, 1)

Year = Var()
GetYear.Once(Date, Year)
print Year.value() # prints 2012

Lists can also be used in facts.

# quickfact is a convenience method for declaring
# a relation with a single fact defining it
Make2ElmList = quickfact(free.Val1, free.Val2,
                         [free.Val1, free.Val2])

PyLogical also supports the use of the special | operator for splitting a list into a head and a tail.

Head = quickfact(free.Head, [free.Head | free.Tail])

This fact can be used to check if Head is the same as the head of the list, or it can extract Head if the variable is unbound. Another potential list operation is prepend.

Prepend = quickfact(free.Head, free.List1,
                    [free.Head | free.List1])

NewList = Var()
Prepend.once(3, [2, 3, 4], NewList)
print NewList.evaluate() # prints [3, 2, 3, 4]
It should be noted that the | syntax is limited to extracting a single element from a list. It is anticipated that a future release of PyLogical will introduce a method for expressing more than one element to be extracted similar to Prolog (e.g., [X, Y | List]).

4.5 Putting it Altogether

Many powerful logic programming operations can be constructed using the basic elements we have described. One of the most fascinating aspects of logic programming syntax is generating questions to answers. Consider the following code sample.

class Add(Relation):
    zero = facts( (0, free.Y, free.Y) ) # 0 + Y = y
    @rule
    # X + Y = Z *IF* (X-1) + Y = (Z-1)
    def r1(self, X, Y, Z):
        return ( X.unify("s", free.X1),
                Z.unify("s", free.Z1),
                Add.unify(free.X1, Y, free.Z1))

This Add relation is capable of adding tuples representing the successor operation. In the following example, 0 represents the numeric 0 and ("s", 0) represents the successor to 0, or 1. The examples shows Add both finding solutions and generating questions.

# find the answer to 2+3
for x, y, sum in Add.unify(Var(),
                             Var(),
                             ("s",("s",0))):
    print x, "plus", y, "is", sum

# find all the positive X+Y’s that equal 3
for x, y, sum in Add.unify(Var(),
                           Var(),
                           ("s",("s",0))):
    print x, "plus", y, "is", sum

This is the output of the code:

("s",("s",0)) plus ("s",("s",0)) is \
("s",("s",("s",0))))

0 plus ("s",("s",0)) is ("s",("s",("s",0)))
("s",0) plus ("s",("s",0)) is ("s",("s",("s",0)))
("s",("s",0)) plus ("s",0) is ("s",("s",("s",0)))
("s",("s",0)) plus 0 is ("s",("s",("s",0)))

4.6 Definite Clause Grammars

The final PyLogical feature we present in this paper is Definite Clause Grammars (DCG’s). DCG’s provide an easy and natural way to express grammatical parsing relationships. Consider this grammar [11].

S -> NP VP
NP -> Det N
VP -> TV NP
VP -> V
Det -> the
Det -> a
Det -> every
N -> man
N -> woman
N -> park
TV -> loves
TV -> likes
V -> walks

The following PyLogical code sample illustrates the special DCG declaration syntax using this grammar as an example.

```python
grammar1 = Grammar(
    s = lambda: (np, vp),
    np = lambda: (det, n),
    vp = (lambda: (tv, np), lambda: v),
    det = ["the", "a", "every"],
    n = ["man", "woman", "park"],
    tv = ["loves", "likes"],
    v = ["walks"])

("a man loves a woman".split(" "), []) in grammar1.s
```

5 Prolog

As stated previously, PyLogical's design was inspired from the Prolog language and a brief comparison with Prolog is warranted. The following Prolog samples correspond to PyLogical examples in this paper.

Our first example Prolog code creates the `father-of` and `grandfather-of` relationships.

```prolog
father(james, seth).
father(seth, alex).

grandfather(Grandfather, Grandson) :-
    father(Grandfather, Son),
    father(Son, Grandson).
```

Hopefully the very tight coupling between the Prolog syntax and the PyLogical pseudo-syntax is apparent. It should be noted that the PyLogical code is very close in size to its inspiration. Perhaps more importantly, the layout and structure of the code are also very similar.

For our second example, we show the code for the `Add` relationship.

```prolog
add(0, Y, Y).
add(s(X), Y, s(Z)) :- add(X, Y, Z)
```

This is an example where the PyLogical code is not as compact. Note that the Prolog code has `add(s(X), Y, s(Z))` on the left hand side of the `:-`. It may be tempting for the imperative programmer to incorrectly view `s(X)`, and `s(Z)` as parameters when in fact they are values.

In Prolog, one can use values as well as variables on the left-hand side of a rule definition. Prolog will attempt to `unify` these values to whatever input is
being checked. The behavior of this unification is equivalent to the variable
unification we provide in PyLogical. If the incoming variable is unbound,
the unification succeeds and the variable is bound to the value. Otherwise,
the value of the bound variable is unified with the value. If any unification of
the incoming variables fails, the right-hand side of the rule will never even
be evaluated.

Of course, in Python, we have no convenient way to express this. Our
parameters to a rule are always variables and we cannot evaluate them until
we are inside the body of that rule. We achieve the same expressive power
by providing a unify method on PyLogical variables but the code is not as
compact.

Also, in our Add example, we used \"s\", X) and Prolog just uses
\( s(X) \). Prolog would call \( s \) an atom. Because creating atoms was non-trivial
and because they provide no extra functionality, we chose to not support them
and simply replace them with tuples. This is another example of PyLogical
having similar expressive power but in a less compact form.

Finally, here is DCG syntax in Prolog:

\[
\begin{align*}
  s &\rightarrow np, vp. \\
  np &\rightarrow det, n. \\
  vp &\rightarrow tv, np. \\
  vp &\rightarrow v. \\
  det &\rightarrow [the]. \\
  det &\rightarrow [a]. \\
  det &\rightarrow [every]. \\
  n &\rightarrow [man]. \\
  n &\rightarrow [woman]. \\
  n &\rightarrow [park]. \\
  tv &\rightarrow [loves]. \\
  tv &\rightarrow [likes]. \\
  v &\rightarrow [walks].
\end{align*}
\]

Again, the PyLogical code compares favorably in code size and expres-
siveness. As demonstrated, our module can support all major Prolog-like
functionality. To date, the only feature that we do not have an equivalent op-
eration for is extracting multiple elements from the head of a list (e.g., \([X, Y |
Tail]\)).

6 Summary

In this paper, we have re-introduced PyLogical, our extended logic program-
m ing for Python module.

Our work is both experimental and unfinished. A major open question is
the readability of our syntax to general programmers. In this paper we only
showed one particular syntax for any given operation, but in many cases there
are several version of the syntax. Which version is the most expressive and
the most clear is a very difficult question. We plan to investigate this in the
future using empirical approaches.

Even more significant is the question of usefulness to Python in general.
We would like to know if certain problems can be solved more succinctly
or more readably with a combined OO-Logic approach than either system
individually. We are currently experimenting with a make-like build system
as a testbed for exploring multiparadigm constructs.

In conclusion, our module for prolog-like logic programming in Python
provides advantages we have not found in any other similar systems. With
a few minor exceptions, it can represent all Prolog programming constructs with a small amount of additional syntactic overhead. In the future, we plan to investigate questions such as how easily our syntax can be understood by other programmers as well as best uses for the combined OO-Logic syntax.

For readers interested in using PyLogical, it is available from BitBucket at https://bitbucket.org/sethjn/MP-Python. This is an experimental project and has no installer. The pylogical directory must be in your python path or the python path must be modified. Once on the path, set up your environment for the code samples with the following code:

```python
from pylogical import
```

References


