vjwukP54Uqlh4bQ

April 11, 2025

[3]:
```
from expert_ceylon import *
import random
```

1 Classes

1.1 WinTotals

[4]:
```
class WinTotals(Fact):
    human = Field(int, default=0)
    computer = Field(int, default=0)
    ties = Field(int, default=0)
```

1.2 Results

[5]:
```
class Results(Fact):
    winner = Field(str, mandatory=True)
    loser = Field(str, mandatory=True)
    why = Field(str, mandatory=True)
```

1.3 ValidAnswer

[6]:
```
class ValidAnswer(Fact):
    answer = Field(str, mandatory=True)
    key = Field(str, mandatory=True)
```

1.4 Action, HumanChoice, ComputerChoice

[7]:
```
class Action(Fact):
    pass


class HumanChoice(Fact):
    pass


class ComputerChoice(Fact):
    pass
```

1.5 RockPaperScissors

[8]:
```
# Subclass KnowledgeEngine
# declaring facts: declare(Fact())
# retracting facts: retract(Fact())

class RockPaperScissors(KnowledgeEngine):

    @DefFacts()
    def game_rules(self):
        """Declare game rules and valid input keys for the user."""
        print('DefFacts: setting up game rules.')
        self.valid_answers = dict()

        yield Results(winner='rock', loser='scissors', why='Rock smashes␣
 ↪scissors')
        yield Results(winner='paper', loser='rock', why='Paper covers rock')
        yield Results(winner='scissors', loser='paper', why='Scissors cut␣
 ↪paper')
        yield ValidAnswer(answer='rock', key='r')
        yield ValidAnswer(answer='paper', key='p')
        yield ValidAnswer(answer='scissors', key='s')

        print('All rules in place ...')



    @Rule()
    def startup(self):
        print("Lets play a game!")
        print("You choose rock, paper, or scissors,")
        print("and I'll do the same.")
        self.declare(WinTotals(human=0, computer=0, ties=0))
        self.declare(Action('get-human-move'))

    @Rule(NOT(Action()),
          ValidAnswer(answer=MATCH.answer, key=MATCH.key))
    def store_valid_answers(self, answer, key):
        print('store_valid_answers', key, answer)
        self.valid_answers[key] = answer

    #
    # HUMAN MOVE RULES
    #

    @Rule(Action('get-human-move'))
    def get_human_move(self):
```

```
        question = ", ".join("{name} ({key})".format(name=a[1].title(),
                                                     key=a[0].upper()) for a in␣
↪self.valid_answers.items()) + '? '
        res = input(question).lower()
        self.declare(HumanChoice(res))

    @Rule(AS.f1 << HumanChoice(MATCH.choice),
          ValidAnswer(answer=MATCH.answer, key=MATCH.choice),
          AS.f2 << Action('get-human-move'))
    def good_human_move(self, f1, f2, answer):
        self.retract(f1)
        self.retract(f2)
        self.declare(HumanChoice(answer))
        self.declare(Action('get-computer-move'))

    # retracting a rule from inference engine
    # retract() Removes an existing fact from the factlis
    @Rule(AS.f1 << HumanChoice(MATCH.choice),
          NOT(ValidAnswer(key=MATCH.choice)),
          AS.f2 << Action('get-human-move'))
    def bad_human_move(self, f1, f2, choice):
        print("Sorry %s is not a valid answer" % choice)
        self.retract(f1)
        self.retract(f2)
        self.declare(Action('get-human-move'))

    #
    # COMPUTER MOVE RULES
    #
    @Rule(AS.f1 << Action('get-computer-move'))
    def get_computer_move(self, f1):
        choice = random.choice(list(self.valid_answers.values()))
        self.retract(f1)
        self.declare(ComputerChoice(choice))
        self.declare(Action('determine-results'))

    #
    # WIN DETERMINATION RULES
    #
    @Rule(AS.f1 << Action('determine-results'),
          AS.f2 << ComputerChoice(MATCH.cc),
          AS.f3 << HumanChoice(MATCH.hc),
          AS.w << WinTotals(computer=MATCH.cw),
          Results(winner=MATCH.cc, loser=MATCH.hc, why=MATCH.explanation))
    def computer_wins(self, f1, f2, f3, w, cw, explanation):
        self.retract(f1)
        self.retract(f2)
```

```
        self.retract(f3)
        self.modify(w, computer=cw + 1)
        print("Computer wins!", explanation)
        self.declare(Action('determine-play-again'))

    @Rule(AS.f1 << Action('determine-results'),
          AS.f2 << ComputerChoice(MATCH.cc),
          AS.f3 << HumanChoice(MATCH.hc),
          'w' << WinTotals(human=MATCH.hw),
          Results(winner=MATCH.hc, loser=MATCH.cc, why=MATCH.explanation))
    def humans_wins(self, f1, f2, f3, w, hw, explanation):
        self.retract(f1)
        self.retract(f2)
        self.retract(f3)
        self.modify(w, human=hw + 1)
        print("You win!", explanation)
        self.declare(Action('determine-play-again'))

    # @Rule(AS.f1 << Action('determine-results'),
    #       AS.f2 << ComputerChoice(MATCH.cc),
    #       AS.f3 << HumanChoice(MATCH.cc),
    #       AS.w << WinTotals(ties=MATCH.nt))
    # def tie(self, f1, f2, f3, w, nt):
    #     self.retract(f1)
    #     self.retract(f2)
    #     self.retract(f3)
    #     self.modify(w, ties=nt + 1)
    #     print("Tie! Ha-ha!")
    #     self.declare(Action('determine-play-again'))

    #
    # PLAY AGAIN RULE
    #
    @Rule(AS.f1 << Action('determine-play-again'),
          WinTotals(computer=MATCH.ct, human=MATCH.ht, ties=MATCH.tt))
    def play_again(self, f1, ct, ht, tt):
        self.retract(f1)
        if not self.yes_or_no("Play again?"):
            print("You won", ht, "game(s).")
            print("Computer won", ct, "game(s).")
            print("We tied", tt, "game(s).")
            self.halt()
        else:
            self.declare(Action('get-human-move'))

    def yes_or_no(self, question):
        return input(question).upper().startswith('Y')
```

2 Play

[9]:
```
rps = RockPaperScissors()
```

[10]:
```
rps.reset()
rps.run()
```

```
DefFacts: setting up game rules.
All rules in place …
store_valid_answers s scissors
store_valid_answers p paper
store_valid_answers r rock
Lets play a game!
You choose rock, paper, or scissors,
and I'll do the same.
```