Coloring US States with Prolog using Pytholog

This tutorial demonstrates how to use the Pytholog library to integrate Python with Prolog, focusing on a constraint satisfaction problem of coloring US states.

Setting Up

First, install the Pytholog library:

```
pip install pytholog
```

Creating the Knowledge Base

Next, create a Python script (e.g., `city_color.py`) with the following code:

```python
import pytholog as pl

# Create a new knowledge base object
city_color = pl.KnowledgeBase("city_color")

# Define the rules and facts
city_color([
    "different(red, green)",
    "different(red, blue)",
    "different(green, red)",
    "different(green, blue)",
    "different(blue, red)",
    "different(blue, green)",
    "coloring(A, M, G, T, F) :- different(M, T), different(M, A),different(A, T),different(A, M),different(A, G),different(A, F),different(G,F),different(G, T)"
])

# Query the knowledge base
answer = city_color.query(pl.Expr("coloring(Alabama, Mississippi, Georgia, Tennessee, Florida)"))
```

This code sets up a knowledge base for coloring US states with three colors (red, green, and blue), ensuring that adjacent states have different colors.

Understanding the Code

The knowledge base consists of:

- Facts defining which colors are different from each other
- A rule `coloring(A, M, G, T, F)` that ensures different colors for adjacent states

The query asks for a valid coloring of Alabama, Mississippi, Georgia, Tennessee, and Florida.

Extending the Example

To improve the code organization, you can separate the Prolog knowledge base into a separate file:

1. Create a file named `city_color.pl` with the following content:

```
different(red, green).
different(red, blue).
different(green, red).
different(green, blue).
different(blue, red).
different(blue, green).
coloring(A, M, G, T, F) :- different(M, T), different(M, A), different(A, T),
different(A, M), different(A, G), different(A, F), different(G, F), different(G,
T).
```

2. Create a Python module (`prolog_converter.py`) to read the Prolog file:

```
# prolog_converter.py
# (see code and explanation below)
```

3. Create the main Python script (`city_color.py`):

```
import pytholog as pl
from prolog_converter import converter

prolog_file = 'city_color.pl'
pytholog_kb = converter.parse_prolog_file(prolog_file)
print(pytholog_kb)

city_color = pl.KnowledgeBase("city_color")
city_color(pytholog_kb)

answer = city_color.query(pl.Expr("coloring(Alabama, Mississippi, Georgia,
Tennessee, Florida)"))
```

This approach separates the Prolog knowledge base from the Python code, making it easier to maintain and modify the rules.

Exercise

As an exercise, you can try to identify and remove the redundant predicate in the `coloring` rule. After making the change, run the code to validate that it still works correctly.

Reading a Prolog knowledge base in Python

Here's the code for the `prolog_converter.py` file:

```python
def parse_prolog_file(file_path):
    clauses = []

    # Read the file
    with open(file_path, "r") as file:
        lines = file.readlines()

    # Process each line to extract individual clauses
    current_clause = ""
    for line in lines:
        line = line.strip()  # Remove leading/trailing whitespace
        if line and not line.startswith("%"):  # Ignore empty lines and comments
            if line.endswith('.'):  # Check if the line ends with a dot
                current_clause += line[:-1]  # Remove the dot and append to the
current clause
                clauses.append(current_clause)  # Add the completed clause to the
list
                current_clause = ""  # Reset the current_clause for the next
clause
            else:
                current_clause += line  # Append the line to the current clause

    return clauses

'''
# Example usage:
clauses = read_clauses_from_file("kb.pl")
print(clauses)
'''
```

This Python script defines a function `parse_prolog_file` that reads and parses a Prolog file, extracting individual clauses. Here's a breakdown of how it works:

1. The function takes a `file_path` as an argument, which is the path to the Prolog file to be parsed.

2. It initializes an empty list called `clauses` to store the extracted Prolog clauses.

3. The file is opened and read using a `with` statement, which ensures proper file handling and closure.

4. The script then iterates through each line of the file:

    - It strips leading and trailing whitespace from each line.
    - Empty lines and comments (lines starting with '%') are ignored.

5. For non-empty, non-comment lines:

    - If a line ends with a dot ('.'), it's considered the end of a clause:

        - The dot is removed, and the line is appended to the `current_clause`.
        - The completed `current_clause` is added to the `clauses` list.
        - `current_clause` is reset for the next clause.
    - If a line doesn't end with a dot, it's appended to `current_clause` (handling multi-line clauses).

6. Finally, the function returns the list of extracted clauses.

The commented-out section at the end shows an example of how to use this function, calling it with a Prolog file named `kb.pl` and printing the resulting clauses.

```
different(red, green).
different(red, blue).
different(green, red).
different(green, blue).
different(blue, red).
different(blue, green).
coloring(A, M, G, T, F) :-
    different(M, T),
    different(M, A),
    different(A, T),
    different(A, M),
    different(A, G),
    different(A, F),
    different(G, F),
    different(G, T).
```

This Prolog knowledge base defines a constraint satisfaction problem for coloring adjacent US states using three colors: red, green, and blue. Let's break down the components:

Facts: Color Differences

The first six lines define facts about which colors are different from each other:

```
different(red, green).
different(red, blue).
different(green, red).
different(green, blue).
different(blue, red).
different(blue, green).
```

These facts establish that red is different from green and blue, green is different from red and blue, and blue is different from red and green. This set of facts is symmetric, meaning if color X is different from color Y, then color Y is also different from color X.

Rule: State Coloring

The last part of the code defines a rule for coloring five states:

```
coloring(A, M, G, T, F) :-
    different(M, T),
    different(M, A),
    different(A, T),
    different(A, M),
    different(A, G),
    different(A, F),
    different(G, F),
    different(G, T).
```

This rule, named `coloring`, takes five variables as input, representing five states (likely Alabama, Mississippi, Georgia, Tennessee, and Florida, based on the Python code in the search results). The rule is satisfied if all the conditions after the `:-` symbol are true.

Each `different` predicate in the rule body ensures that adjacent states have different colors:

- M (Mississippi) must be different from T (Tennessee) and A (Alabama)
- A (Alabama) must be different from T (Tennessee), M (Mississippi), G (Georgia), and F (Florida)
- G (Georgia) must be different from F (Florida) and T (Tennessee)

This rule effectively encodes the constraint that no two adjacent states can have the same color.

Redundancy in the Rule

It's worth noting that there is a redundant predicate in the `coloring` rule. The condition `different(A, M)` is stated twice, which is unnecessary and can be removed without changing the behavior of the rule.

By using this knowledge base with a Prolog engine or a Prolog-like system such as Pytholog, one can query for valid colorings of these five states that satisfy all the constraints defined in the facts and rule.

ANNEX: How to create the module `prolog_converter`

Here's how the directory structure would look:

```
project_directory/
    ├── colors.py
    └── prolog_converter/
        ├── __init__.py
        └── prolog_converter.py
```

With this structure, you can import the converter module in colors.py using:

```
from prolog_converter import converter
```

This import statement works because:

1. Python looks for modules in the same directory as the script being executed (`colors.py` in this case).
2. It finds the 'prolog_converter' directory, which is treated as a package due to the presence of `__init__.py`.
3. The `__init__.py` file should import the converter function from `prolog_converter.py`