Defworld Documentation

Release 0.0.1

Yu Jae-myoung

August 21, 2012

CONTENTS

CHAPTER

ONE

DEFWORLD

Defworld is a multiagent simulation library for interactive storytelling.

1.1 Rule

Each agent manipulates ideas on its working memory(WM) according to a set of IF-THEN style rules. For example, the following code means "If Socrates is a human, then Socrates is mortal".

```
Rule('socrates-mortal', [          # A 'socrates-mortal' rule:
    Fact('Socrates is a human'),   # If Socrates is a human,
    ],[                            # then
    Fact('Socrates is mortal')])   # Socrates is mortal.
```

`defworld.patterns.Fact` is a WM item about facts. The 'socrates-mortal' rule append a fact "Socrates is mortal" on the WM when there is a fact "Sorcrates is a human" on WM.

On the other hand, some rules executes actions. For example, the follwing code means "If the phone rings, answer the phone"

```
Rule('answer-phone', [             # A 'answer-phone' rule:
    Fact('the phone rings'),       # If the phone rings,
    ],[                            # then
    Action('answer the phone')])   # answer the phone
```

`defworld.patterns.Action` is also a WM item. This 'answer-phone' rule append the `defworld.patterns.Action` , 'answer the phone' on WM, and then ActionModule executes the action in the designated way.

Every WM item has a name and slots. For example,

```
Fact('human', name='Socrates')
Action('answer', target='phone')
```

Slot values could be variables. Defworld fills these variables with pattern matching.

```
Rule('human-mortal', [               # A 'human-mortal' rule:
    Fact('human', name=Var('x')),    # If x is a human
    ],[                              # then
    Fact('mortal', name=Var('x'))]) # x is mortal
```

If there is a fact `Fact('human', name='Socrates')` on WM, then a new item `Fact('mortal', name='Socrates')` is appended on WM.

1.2 Agent

An agent has initial facts and rules. Initial facts are loaded on WM when the agents are created.

```
Agent(
    # initial facts
    [ Fact('human', name='Socrates') ], # A human whose name is 'Socrates'
    # rules
    [ Rule('socrates-mortal', [          # A 'socrates-mortal' rule:
        Fact('human', name=Var('x')),    # If x is a human,
        ],[                              # then
        Fact('mortal', name=Var('x'))]) # x is mortal.
    ])
```

1.2.1 defworld — Interactive storytelling

defworld.agent — Agents

class defworld.agent.**Agent**(*initial_facts*, *rules*)
A forward inference agent

activate()
Activate the rule on the top of agenda

linearize()
Return linearized rete tree

class defworld.agent.**WorkingMemory**(*initial_facts*=[ ])
A working memory of an Agent which stores patterns.Fact and patterns.Entity

Parameters **initial_facts** (a list of patterns.Fact) –

append(*fact*)
Append a Fact to the WorkingMemory

Parameters **fact** (patterns.Fact) – a pointer to PixelWand to compare

defworld.basic — Basic types

defworld.exception — Errors

exception defworld.exception.**DiffFactName**
raised when trying to match facts whose names are different. To prevent Not test catching the exception.

defworld.patterns — Patterns

class defworld.patterns.**Assign**(*var*, *pattern*)
Assign a fact to a variable

Parameters

- **var** (basic.Var) – a variable
- **pattern** (Fact) – a pattern of the fact to assign

match(*other=None*, *item=None*)
Return pattern matching

class defworld.patterns.**Entity**(*_name*, **args*, ***kwargs*)
A Fact represents a unique object.

apply(*wm*)
Alter the entity the name of which is same with one of this entity. If the working memory does not have one, then just append this to it.

Parameters **wm** (agent.WorkingMemory) – An agent's working memory

class defworld.patterns.**Fact**(*_name*, **args*, ***kwargs*)

Parameters

- **_name** (basestring or basic.Var) – a kind of Fact
- **args** – ordered values
- **kwargs** – slot values

apply(*wm*)
Append self to the working memory

Parameters **wm** (agent.WorkingMemory) – An agent's working memory

class defworld.patterns.**Func**(*func*)
Function wrapper to evaluate the function at the time the rule is activated.

Parameters **func** – the function to evaluate

class defworld.patterns.**Retract**(*fact*)
Retract a fact from the working memory

Parameters **fact** (Fact) – the fact to retract

class defworld.patterns.**Template**(*name*, *kind=<class 'defworld.patterns.Fact'>*)
Template for a Fact or Entity

```
human = Template('human')
human('Socrates') == Fact('human', 'Socrates')
```

Parameters

- **name** – the name of Fact
- **kind** – Fact (default) or Entity

class defworld.patterns.**Update**(*fact*, **args*, ***kwargs*)
Update a fact in the working memory

Parameters

- **fact** (Fact) – the fact to update
- **args** – ordered values
- **kwargs** – slot values

defworld.rete.node — Rete network

class defworld.rete.node.**Kind**(*wm*)
Kind nodes manager

CHAPTER

TWO

EXAMPLES

2.1 Linear Equation Solver

You can use defworld library not only for interactive storytelling, but also for rule-based application. In this example, I explain how to build a linear equation solver using defworld.

A linear equation is an equation in which each term is either a constant or the product of a constant and a single variable, e.g. 2x + 3 = 5. Solving methods are quite easy: subtract a same number from both side or divide both side by a same number repeatedly until the unknown remains solely on left side and a constant on right side.

2.1.1 Terminal condition

Let's define the terminal condition, first.

```
Rule('x = b', [
        Not(solution()),  # not yet solved
        equation(x, b)    # x = b
    ],[
        solved(b)         # the solution is b
    ]),
```

`solution` is a `defworld.patterns.Template` for the solution, defined by the following:

```
solution = Template('solution', Entity)
```

`equation` is a `defworld.patterns.Template` for an `defworld.patterns.Entity` represents the equation.

```
equation = Template('equation', Entity)
```

`solved` is also a function returns an `defworld.patterns.Entity`.

```
@Func
def solved(x):
    print x.value
    return solution(value=x)
```

This function is called when the rule 'x = b' is fired. That's why `solved` is decorated by `defworld.patterns.Func`.

To sum up, the 'x = b' rule is fired when the equation is not yet solved and is in the form of 'x = b'. Once the rule is fired, the solution is appended on the agent's working memory and printed out.

2.1.2 Subtracting a same number from both side

If there is an equation $x + 2 = 5$, you just subtract 2 from both side to solve this equation. In this section, I show you how to implement this rule.

```
Rule('expr + a = b', [
        Not(solution()),            # not yet solved
        equation(add(expr, a), b)   # expr + a = b
    ],[
        equation(expr, Sub(b, a))   # expr = b - a
    ]),
```

That's it. `add` is a function returns a `defworld.patterns.Fact` represents addition, defined by the following:

```
add = Template('add')
```

`Sub` is a function decorated by `defworld.patterns.Func` which returns the difference between two value.

2.1.3 Full source code

This is the full source code.

```
from defworld.agent import Agent
from defworld.basic import Var
from defworld.patterns import Entity, Func, Not, Rule, Template

if __name__ == '__main__':
    solution = Template('solution', Entity)

    a = Var('a')
    b = Var('b')
    expr = Var('expr')
    x = Entity('unknown')

    product = Template('product')
    add = Template('add')

    equation = Template('equation', Entity)

    @Func
    def Div(a, b):
        return a/b

    @Func
    def Sub(a, b):
        return a-b

    @Func
    def solved(x):
        print x.value
        return solution(x)

    agent = Agent([
            equation(add(product(2,x), 3), 5)       # 2x + 3 = 5
        ],[
            Rule('x = b', [
                    Not(solution()),  # not yet solved
                    equation(x, b)    # x = b
```

```
            ],[
                solved(b)          # the solution is b
            ]),

        Rule('a * expr = b', [
                Not(solution()),                # not yet solved
                equation(product(a, expr), b)   # a * expr = b
            ],[
                equation(expr, Div(b, a))       # expr = b / a
            ]),

        Rule('expr + a = b', [
                Not(solution()),            # not yet solved
                equation(add(expr, a), b)   # expr + a = b
            ],[
                equation(expr, Sub(b, a))   # expr = b - a
            ]),
    ])
agent.match()
agent.run() # print 1
```

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

d