Pylog : Prolog in Python

No Author Given

No Institute Given

Abstract. Pylog inhabits three programming contexts.

a) Pylog explores the integration of two distinct programming language paradigms: (i) the modern, general purpose programming paradigm, which includes procedural, object-oriented, and functional programming, and (ii) logic programming, including logic variables (and unification) and depth-first, search. Pylog illustrates how logic programming features can be implemented in and integrated into Python.
b) Pylog demonstrates the breadth and broad applicability of Python. Python is both the most widely used language for teaching introductory programming, and it has become widely used for sophisticated programming challenges.
c) Pylog exemplifies programming at its best, using Python features in innovative yet clear ways. The overall result is software worth studying.

1 Introduction

Prolog, a programming language derived from logic, was developed in the early 1970s. It became very popular during the 1980s as an AI language, especially as part of the Japanese 5th generation project.

Prolog went out of favor because it was difficult to trace the execution of Prolog programs—which made debugging very challenging. But Prolog didn't die and has been making something of a comeback.

- SWI Prolog (free), GNU Prolog (free), and Sictus Prolog (commercial) have kept the Prolog flame burning and have large and active communities of Prolog users.
- Sagar [15] and Raturi [13] both recommend Prolog for AI programming.

Prolog is both one of the syntactically simplest–you can learn it very quickly–and at the same time most interesting of all programming language. It is strongly declarative. One first declares facts and rules. (The rules are the Prolog programs.) One then constructs what are called queries, typically with embedded variables, and asks the system to find values for those variables so that the query satisfies the facts and rules.

Two of Prolog's most distinctive features are logic variables (which support unification) and built-in backtracking search. Prolog was one of the first programming languages with immutable variables. All in all Prolog is seductively elegant and powerful.

Python, of course, is a very well-known and widely used language. It is in first place in the two lists of AI languages mentioned above. Python is one of the easiest programming languages to learn and is used in more introductory programming courses

than any other. In addition, Python includes many powerful computational and meta-level capabilities, which facilitate the development of quite sophisticated programs.

Python supports the procedural, object-oriented, and functional programming paradigms. It does not support Prolog's logic programming paradigm. An objective of this work is to show how logic programming can be integrated into standard Python programming.

2 Related work

Quite a bit of work has been done in implementing Prolog features in Python, much of it fairly recently. As far as we can tell, none of it is as complete and as fully thought out as Pylog. But nearly all of it makes important contributions. Although, we cannot review the work in detail, following are brief (paraphrased) descriptions from the authors.

Berger (2004) [2]. Pythologic
: Python's meta-programming features are used to enable the writing of functions that include Prolog-like features.

Bolz (2007) [3] A Prolog Interpreter in Python.
: A proof-of-concept implementation of a Prolog interpreter in RPython, a restricted subset of the Python language intended for system programming. Performance compares reasonably well with other embedded Prologs.

Delford (2009) [5] PyLog.
: A proof-of-concept implementation of a Prolog interpreter in RPython.

Frederiksen (2011) [6] Pike
: A form of Logic Programming that integrates with Python.

Meyers (2015) [9] Prolog in Python.
: A hobby project developed over a number of years.

Maxime (2016) [8] Prology: Logic programming for Python3.
: A minimal library that brings Logic Programming to Python.

Piumarta (2017) [12] Notes and slides from a course on programming paradigms.

Thompson (2017) [17] Yield Prolog.
: Enables the embedding of Prolog-style predicates directly in Python.

Santini (2018) [16] The pattern matching for python you always dreamed of.
: Pampy is small, reasonably fast, and often makes code more readable.

Cesar (2019) [4] Prol: a minimal Prolog interpreter in a few lines of Python.

Kopec (2019) [7] Constraint-Satisfaction Problems in Python.
: Chapter 3 of Kopec (2019) *Classic Computer Science Problems in Python*.

Miljkovic (2019)[10]
: A simple Prolog Interpreter written in a few lines of Python 3.

Niemeyer and Celles (2019) [11] A python-constraint library.
: Pure Python solvers for Constraint Satisfaction Problems.

Rocklin (2019) [14] kanren: Logic Programming in Python.
: Enables the expression of relations and the search for values that satisfy them. (Rocklin is the author of the widely used Python toolz library.)

As this short survey illustrates, most of the ideas that provide the foundation for embedding Prolog-like capabilities in Python have been known for a while. What Pylog offers is a more fully developed, more fully explained, and more integrated version.

3 From Python to Prolog and back

This section offers a reasonably detained overview of Pylog and how it relates to Prolog. We discuss five small programs, which transition gradually from straight Python to straight Prolog. Listings of these programs are gathered together in section 3.2.

3.1 Five programs that show the way

Our strategy is to show how a standard Python program can be transformed, step-by-step, into a structurally similar Prolog program. As an example problem, we use the computation of a transversal.

Given a sequence of sets (in our case lists), a transversal is a non-repeating sequence of elements with the property that the n^{th} element of the traversal belongs to the n^{th} list/set in the sequence. For example, the sets (actually lists[1]) [[1, 2, 3], [2, 4], [1]] have three transversals: [2, 4, 1], [3, 2, 1], and [3, 4, 1]. We use the transversal problem because it lends itself to depth-first search, the default Prolog control structure.[2]

We will discuss five functions for finding transversals. In the course of discussing these programs we will introduce various Pylog features. Here is a road-map for the programs to be discussed and the Pylog features they illustrate. The first four are written in Python. The final one is in standard Prolog.

1. `transversal_dfs_first` is a standard Python program that performs a depth-first search. It stops and returns the first transversal it finds. It contains no Pylog features, but it defines a structure the other programs follow.
2. `transversal_dfs_all`. In contrast to `transversal_dfs_first`, the program `transversal_dfs_all` finds and returns *all* transversals. The most common strategy, and the one `transversal_dfs_all` uses, is to gather all transversals into a package, which is returned at the end.
3. `transversal_dfs_yield` solves the problem of finding all transversals, but it returns them one at a time. `transversal_dfs_all` does this through the use of the Python generator structure, i.e., the **`yield`** statement. This moves us an important step toward a Prolog-like control structure.
4. `transversal_dfs_yield_lv` introduces logic variables, one of the most important features of Prolog.

[1] From here on, we refer informally to the lists in our example as *sets*.

[2] We use traditional, i.e., naive, depth-first search. Most modern Prologs include a constraint processing package such as CLP(FD)[18], which makes search much more efficient.

Instead of scanning the sets in the order given, one can select the next set to scan based on how constrained the sets are. In the case of [[1, 2, 3], [2, 4], [1]], the third set would be scanned first, with 1 selected as its representative in any transversal.

Another efficiency measure involves propagating constraints. Suppose our example sets are scanned from left to right. If 1 is selected from the first set, that choice would be propagated forward, eliminating 1 from the final set. Since the final set would then have no choices left, one can conclude that selecting 1 from the first set does not lead to a solution.

Application of such constraint rules eliminate much of the backtracking inherent in naive depth-first search. Powerful as they are,we do not consider such constraint techniques.

5. *transversal_prolog* is a straight Prolog program. It is operationally identical to *transversal_dfs_yield_lv*, but syntactically very different.

Now we will look at these programs in a bit more detail. The first three Python programs have similar signatures.

```
def transversal_python_1_2_3(sets: List[List[int]],
                             partial_transversal: List[int])
                            -> <some return type>:
```

(The return types differ from one program to an other.)

Both the fourth Python program and the Prolog program have a third parameter. Their return type, if any, is not meaningful for our purposes. Each transversal, when found, is returned through the third parameter—as one does in Prolog.

```
def transversal_python_4(sets: List[List[int]],
                         partial_transversal: List[int],
                         Complete_Transversal: Var)
```

```
    transversal_prolog(+Sets,
                       +Partial_Transversal,
                       -Complete_Transversal)
```

The signatures have the following in common.

1. The first argument lists the sets for which a transversal is desired. Initially this is the full list of sets. The programs recursively step through the list, selecting an element from each. At each recursive call, the first argument lists the remaining sets.
2. The second argument is a partial transversal consisting of elements selected from sets that have already been scanned. Initially, this argument is the empty list.
3. The third parameter and the return type.
   (a) The first two programs have no third argument. They return a single transversal, a set of transversals, or **None** through the normal **return** mechanism.
   (b) The final Python function and the Prolog predicate both have a third parameter. Neither returns a value through a normal **return** mechanism. In both, the third argument is initially an uninstantiated logic variable, which will be unified with a transversal that is being returned.

The following discusses the functions in more detail.

- *transversal_dfs_first*. The first Python program uses standard depth-first recursion to find a single transversal. As listing 1.1 shows, when we reach the end of the list of sets, we are done. At that point we return *partial_transversal*, which at that point is known to be a complete transversal. The return type is *Optional[List[int]]*, i.e., either a list of *int*s, or **None** for the case in which no transversal is found. The latter situation occurs when, after considering all elements of set (line 10), we have not found a complete transversal. The program then runs off the end of the **else** clause, returning **None**.

It may be instructive to look at listing 1.2, created by the print statement in the first line.[3] It shows the value of the parameters at the start of each execution of the function. When `sets` is the empty list (line 7), we have found a transversal.

On the other hand, when the function reaches a dead-end, it backtracks to the most recent point at which it selected a tentative transversal element. For example, the first three lines show that we have selected `[1, 2]` as the `partial_transversal` and must now select an element of `[1]`, the remaining set. Since `1` is already in the `partial_transversal`, it can't be selected to represent the final set. So we (blindly, as is the case with naive depth-first search) backtrack to the selection from the second set. We had initially selected `2`. Line 4 shows that we now select `4`. Of course that doesn't help. Having exhausted all elements of the second set, we backtrack all the way to our selection from the first set.

Line 5 of the log shows that we have now selected `2` from the first set and are about to make a selection from the second set. The log does not show that we considered selecting `2` from the second set but that since it was already in the `partial_transversal`, we moved on to selecting `4` from the second set. We are then able to select `1` from the final set.

Even though this is a simple Python depth-first search, it incorporates (what appears to be) backtracking, one of the mainstays of Prolog. What implements the backtracking? There is no backtracking. We simply have recursively nested **`for`** loops, which produce a backtracking effect.

Prolog, uses the term *choicepoint* for places in the program at which (a) multiple choices are possible and (b) one wants to try them all if necessary. Pylog implements choicepoints by means of such nested **for** loops and related mechanisms.

- `transversal_dfs_all`. The second Python program finds and returns *all* transversals. It has the same structure as `transversal_dfs_first` except that instead of returning a single transversal, each transversal is added to `all_transversals` (line 12). The entire list is returned when the program terminates.

  Note that `transversal_dfs_first` returns **`None`** if no transversal is found whereas `transversal_dfs_all` returns an empty list. Hence the return type of `transversal_dfs_all` is `List[List[int]]` rather than `Optional[List[int]]`.

- `transversal_yield`. The third Python program, although quite similar to the first, takes a significant step toward mimicking Prolog. Whereas `transversal_dfs_first` **`return`**s the first transversal it finds, `transversal_yield` **`yield`**s *all* the transversals it finds—but one at a time. Instead of looking for a single transversal on lines 12 and 13 with:

```
complete_transversal = \
  transversal_dfs_first(ss, partial_transversal + [element])
```

and then **`return`**ing those that are not **`None`**, `transversal_yield` uses (line 12) **`yield from`** to search for and **`yield`** *all* transversals—but one at a time.

```
yield from transversal_yield(ss, partial_transversal + [element])
```

[3] All the programs in this section produce a log. This is the only log included in this paper.

- `transversal_yield_lv`. The fourth Python program moves toward Prolog along a second dimension—the use of logic variables.

  One of Prolog's defining features is its logic variables. A logic variable is similar to a variable in mathematics. It may or may not have a value, and once it gets a value, its value never changes, i.e., logic variables are immutable.

  The primary operation on logic variables is known as *unification*. When a logic variable is *unified* with what is known as a *ground term*, e.g., a number, a string, etc., it acquires that term as its value. For example, if `X` is a logic variable,[4] then after `unify(3, X)`,[5] `X` has the value `3`. Like mathematical variables, logic variables may be set equal to each other—even if neither has a value. So if `X` and `Y` are two logic variables, then after `unify(X, Y)` whatever value either eventually gets will be considered the value of the other. For example, after

```
(A, B, C, D, E) = (Var(), Var(), Var(), Var(), 'def')
for _ in unify(A, B):
  for _ in unify(D, C):
    for _ in unify(A, C):
      for _ in unify(E, D):
```

  `A`, `B`, `C`, `D`, and `E` all have the value `'def'`.

  Two convenience methods[6] make it possible to write the preceding more concisely.

```
(A, B, C, D, E) = (*n_Vars(4), 'def')
for _ in unify_pairs([(A, B), (D, C), (A, C), (E, D)]):
```

  Since *unify* and *unify_pairs* are both generators, they must be called from something like a **for** loop as shown—rather than as standard function calls.

  Here are a few additional important considerations.

  - *transversal_yield_lv* has a third parameter, *Complete_Transversal*, which is declared as a *Var*, i.e., a logic variable. When *transversal_yield_lv* is called initially, *Complete_Transversal* is uninstantiated. If *Sets* is empty, we perform *unify(Partial_Transversal, Complete_Transversal)*, which gives *Complete_Transversal* the same value as *Partial_Transversal*. This is typical of how Prolog programs return values: unify an argument with the value to be returned.
  - The **`else`** clause (line 8) defines *Element* to be a *Var*. The line

```
    for _ in member(Element, S):
```

[4] A note about identifiers. The Python convention is to use only lower case letters in identifiers other than class names. The Prolog convention is that the first letter of an identifier determines whether it's a constant term or a variable. Variables begin with upper case letters.

In the first three programs we have used strictly lower case letters in identifiers. In `transversal_yield_lv`, and of course in the Prolog program to follow, we use upper case letters to begin identifiers that refer to Prolog or Prolog-like variables. Thus the `X` and `Y` in this discussion begin with upper case letters. In `transversal_yield_lv`, the identifiers `Partial_Transversal` and `Complete_Transversal` begin with upper case letters. Even though they are Python variables, they are used as Pylog logic variables.

[5] or `unify(X, 3)`, the order of the arguments is not relevant

[6] • `n_Vars` takes an integer argument and generates that many `Var` objects.
• `unify_pairs` takes a list of pairs (as tuples) and unifies the two elements of each pair.

unifies *Element* with a member of *S* each time around the loop.

- The syntax of the **for** iterator/generator loop is worth a remark. The function *member* is written to perform its own *unify* operation and then to perform a *yield*. This makes it suitable for use in a **for** iterator/generator loop as shown.

  The **for** loop itself does not produce a value in the normal way. (Note the underscore.) Instead, after *member* unifies its first argument with an element of its second, it **yield**s to indicate that the unification is complete. For example,

```
Element = Var()
for _ in member(Element, S):
  print(Element)
```

  prints the elements of *S*.

- In earlier functions (line 10) we had written

```
if element not in partial_transversal:
```

  In `transversal_yield_lv` we write (line 12)

```
for _ in fails(member)(Element, Partial_Transversal):
```

  The Pylog *fails* function does the same job as \+, i.e., negation, in Prolog. *fails* takes another function as an argument—much like a Python decorator —and returns a function that succeeds or fails when its argument function fails or succeeds. Thus, although it's not boolean,

```
for _ in fails(member)(Element, Partial_Transversal):
```

  is approximately equivalent to

```
if element not in partial_transversal:
```

  For double negation, i.e., Prolog's \+\+, Pylog offers *would_succeed* rather than a pair of nested *fails*.

```
for _ in would_succeed(member)(Element, S):
```

  succeeds if and only if

```
for _ in member(Element, S):
```

  would succeed. The only (but very important) difference is that, as in Prolog's double negation, *would_succeed* does not unify any variables.

- Consider line 13 of the listing. It uses Python's **yield from** construct.

```
  yield from <something>
```

  can be considered shorthand for

```
for X in <something>:
  yield X
```

  As before, *X* may be an underscore, in which case nothing is returned, and the **yield** statement is simply **yield** with no argument.

  Consider how **yield from** is used in this program. It performs four functions.

1. It calls the remaining program to be executed. In this case, it's a recursive call, but that need not be the case.
2. It passes to the called function values from previously executed code.
3. It also passes on an uninstantiated variables—the third argument—which the remainder of the program will (presumably) instantiate.
4. Since it functions as a **`yield`**, it returns what will be the newly instantiated argument back up the **`yield`** chain.

We can put this into a Prolog context. Consider a standard Prolog clause.

```
head(<args>) :-
    term_1(<args_1>),
    term_2(<args_2>),
    ...
    term_n(<args_n>).
```

The relationship between a clause head and its body as well as that between each term and the rest of the body is exactly a **`yield from`** relationship.

- *`transversal_prolog`*. The final program is straight Prolog. Inspection of the listing will show that *transversal_prolog* and *transversal_yield_lv* are the same program expressed in different languages.

3.2 Listings

```
def transversal_dfs_first(sets: List[List[int]],
                          partial_transversal: List[int])
                         -> Optional[List[int]]:
  print(f'sets/{sets}; '
        f'partial_transversal/{partial_transversal}')
  if not sets:
    return partial_transversal
  else:
    (s, ss) = (sets[0], sets[1:])
    for element in s:
      if element not in partial_transversal:
        complete_transversal = \
          transversal_dfs_first(ss, partial_transversal + [element])
        if complete_transversal is not None:
          return complete_transversal
```

Listing 1.1. transversal_dfs_first

```
sets/[[1, 2, 3], [2, 4], [1]]; partial_transversal/[]
sets/[[2, 4], [1]]; partial_transversal/[1]
sets/[[1]]; partial_transversal/[1, 2]
sets/[[1]]; partial_transversal/[1, 4]
sets/[[2, 4], [1]]; partial_transversal/[2]
sets/[[1]]; partial_transversal/[2, 4]
sets/[]; partial_transversal/[2, 4, 1]
                                => [2, 4, 1]
```

Listing 1.2. transversal_dfs_first trace

```
def transversal_dfs_all(sets: List[List[int]],
                        partial_transversal: List[int])
                       -> List[List[int]]:
  print(f'sets/{sets}; partial_transversal/{list(partial_transversal)}')
  if not sets:
    return [partial_transversal]
  else:
    (s, ss) = (sets[0], sets[1:])
    all_transversals = []
    for element in s:
      if element not in partial_transversal:
        all_transversals += \
          transversal_dfs_all(ss, partial_transversal + [element])
    return all_transversals
```

Listing 1.3. transversal_dfs_all

```
def transversal_yield(sets: List[List[int]],
                      partial_transversal: List[int])
                     -> Generator[List[int], None, None]:
  print(f'sets/{sets}; '
        f'partial_transversal/{partial_transversal}')
  if not sets:
    yield partial_transversal
  else:
    (s, ss) = (sets[0], sets[1:])
    for element in s:
      if element not in partial_transversal:
        yield from transversal_yield(ss, partial_transversal + [element])
```

Listing 1.4. transversal_dfs_yield

```
def transversal_yield_lv(Sets: List[PyList],
                         Partial_Transversal: PyList,
                         Complete_Transversal: Var):
  print(f'Sets/[{", ".join([str(S) for S in Sets])}]; '
        f'Partial_Transversal/{Partial_Transversal}')
  if not Sets:
    yield from unify(Partial_Transversal,Complete_Transversal)
  else:
    (S, Ss) = (Sets[0], Sets[1:])
    Element = Var( )
    for _ in member(Element, S):
      for _ in fails(member)(Element, Partial_Transversal):
        yield from
          transversal_yield_lv(Ss, Partial_Transversal + PyList([Element]),
                               Complete_Transversal)
```

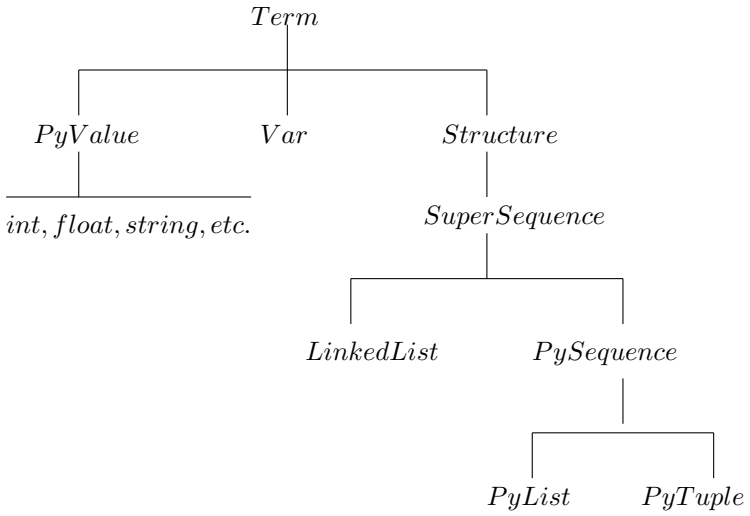
Listing 1.5. transversal_dfs_yield_lv

```
1  transversal_prolog(Sets,
2                     Partial_Transversal,
3                     _Complete_Transversal) :-
4      reverse(Partial_Transversal, Partial_Transversal),
5      writeln('Sets'/Sets;
6              'Partial_Transversal'/Partial_Transversal),
7      fail.
8
9  transversal_prolog([],
10                    Partial_Transversal,
11                    Complete_Transversal) :-
12     reverse(Partial_Transversal, Answer),
13     format('                                    '),
14     writeln('Complete_Transversal '=Complete_Transversal), nl.
15
16 transversal_prolog([S|Ss],
17                    Partial_Transversal,
18                    Complete_Transversal) :-
19     member(X, S),
20     \+ member(X, Partial_Transversal),
21     append(Partial_Transversal, [X], Partial_Transversal_X),
22     transversal_prolog(Ss,
23                        Partial_Transversal_X,
24                        Complete_Transversal_X).
```

Listing 1.6. transversal_prolog

4 Logic variables

The following diagram shows Pylog's primary logic variable classes. `Term`, the root class, is abstract. This section discusses `PyValue`, `Var`, `Structure`, and `LinkedList`.



- **PyValue**. A *PyValue* serves as the bridge between logic variables and Python values. A *PyValue* may hold any immutable Python value, such as numbers, strings, and tuples. Unlike Python, tuples are allowed only if their components are also immutable. In the example of the preceding section, the logic variable *E* with value *'def'* was actually *PyValue('def')* behind the scenes.
- **Var**. A *Var* functions as a traditional logic variable. Unification is surprisingly easy to implement. Each *Var* object includes a *next* field. That field is initially **None**. When two *Var*s are unified, the *next* field of one is set to point to the other. (It makes no difference, which points to which!) A chain of links unify all the included *Var*s.

  Consider again the example we looked at earlier.

```
(A, B, C, D, E) = (*n_Vars(4), 'def')
for _ in unify_pairs([(A, B), (D, C), (A, C), (E, D)]):
```

  After the first two unifications, we have the following.

$$\begin{array}{l} A \to B \\ D \to C \end{array} \tag{1}$$

  The next unification, *A* with *C*, points *B* to *C*. The variable *B*, rather than *A*, is pointed to *C* because *B* is at the end of *A*'s unification chain. This produces:

$$\begin{array}{c} A \to B \\ \downarrow \\ D \to C \end{array} \tag{2}$$

  Finally, unifying *E* with *D* points *C* to *E*. First, the system notices that *D* is a *Var*, and *E* is a *PyValue*. So the link must go from *D* to *E*, rather than *E* to *D*. Since *D* is already in a unification chain, it's the end of *D*'s chain, i.e., *C*, that gets the pointer to *E*. The final set of links looks like this.

$$\begin{array}{c} A \to B \\ \downarrow \\ D \to C \to E\,('def') \end{array} \tag{3}$$

  When one needs a value for any *Var*, the *Var*'s unification chain is followed to the end. If the end is a *PyValue*, its value is taken as the *Var*'s value. That's what we have in state (3). All *Var*s have the value *'def'*.

  If the end of a unification chain is an uninstantiated *Var*, as it was in state (2)—all *Var*s have *C* at the end of their unification chains—all the *Var*'s are mutually unified, but none yet has a value.
- **Structure**. The *Structure* class enables the construction of Prolog terms. The Zebra puzzle below uses *Structure* to create *house* terms.

  A *Structure* object consists of a functor along with a tuple of values. For the Zebra puzzle, the functor is *house* and the tuple of values are th house attributes: *nationality*, *cigarette*, *pet*, *drink*, and *house color*.

  One of the houses in the solution looks like this.

```
house(japanese, parliament, zebra, coffee, green)
```

The puzzle asks which house has the zebra. The answer is that it's the house in which the Japanese live.

`Structure` objects can be unified—but, as in Prolog, only if they have the same functor and the same number of tuple elements. When two `Structure` objects are unified, their corresponding tuple components are also unified.

If `N` and `P` are `Var`s, consider unifying the following two `house` objects.[7]

```
house(japanese, _, P, coffee, _)
house(N, _, zebra, coffee, _)
```

Unification would leave both `house` objects like this:

```
house(japanese, _, zebra, coffee, _)
```

This functionality is central to how Prolog solves such puzzles so easily.

(Note that unification would have failed had the two `house` objects had different values for their `drink` attribute.)

- **`Lists`**. Pylog includes two list classes. `PyList` objects mimic Python tuples. They are fixed in size; they are immutable; and their components are (recursively) required to be immutable.[8]

  More interestingly, Pylog also offers a `LinkedList` class. Its functionality is similar to Prolog lists. In particular, a `LinkedList` may have an uninstantiated tail. (That is not possible with Python lists or `PyList` objects.)

  The paradigmatic Prolog list function is `append/3`. Pylog's `append/3` has Prolog functionality for both `LinkedList`s and `PyList`s. For example, running:

```
(Xs, Ys, Zs) = (Var(), Var(), LinkedList([1, 2, 3]))
for _ in append(Xs, Ys, Zs):
  print(f'Xs = {Xs}\nYs = {Ys}\n')
```

  produces this output.[9]

```
Xs = []
Ys = [1, 2, 3]

Xs = [1]
Ys = [2, 3]

Xs = [1, 2]
Ys = [3]

Xs = [1, 2, 3]
Ys = []
```

  Pylog's `append/3` for `LinkedList`s parallels Prolog's `append/3`.

  First the Prolog version.

[7] Like Prolog, we use underscores for uninstantiated variables whose identities are not of interest. (The corresponding `Var`s are unified, but that plays no role in this example.)

[8] There is also a `PyTuple` class, which mimics the `PyList` class. The only difference is that `PyTuple`s are displayed with parentheses; `PyList`s are displayed with square brackets.

[9] The output is the same whether we use `PyList`s or `LinkedList`s.

```
append([], Ys, Ys).
append([XZ|Xs], Ys, [XZ|Zs]) :- append(Xs, Ys, Zs).
```

Now the wordier but isomorphic Pylog version.[10] (For a cleaner presentation, declarations are dropped. All parameters are: *Union[LinkedList, Var]*.)

```
def append(Xs, Ys, Zs):
  # Corresponds to: append([], Ys, Ys).
  yield from unify_pairs([(Xs, LinkedList([])), (Ys, Zs)])

  # Corresponds to: append([XZ|Xs], Ys, [XZ|Zs]) :- append(Xs, Ys, Zs).
  (XZ_Head, Xs_Tail, Zs_Tail) = n_Vars(3)
  for _ in unify_pairs([(Xs, LinkedList(XZ_Head, Xs_Tail)),
                        (Zs, LinkedList(XZ_Head, Zs_Tail))]):
    yield from append(Xs_Tail, Ys, Zs_Tail)
```

Note that **yield from** appears twice. If after the first **yield from**, *append/3* is called for another result, e.g., as a result of "backtracking," it continues on to the second **yield from**—just as in Prolog. This is standard for Python generators.

5 Zebra Problem

The Zebra Puzzle, also known as the Einstein's Riddle, is a well known logic puzzle.

There are five houses in a row. Each is occupied by a family of unique nationality. Each has a unique favorite smoke, a unique pet, a unique favorite drink, and a unique color. Fourteen clues, see below, provide additional constraints. With the given information, determine: *Who has a zebra and who drinks water?*

One can easily write Prolog programs to solve this and similar puzzles.

- We represent a house as a Prolog term with five fields:

```
house(<nationality>, <smoke>, <pet>, <drink>, <color>)
```

- Our "world" will be a list of 5 houses, with all fields initially uninstantiated.
- The clues can be written as more-or-less direct translations of the English.

```
zebra_problem(Houses) :-
    Houses = [house(_,_,_,_,_),house(_,_,_,_,_),house(_,_,_,_,_),
              house(_,_,_,_,_),house(_,_,_,_,_)],
    % 1. The English live in the red house.
    member(house(english,_,_,_,red), Houses),
    % 2. The Spanish have a dog.
    member(house(spanish,_,dog,_,_), Houses),
    % 3. They drink coffee in the green house.
    member(house(_,_,_,coffee,green), Houses),
    % 4. The Ukrainians drink tea.
    member(house(ukranians,_,_,tea,_), Houses),
```

[10] To read the code, you should know that *LinkedList*s may be created in two ways.

- Pass the *LinkedList* class the desired head and tail, e.g., *Xs = LinkedList(Xs_Head, Xs_Tail)*.
- Pass the *LinkedList* class a Python list. In particular, an empty *LinkedList* is created through the use of an empty Python list: *LinkedList([])*

```
    % 5. The green house is immediately to the right of the white house.
    nextto(house(_,_,_,_,white), house(_,_,_,_,green), Houses),
    % 6. The Old Gold smokers have snails.
    member(house(_,old_gold,snails,_,_), Houses),
    % 7. They smoke Kool in the yellow house.
    member(house(_,kool,_,_,yellow), Houses),
    % 8. They drink milk in the middle house.
    Houses = [_, _, house(_,_,_,milk,_), _, _],
    % 9. The Norwegians live in the first house on the left.
    Houses = [house(norwegians,_,_,_,_) | _],
    % 10. The Chesterfield smokers live next to the fox.
    next_to(house(_,chesterfield,_,_,_), house(_,_,fox,_,_), Houses),
    % 11. They smoke Kool in the house next to the horse.
    next_to(house(_,kool,_,_,_), house(_,_,horse,_,_), Houses),
    % 12. The Lucky smokers drink juice.
    member(house(_,lucky,_,juice,_), Houses),
    % 13. The Japanese smoke Parliament.
    member(house(japanese,parliament,_,_,_), Houses),
    % 14. The Norwegians live next to the blue house.
    next_to(house(norwegians,_,_,_,_), house(_,_,_,_,blue), Houses),
```

We can run this using SWI-Prolog online. SWI-Prolog includes `member` and `nextto` predicates. SWI-Prolog's `nextto` means in the order given as in clue 5.

SWI-Prolog does not include a predicate for "next to" in the sense of clues 10, 11, and 14. We are not told the order in which the elements appear. But it is easy enough to write our own, called, say, `next_to`.

```
next_to(A, B, List) :- nextto(A, B, List).
next_to(A, B, List) :- nextto(B, A, List).
```

A final issue needs resolution. None of the clues mention either a zebra or water. This implicit clue solves that problem.

```
    % 15 (implicit).
    member(house(_,_,_,water,_), Houses),
    member(house(_,_,zebra,_,_), Houses).
```

With all this in place we can get an almost instantaneous answer (manually formatted).

```
?- zebra_problem(Houses).
[
    house(norwegians, kool, fox, water, yellow),
    house(ukranians, chesterfield, horse, tea, blue),
    house(english, old_gold, snails, milk, red),
    house(spanish, lucky, dog, juice, white),
    house(japanese, parliament, zebra, coffee, green)
]
```

The Japanese have a zebra, and the Norwegians drink water.

Before going on to the Python version, let's look at how Prolog actually works. It's trivial to write a Prolog interpreter in Prolog, e.g., [1]. The following `solve` predicate is given a list with a single `goal`, which it is asked to satisfy. Each Prolog clause is available as `clause(Head, Body)`, where `Body` is a list of terms. (If `Head` is a Prolog fact, its `Body` is the empty list.)

```
solve([]).
solve([Term|Terms]):-
  clause(Term, Body),
  append(Body, Terms, New_Terms),
  solve(New_Terms).
```

It feels like cheating to use Prolog to write a Prolog interpreter. Unification and backtracking are both taken for granted. So there is hardly any work to do!

After presenting our Python version of the problem we discuss the three Python approaches to rule interpretation we developed.

To write and run the Zebra problem in Pylog we built a more general framework.

- We created a House class with named arguments.
- Each clue is expressed as a Python **def** function.
- The *Houses* list may be either a *LinkedList* or a *PyList*, i.e., *SuperSequence*.
- Users may select a house property as a pseudo-functor for displaying houses. We selected *nationality*. (See solution list below.)
- We added some simple constraint checking.

Here's how a few of the clues look.

```
  def clue_1(self, Houses: SuperSequence):
    """ 1. The English live in the red house.  """
    yield from member(House(nationality='English', color='red'), Houses)
# ...
  def clue_8(self, Houses: SuperSequence):
    """ 8. They drink milk in the middle house.
           Note the use of slice notation. Houses[2] picks the middle house.
           We can do this with both LinkedLists and PyLists.              """
    yield from unify(House(drink='milk'), Houses[2])
# ...
  def clue_11(self, Houses: SuperSequence):
    """ 11. They smoke Kool in the house next to the horse. """
    yield from next_to(House(smoke='Kool'), House(pet='horse'), Houses)
```

When run, the answer is the same. (The system helpfully reports clue evaluations.)

```
After 1392 rule applications,
  1. Norwegians(Kool, fox, water, yellow)
  2. Ukrainians(Chesterfield, horse, tea, blue)
  3. English(Old Gold, snails, milk, red)
  4. Spanish(Lucky, dog, juice, white)
  5. Japanese(Parliament, zebra, coffee, green)
The Japanese own a zebra, and the Norwegians drink water.
```

As indicated above, we developed three Python approaches to rule interpretation. We show abbreviated versions of each.

1. *forall*. We developed a *forall* construct.[11] The zebra problem is coded as follows.

```
def zebra_problem(Houses) :-
    for _ in forall{[
        % 1. The English live in the red house.
        lambda: member(house(english,_,_,_,red), Houses),
        % 2. The Spanish have a dog.
        lambda: member(house(spanish,_,dog,_,_), Houses),
        % ...
        ]}
```

forall succeeds if and only if all members of the list succeed. Each list element must be protected within a *lambda* construct to prevent premature evaluation.

forall itself is coded as follows.

[11] We also developed a parallel *forany* construct

```
def forall(lambdas: List[Callable]):
  if not lambdas:
    # They have all succeeded.
    yield
  else:
    # Execute lambdas[0]. (Note the parentheses after lambdas[0].)
    for _ in lambdas[0]( ):
      yield from forall(lambdas[1:])
```

2. *run_all_rules*. We developed a Python function that accepts a list of functions along with a list, e.g., of houses, reflecting the state of the world. It succeeds if and only if they all succeed. Following is a somewhat simplified version.

```
def run_all_clues(self, World_List: List[Term], clues: List[Callable]):
    if not clues:
      # Ran all the clues. Succeed.
      yield
    else:
      # Run the current clue and then the rest of the clues.
      for _ in clues[0](World_List):
        yield from self.run_all_clues(World_List, clues[1:])
```

3. Embed rule chaining in the rules themselves. For example,

```
  def clue_1(self, Houses: SuperSequence):
    """ 1. The English live in the red house.  """
    for _ in member(House(nationality='English', color='red'), Houses):
      yield from clue_2(Houses)

  def clue_2(self, Houses: SuperSequence):
    """ 2. The Spanish have a dog. """
    for _ in member(House(nationality='Spanish', pet='dog'), Houses):
      yield from clue_3(Houses)
  # ...
```

   A call to *clue_1* runs the problem.

The three approaches produce the same solution.

Embedding rule chaining in the clauses themselves suggests a general template.

```
    def some_clause(...):
      for _ in <generate options>:
        yield from next_clause(...)
```

This template illustrates how

- **for** loops can implement backtracking while
- **yield from** can serve as the glue between clauses.

6 Conclusion

Pylog offers a way to integrate logic programming features into a Python environment.

- The magic of unification of logic variables requires little more than linked chains.
- Prolog control structures can be implemented as nested **for** loops, with **yield** and **yield from** gluing the predicates together.

References

1. Roman Bartak. Meta-interpreters. in *Online Prolog Programming*, 1998.
2. Shai Berger. Pythologic - prolog syntax in python (python recipe). `http://code.activestate.com/recipes/`. 303057-pythologic-prolog-syntax-in-python/.
3. Carl Friedrich Bolz. A prolog interpreter in python., 2007.
4. Bruno Kim Medeiros Cesar. Prol: a minimal, inefficient prolog interpreter in a few locs of python. `https://gist.github.com/brunokim/`, 2019.
5. Christophe Delord. Pylog, 2009.
6. Bruce Frederiksen. Pyke, 2011.
7. David Kopec. *Classic Computer Science Problems in Python*, chapter 3 Constraint-satisfaction problems, pages 248 – 430. Manning, 2019.
8. Istasse Maxime. Prology: Logic programming for python3. `https://github.com/mistasse/Prology`, 2016.
9. Chris Meyers. Prolog in python. `http://www.openbookproject.net/py4fun/prolog/prolog1.html`, 2015.
10. Nikola Miljkovic. Python prolog interpreter. `https://github.com/photonlines/Python-Prolog-Interpreter`Python Prolog Interpreter, 2019.
11. Gustavo Niemeyer and Sbastien Celles. python-constraint library. pypi page: python-constraint, 2019. Also discussed in Popovi, Olivera (2019) Constraint Programming with python-constraint.
12. Ian Piumarta. Lecture notes and slides from weeks 5-7 of a course on programming paradigms. `http://www.ritsumei.ac.jp/~piumarta/pl/`, 2017.
13. Gaurav Raturi. 5 best programming languages to choose for developing innovative ai solutions, *Becoming Human*, 2019.
14. Matthew Rocklin. Kanren: Logic programming in python, 2019.
15. Paresh Sagar. Powerful programming languages to code your ai application. *medium.com*, 2019.
16. Claudio Santini. Pampy: The pattern matching for python you always dreamed of, 2018.
17. Jeff Thompson. Yield prolog, 2017.
18. Markus Triska. The Boolean constraint solver of SWI-Prolog: System description. In *FLOPS: Functional and Logic Programming. 13th International Symposium*, volume 9613 of *LNCS*, pages 45–61, 2016.