# pyknow Documentation

*Release 0.0.8*

Roberto Abdelkadder Martinez, David Francos Cuartero

May 29, 2017

# Contents

PyKnow is a Python library for building expert systems strongly inspired by CLIPS.

```
from random import choice
from pyknow import *


class Light(Fact):
    """Info about the traffic light."""
    pass


class RobotCrossStreet(KnowledgeEngine):
    @Rule(Light(color='green'))
    def green_light(self):
        print("Walk")

    @Rule(Light(color='red'))
    def red_light(self):
        print("Don't walk")

    @Rule('light' << Light(color=L('yellow') | L('blinking-yellow')))
    def cautious(self, light):
        print("Be cautious because light is", light["color"])
```

```
>>> engine = RobotCrossStreet()
>>> engine.reset()
>>> engine.declare(Light(color=choice(['green', 'yellow', 'blinking-yellow', 'red'])))
>>> engine.run()
Be cautious because light is blinking-yellow
```

You can find some more examples on GitHub.

CHAPTER 1

# User Guide

## Introduction

### Philosophy

We aim to implement a Python alternative to CLIPS, as compatible as possible. With the goal of making it easy for the CLIPS programmer to transfer all of his/her knowledge to this platform.

### Features

- Python 3 compatible.
- Pure Python implementation.
- Matcher based on the RETE algorithm.

### Difference between CLIPS and PyKnow

1. CLIPS is a programming language, PyKnow is a Python library. This imposes some limitations on the constructions we can do (specially on the LHS of a rule).
2. CLIPS is written in C, PyKnow in Python. A noticeable impact in performance is to be expected.
3. In CLIPS you add facts using *assert*, in Python *assert* is a keyword, so we use *declare* instead.

## Installation

### From PyPI

To install PyKnow, run this command in your terminal:

```
$ pip install pyknow
```

### Getting the source code

PyKnow is developed on Github.

You can clone the repository using the git command:

```
$ git clone https://github.com/buguroo/pyknow.git
```

Or you can download the releases in .zip or .tar.gz format.

Once you have a copy of the source, you can install it running this command:

```
$ python setup.py install
```

## The Basics

An expert system is a program capable of pairing up a set of **facts** with a set of **rules** to those facts, and execute some actions based on the matching rules.

### Facts

*Facts* are the basic unit of information of PyKnow. They are used by the system to reason about the problem.

Let's enumerate some facts about *Facts*, so... metafacts ;)

1. The class *Fact* is a subclass of *dict*.

```
>>> f = Fact(a=1, b=2)
>>> f['a']
1
```

2. Therefore a *Fact* does not mantain an internal order of items.

```
>>> Fact(a=1, b=2)  # Order is arbirary :O
Fact(b=2, a=1)
```

3. In contrast to *dict*, you can create a *Fact* without keys (only values), and *Fact* will create a numeric index for your values.

```
>>> f = Fact('x', 'y', 'z')
>>> f[0]
'x'
```

4. You can mix autonumeric values with key-values, but autonumeric must be declared first:

```
>>> f = Fact('x', 'y', 'z', a=1, b=2)
>>> f[1]
'y'
>>> f['b']
2
```

5. You can subclass *Fact* to express different kinds of data or extend it with your custom functionality.

```
class Alert(Fact):
    """The alert level."""
    pass

class Status(Fact):
    """The system status."""
    pass

f1 = Alert('red')
f2 = Status('critical')
```

```
from pyknow import Fact
from django.contrib.auth.models import User as DjangoUser

class User(Fact):
    @classmethod
    def from_django_model(cls, obj):
        return cls(pk=obj.pk,
                   name=obj.name,
                   email=obj.email)

    def save_to_db(self):
        return DjangoUser.create(**self)
```

## Rules

In PyKnow a **rule** is a callable, decorated with *Rule*.

Rules have two components, LHS (left-hand-side) and RHS (right-hand-side).

- The *LHS* describes (using **patterns**) the conditions on which the rule * should be executed (or fired).
- The *RHS* is the set of actions to perform when the rule is fired.

For a *Fact* to match a *Pattern*, all pattern restrictions must be **True** when the *Fact* is evaluated against it.

```
class MyFact(Fact):
    pass

@Rule(MyFact())  # This is the LHS
def match_with_every_myfact():
    """This rule will match with every instance of `MyFact`."""
    # This is the RHS
    pass

@Rule(Fact('animal', family='felinae'))
def match_with_cats():
    """
    Match with every `Fact` which:

      * f[0] == 'animal'
      * f['family'] == 'felinae'

    """
    print("Meow!")
```

You can use logic operators to express complex *LHS* conditions.

```
@Rule((User('admin') | User('root'))
      & ~Fact('drop-privileges'))
def the_user_has_power():
    """
    The user is a privileged one and we are not dropping privileges.

    """
    enable_superpowers()
```

For a *Rule* to be useful, it must be a method of a *KnowledgeEngine* subclass.

### *Facts* vs *Patterns*

The difference between *Facts* and *Patterns* is small. In fact, *Patterns* are just *Facts* containing **Pattern Conditional Elements** instead of regular data. They are used only in the *LHS* of a rule.

If you don't provide the content of a pattern as a **PCE**, PyKnow will enclose the value in a *LiteralPCE* automatically for you.

Also, you can't declare any Fact containing a **PCE**, if you do, you will receive a nice exception back.

```
>>> ke = KnowledgeEngine()
>>> ke.declare(Fact(L("hi")))
Traceback (most recent call last):
  File "<ipython-input-4-b36cff89278d>", line 1, in <module>
    ke.declare(Fact(L('hi')))
  File "/home/pyknow/pyknow/engine.py", line 210, in declare
    self.__declare(*facts)
  File "/home/pyknow/pyknow/engine.py", line 191, in __declare
    "Declared facts cannot contain conditional elements")
TypeError: Declared facts cannot contain conditional elements
```

## DefFacts

Most of the time expert systems needs a set of facts to be present for the system to work. This is the purpose of the *DefFacts* decorator.

```
@DefFacts()
def needed_data():
    yield Fact(best_color="red")
    yield Fact(best_body="medium")
    yield Fact(best_sweetness="dry")
```

All *DefFacts* inside a KnowledgeEngine will be called every time the *reset* method is called.

**Note:** The decorated method MUST be generators.

## KnowledgeEngine

This is the place where all the magic happens.

The first step is to make a subclass of it and use *Rule* to decorate its methods.

After that, you can instantiate it, populate it with facts, and finally run it.

Listing 1.1: greet.py

```python
from pyknow import *

class Greetings(KnowledgeEngine):
    @DefFacts()
    def _initial_action(self):
        yield Fact(action="greet")

    @Rule(Fact(action='greet'),
          NOT(Fact(name=W())))
    def ask_name(self):
        self.declare(Fact(name=input("What's your name? ")))

    @Rule(Fact(action='greet'),
          NOT(Fact(location=W())))
    def ask_location(self):
        self.declare(Fact(location=input("Where are you? ")))

    @Rule(Fact(action='greet'),
          Fact(name="name" << W()),
          Fact(location="location" << W()))
    def greet(self, name, location):
        print("Hi %s! How is the weather in %s?" % (name, location))

engine = Greetings()
engine.reset()  # Prepare the engine for the execution.
engine.run()  # Run it!
```

```
$ python greet.py
What's your name? Roberto
Where are you? Madrid
Hi Roberto! How is the weather in Madrid?
```

### Handling facts

The following methods are used to manipulate the set of facts the engine knows about.

#### *declare*

Adds a new fact to the factlist (the list of facts known by the engine).

```
>>> engine = KnowledgeEngine()
>>> engine.reset()
>>> engine.declare(Fact(score=5))
<f-1>
>>> engine.facts
<f-0> InitialFact()
<f-1> Fact(score=5)
```

**Note:** The same fact can't be declared twice unless *facts.duplication* is set to *True*.

### *retract*

Removes an existing fact from the factlist.

Listing 1.2: Both, the index and the fact can be used with retract

```
>>> engine.facts
<f-0> InitialFact()
<f-1> Fact(score=5)
<f-2> Fact(color='red')
>>> engine.retract(1)
>>> engine.facts
<f-0> InitialFact()
<f-2> Fact(color='red')
```

### *modify*

Retracts some fact from the factlist and declares a new one with some changes. Changes are passed as arguments.

```
>>> engine.facts
<f-0> InitialFact()
<f-1> Fact(color='red')
>>> engine.modify(engine.facts[1], color='yellow', blink=True)
<f-2>
>>> engine.facts
<f-0> InitialFact()
<f-2> Fact(color='yellow', blink=True)
```

### *duplicate*

Adds a new fact to the factlist using an existing fact as a template and adding some modifications.

```
>>> engine.facts
<f-0> InitialFact()
<f-1> Fact(color='red')
>>> engine.duplicate(engine.facts[1], color='yellow', blink=True)
<f-2>
>>> engine.facts
<f-0> InitialFact()
<f-1> Fact(color='red')
<f-2> Fact(color='yellow', blink=True)
```

## Cycle of execution: DefFacts, reset & run

Because this topic is often a direct cause of misunderstanding, it deserves a special mention here, in the basics.

For a KnowledgeEngine to run, this things must happen:

1. The class must be instantiated, of course.
2. The **reset** method must be called:
   - This declares the special fact *InitialFact*. Necessary for some rules to work properly.
   - Declare all facts yielded by the methods decorated with *@DefFacts*.

3. The **run** method must be called. This starts the cycle of execution.

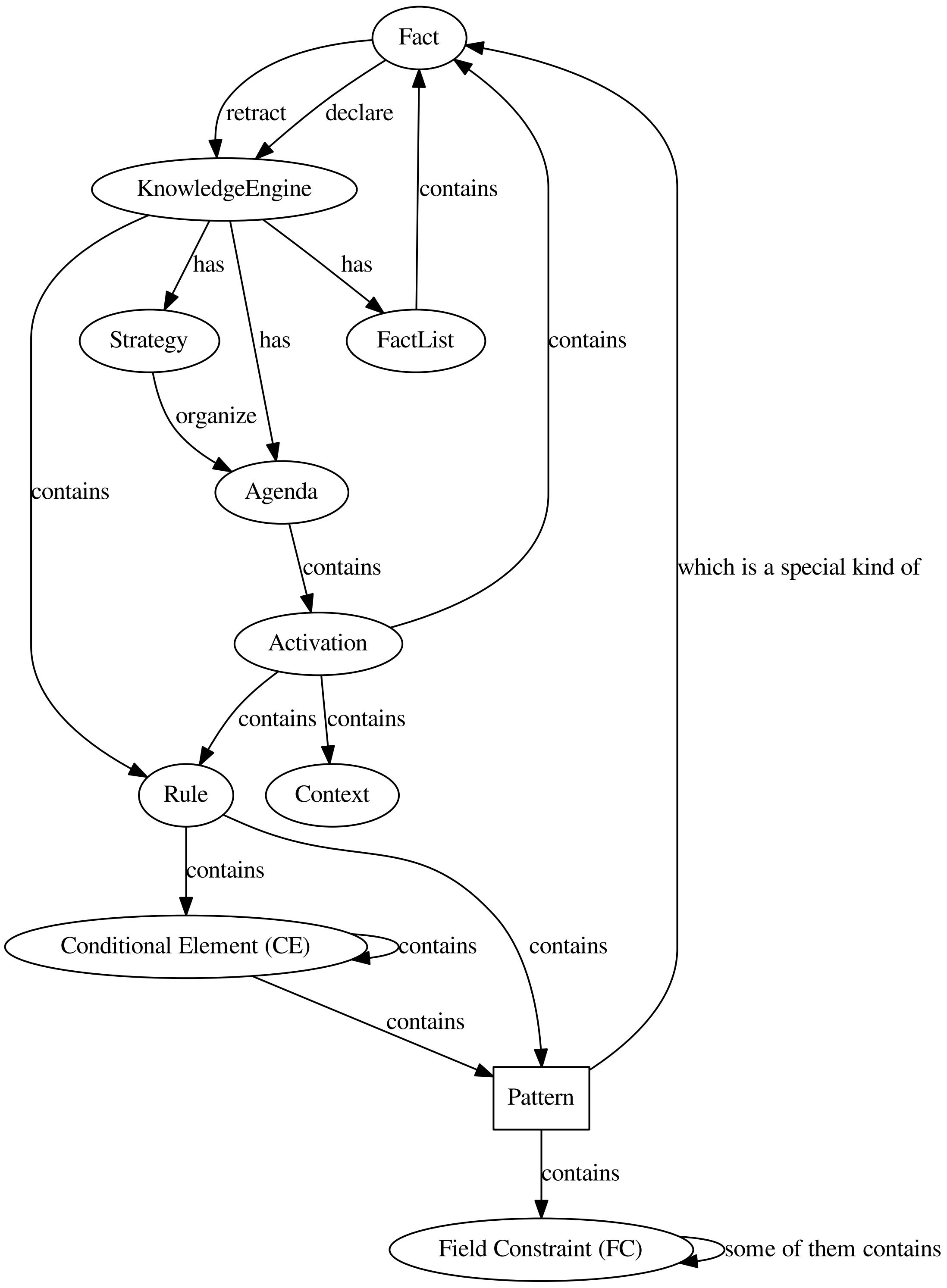
### Difference between *DefFacts* and *declare*

Both are used to declare facts on the engine instance, but:

- *declare* adds the facts directly to the working memory.
- Generators declared with *DefFacts* are called by the **reset** method, and all the yielded facts they are added to the working memory using *declare*.

## Reference

The following diagram shows all the system components and the relationships among them.

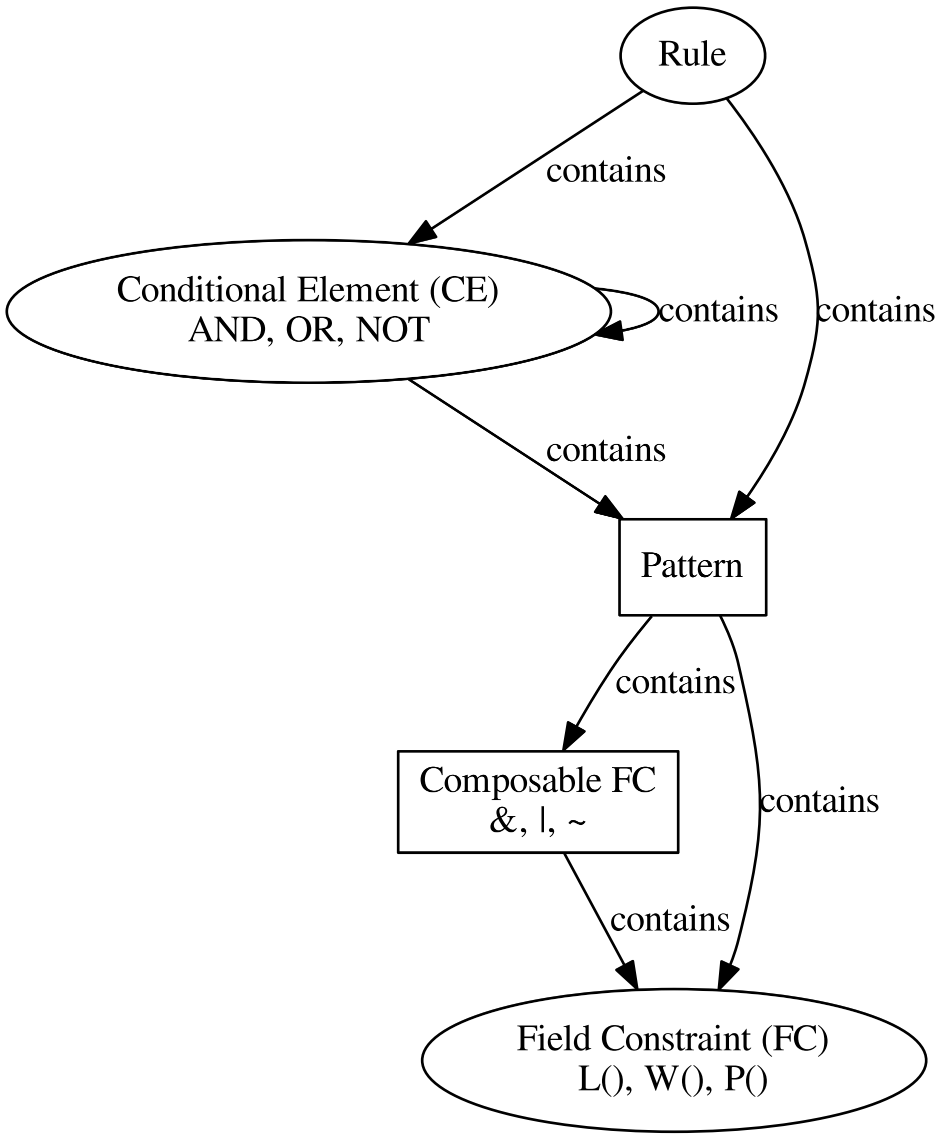

Fact
retract
declare
KnowledgeEngine
contains
has
has
Strategy
has
FactList
contains
organize
contains
Agenda
contains
which is a special kind of
Activation
contains
contains
Rule
Context
contains
Conditional Element (CE)
contains
contains
contains
Pattern
contains
Field Constraint (FC)
some of them contains

## Rule

*Rule* is the basic method of composing patterns. You can add as many patterns or conditional elements as you want to a Rule and it will fire if every one of them matches. Therefore, it behaves like *AND* by default.

```
@Rule(<pattern_1>,
      <pattern_2>,
      ...
      <pattern_n>)
def _():
    pass
```

The following diagram shows the rules of composition of a rule:



### salience

This value, by default *0*, determines the priority of the rule in relation to the others. Rules with a higher salience will be fired before rules with a lower one.

Listing 1.3: *r1* has precedence over *r2*

```
@Rule(salience=1)
def r1():
    pass
```

```
@Rule(salience=0)
def r2():
    pass
```

## Conditional Elements: Composing Patterns Together

### AND

*AND* creates a composed pattern containing all Facts passed as arguments. All of the passed patterns must match for the composed pattern to match.

Listing 1.4: Match if two facts are declared, one matching Fact(1) and other matching Fact(2)

```
@Rule(AND(Fact(1),
          Fact(2)))
def _():
    pass
```

### OR

*OR* creates a composed pattern in which any of the given pattern will make the rule match.

Listing 1.5: Match if a fact matching Fact(1) exists **and/or** a fact matching Fact(2) exists

```
@Rule(OR(Fact(1),
         Fact(2)))
def _():
    pass
```

**Warning:** If multiple facts match, the rule will be fired multiple times, one for each valid combination of matching facts.

### NOT

This element matches if the given pattern does not match with any fact or combination of facts. Therefore this element matches the *absence* of the given pattern.

Listing 1.6: Match if no fact match with Fact(1)

```
@Rule(NOT(Fact(1)))
def _():
    pass
```

### TEST

Check the received callable against the current binded values. If the execution returns *True* the evaluation will continue and stops otherwise.

Listing 1.7: Match for all numbers *a*, *b*, *c* where a > b > c

```
@Rule(Number('a' << W()),
      Number('b' << W()),
      TEST(lambda a, b: a > b),
      Number('c' << W()),
      TEST(lambda b, c: b > c))
def _(a, b, c):
    pass
```

### EXISTS

This CE receives a pattern and matches if one or more facts matches this pattern. This will match only once while one or more matching facts exists and will stop matching when there is no matching facts.

Listing 1.8: Match once when one or more Color exists

```
@Rule(EXISTS(Color()))
def _():
    pass
```

### FORALL

The FORALL conditional element provides a mechanism for determining if a group of specified CEs is satisfied for every occurence of another specified CE.

Listing 1.9: Match when for every Student fact there is a Reading, Writing and Arithmetic fact with the same name.

```
@Rule(FORALL(Student(W('name')),
             Reading(W('name')),
             Writing(W('name')),
             Arithmetic(W('name')))
def all_students_passed():
    pass
```

**Note:** All binded variables captured inside a *FORALL* clause won't be passed as context to the RHS of the rule.

**Note:** Any time the rule is activated the matching fact is the InitialFact.

## Field Constraints: FC for sort

### L (Literal Field Constraint)

This element performs an exact match with the given value. The matching is done using the equality operator ==.

Listing 1.10: Match if the first element is exactly *3*

```
@Rule(Fact(L(3)))
def _():
    pass
```

**Note:** This is the default FC used when no FC is given as a pattern value. pattern.

### W (Wildcard Field Constraint)

This element matches with **any** value.

Listing 1.11: Match if some fact is declared with the key *mykey*.

```
@Rule(Fact(mykey=W()))
def _():
    pass
```

**Note:** This element **only** matches if the element exist.

### P (Predicate Field Constraint)

The match of this element is the result of applying the given callable to the fact-extracted value. If the callable returns *True* the FC will match, in other case the FC will not match.

Listing 1.12: Match if some fact is declared whose first parameter is an instance of int

```
@Rule(Fact(P(lambda x: isinstance(x, int))))
def _():
    pass
```

## Composing FCs: *&*, */* and *~*

All FC can be composed together using the composition operators &, | and ~.

### *ANDFC()* a.k.a. *&*

The composed FC matches if all the given FC match.

Listing 1.13: Match if key *x* of *Point* is a value between 0 and 255.

```
@Rule(Fact(x=P(lambda x: x >= 0) & P(lambda x: x <= 255)))
def _():
    pass
```

### *ORFC()* a.k.a. */*

The composed FC matches if any of the given FC matches.

Listing 1.14: Match if *name* is either *Alice* or *Bob*.

```
@Rule(Fact(name=L('Alice') | L('Bob')))
def _():
    pass
```

### *NOTFC()* a.k.a. ~

This composed FC negates the given FC, reversing the logic. If the given FC matches this will not and vice versa.

Listing 1.15: Match if *name* is not *Charlie*.

```
@Rule(Fact(name=~L('Charlie')))
def _():
    pass
```

## Variable Binding: The << Operator

Any patterns and some FCs can be binded to a name using the << operator.

Listing 1.16: The first value of the matching fact will be binded to the name *value* and passed to the function when fired.

```
@Rule(Fact('value' << W()))
def _(value):
    pass
```

Listing 1.17: The whole matching fact will be binded to *f1* and passed to the function when fired.

```
@Rule('f1' << Fact())
def _(f1):
    pass
```

## Cookbook

CHAPTER 2

# API Documentation

## Modules documentation

### pyknow

### pyknow.abstract

**class** `pyknow.abstract.`**`Matcher`**(*engine*)

Bases: `object`

**`changes`**(*adding=None*, *deleting=None*)

Main interface with the matcher.

Called by the knowledge engine when changes are made in the working memory and return a set of activations.

**`reset`**()

Reset the matcher memory.

**class** `pyknow.abstract.`**`Strategy`**(**args*, ***kwargs*)

Bases: `object`

**`update_agenda`**(*agenda*, *added*, *removed*)

### pyknow.activation

Activations represent rules that matches against a specific factlist.

**class** `pyknow.activation.`**`Activation`**(*rule*, *facts*, *context=None*)

Bases: `object`

Activation object

## pyknow.agenda

**class** `pyknow.agenda.`**`Agenda`**

Bases: `object`

Collection of activations that handles its execution state.

**Note:** Extracted from clips documentation: `The agenda is a collection of activations which are those rules which match pattern entities`

**`get_next`**`()`

Returns the next activation, removes it from activations list.

## pyknow.conditionalelement

**class** `pyknow.conditionalelement.`**`AND`**

Bases: `pyknow.conditionalelement.OperableCE`, `pyknow.conditionalelement.ConditionalElement`

**class** `pyknow.conditionalelement.`**`OR`**

Bases: `pyknow.conditionalelement.OperableCE`, `pyknow.conditionalelement.ConditionalElement`

**class** `pyknow.conditionalelement.`**`NOT`**

Bases: `pyknow.conditionalelement.OperableCE`, `pyknow.conditionalelement.ConditionalElement`

**class** `pyknow.conditionalelement.`**`TEST`**

Bases: `pyknow.conditionalelement.OperableCE`, `pyknow.conditionalelement.ConditionalElement`

**class** `pyknow.conditionalelement.`**`EXISTS`**

Bases: `pyknow.conditionalelement.OperableCE`, `pyknow.conditionalelement.ConditionalElement`

**class** `pyknow.conditionalelement.`**`FORALL`**

Bases: `pyknow.conditionalelement.OperableCE`, `pyknow.conditionalelement.ConditionalElement`

## pyknow.engine

`pyknow engine` represents `CLIPS modules`

**class** `pyknow.engine.`**`KnowledgeEngine`**

Bases: `object`

This represents a clips' `module`, wich is an `inference engine` holding a set of `rules` (as *`pyknow.rule.Rule`* objects), an `agenda` (as *`pyknow.agenda.Agenda`* object) and a `fact-list` (as *`pyknow.factlist.FactList`* objects)

This could be considered, when inherited from, as the `knowlege-base`.

**`declare`**`(`**facts*`)`

Declare from inside a fact, equivalent to `assert` in clips.

**Note:** This updates the agenda.

**duplicate**(*template_fact*, ***modifiers*)
Create a new fact from an existing one.

**get_activations**()
Return activations

**get_deffacts**()
Return the existing deffacts sorted by the internal order

**get_rules**()
Return the existing rules.

**halt**()

**modify**(*declared_fact*, ***modifiers*)
Modifies a fact.

Facts are inmutable in Clips, thus, as documented in clips reference manual, this retracts a fact and then re-declares it

*modifiers* must be a Mapping object containing keys and values to be changed.

To allow modifying positional facts, the user can pass a string containing the symbol "_" followed by the numeric index (starting with 0). Ex:

```
>>> ke.modify(my_fact, _0="hello", _1="world", other_key="!")
```

**reset**()
Performs a reset as per CLIPS behaviour (resets the agenda and factlist and declares InitialFact())

**Note:** If persistent facts have been added, they'll be re-declared.

**retract**(*idx_or_declared_fact*)
Retracts a specific fact, using its index

**Note:** This updates the agenda

**run**(*steps=inf*)
Execute agenda activations

## pyknow.factlist

`fact-list` implementation from CLIPS.

See Making a List section on the user guide. Also see retrieving the fact-list on the clips programming manual

**class** `pyknow.factlist.`**FactList**
Bases: `collections.OrderedDict`

Contains a list of facts (`asserted` data).

In clips, there is the concept of "modules" (*pyknow.engine.KnowledgeEngine*), wich have their own *pyknow.factlist.FactList* and *pyknow.agenda.Agenda*

A factlist acts as both the module's factlist and a `fact-set` yet currently most methods from a `fact-set` are not yet implemented

**`changes`**
Return a tuple with the removed and added facts since last run.

**`declare`** (*fact*)
Assert (in clips terminology) a fact.

This keeps insertion order.

> **Warning:** This will reject any object that not descend from the Fact class.

**Parameters** **`fact`** – The fact to declare, **must** be derived from *`pyknow.fact.Fact`*.

**Returns** (int) The index of the fact in the list.

**Throws ValueError** If the fact providen is not a Fact object

**`retract`** (*idx_or_fact*)
Retract a previously asserted fact.

See "Retract that fact" in Clips User Guide.

**Parameters** **`idx`** – The index of the fact to retract in the factlist

**Returns** (int) The retracted fact's index

**Throws IndexError** If the fact's index providen does not exist

## pyknow.fact

**class** `pyknow.fact.`**`Fact`** (**args*, ***kwargs*)
Bases: `pyknow.conditionalelement.OperableCE`, `pyknow.pattern.Bindable`, `dict`

Base Fact class

**`copy`** ()

**classmethod** **`from_iter`** (*pairs*)

**`has_field_constraints`** ()

**static** **`is_special`** (*key*)

**`update`** (*mapping*)

**class** `pyknow.fact.`**`InitialFact`** (**args*, ***kwargs*)
Bases: *`pyknow.fact.Fact`*

## pyknow.fieldconstraint

**class** `pyknow.fieldconstraint.`**`L`**
Bases: `pyknow.pattern.Bindable`, `pyknow.fieldconstraint.FieldConstraint`

Literal Field Constraint

**`value`**

**class** `pyknow.fieldconstraint.`**`W`**

Bases: `pyknow.pattern.Bindable`, `pyknow.fieldconstraint.FieldConstraint`

Wildcard Field Constraint

**class** `pyknow.fieldconstraint.`**`P`**

Bases: `pyknow.pattern.Bindable`, `pyknow.fieldconstraint.FieldConstraint`

Predicate Field Constraint

**`match`**

## pyknow.rule

**class** `pyknow.rule.`**`Rule`**

Bases: `pyknow.conditionalelement.ConditionalElement`

Base `CE`, all `CE` are to derive from this class.

This class is used as a decorator, thus provoking __call__ to be called twice:

1. The first call is when the decorator is been created. At this point we assign the function decorated to `self._wrapped` and return `self` to be called the second time.
2. The second call is to execute the decorated function, se we pass all the arguments along.

**`new_conditions`**(**args*)

Generate a new rule with the same attributes but with the given conditions.

## pyknow.strategies

**class** `pyknow.strategies.`**`DepthStrategy`**(**args*, ***kwargs*)

Bases: *`pyknow.abstract.Strategy`*

`pyknow.strategies.`**`listdict`**()

Defaultdict of a list

## pyknow.watchers

Watchers are loggers that log detailed information on CLIPS, disabled by default and that can be enabled by the *(watch)* method.

Here, we expose a rule, fact and agenda watchers as well as a method to enable/disable them both individually and all of them.

`pyknow.watchers.`**`watch`**(**what*, *level=10*)

Enable watchers.

Defaults to enable all watchers, accepts a list names of watchers to enable.

`pyknow.watchers.`**`unwatch`**(**what*)

Disable watchers.

Defaults to enable all watchers, accepts a list names of watchers to enable.

## pyknow.matchers

### pyknow.matchers.rete

RETE algorithm implementation.

This is implemented as described by Charles L. Forgy in his original Ph.D thesis paper. With minor changes to allow CLIPS like matching and a more pythonic approach.

**class** `pyknow.matchers.rete.`**`ReteMatcher`**`(`**args*, ***kwargs*`)`
Bases: *`pyknow.abstract.Matcher`*

RETE algorithm with *pyknow* matcher interface.

**static** **`build_alpha_part`**`(`*ruleset*, *root_node*`)`
Given a set of already adapted rules, build the alpha part of the RETE network starting at *root_node*.

**static** **`build_beta_part`**`(`*ruleset*, *alpha_terminals*`)`
Given a set of already adapted rules, and a dictionary of patterns and alpha_nodes, wire up the beta part of the RETE network.

**`build_network`**`()`

**`changes`**`(`*adding=None*, *deleting=None*`)`
Pass the given changes to the root_node.

**static** **`prepare_ruleset`**`(`*engine*`)`
Given a *KnowledgeEngine*, generate a set of rules suitable for RETE network generation.

**`print_network`**`()`
Generate a graphviz compatible graph.

**`reset`**`()`

**`show_network`**`()`

### pyknow.matchers.rete.abstract

Abstract base classes for the RETE implementation.

**class** `pyknow.matchers.rete.abstract.`**`Check`**
Bases: `object`

**class** `pyknow.matchers.rete.abstract.`**`Node`**
Bases: `object`

Node interface.

**`add_child`**`(`*child*, *callback*`)`
Add a child to *self.children* if necessary.

**`reset`**`()`
Reset itself and recursively all its children.

**class** `pyknow.matchers.rete.abstract.`**`OneInputNode`**
Bases: *`pyknow.matchers.rete.abstract.Node`*

Nodes which only have one input port.

**`activate`**`(`*token*`)`
Make a copy of the received token and call *self._activate*.

**class** `pyknow.matchers.rete.abstract.`**`TwoInputNode`**
Bases: *`pyknow.matchers.rete.abstract.Node`*

Nodes which have two input ports: left and right.

**`activate_left`**(*token*)
Make a copy of the received token and call *_activate_left*.

**`activate_right`**(*token*)
Make a copy of the received token and call *_activate_right*.

## pyknow.matchers.rete.check

**class** `pyknow.matchers.rete.check.`**`CheckFunction`**(*key_a*, *key_b*, *expected*, *check*)
Bases: `tuple`

**`check`**
Alias for field number 3

**`expected`**
Alias for field number 2

**`key_a`**
Alias for field number 0

**`key_b`**
Alias for field number 1

**class** `pyknow.matchers.rete.check.`**`FactCapture`**
Bases: *`pyknow.matchers.rete.abstract.Check`*, `pyknow.matchers.rete.check._FactCapture`

**class** `pyknow.matchers.rete.check.`**`FeatureCheck`**
Bases: *`pyknow.matchers.rete.abstract.Check`*, `pyknow.matchers.rete.check._FeatureCheck`

**`get_check_function`**()
staticmethod(function) -> method

Convert a function to be a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

class C: def f(arg1, arg2, ...): ... f = staticmethod(f)

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. For a more advanced concept, see the classmethod builtin.

**class** `pyknow.matchers.rete.check.`**`SameContextCheck`**
Bases: *`pyknow.matchers.rete.abstract.Check`*

**class** `pyknow.matchers.rete.check.`**`TypeCheck`**
Bases: *`pyknow.matchers.rete.abstract.Check`*, `pyknow.matchers.rete.check._TypeCheck`

**class** `pyknow.matchers.rete.check.`**`WhereCheck`**
Bases: *`pyknow.matchers.rete.abstract.Check`*, `pyknow.matchers.rete.check._WhereCheck`

## pyknow.matchers.rete.dnf

Rewrite engine to get disjuntive normal form of the rules

`pyknow.matchers.rete.dnf.`**`dnf`**`(`*exp*`)`

`pyknow.matchers.rete.dnf.`**`unpack_exp`**`(`*exp*, *op*`)`

## pyknow.matchers.rete.mixins

Mixing classes for the RETE nodes.

**class** `pyknow.matchers.rete.mixins.`**`AnyChild`**

Bases: `object`

This node allow any kind of node as a child.

**`add_child`**`(`*node*, *callback*`)`

Add node and callback to the children set.

**class** `pyknow.matchers.rete.mixins.`**`ChildNode`**`(`*node*, *callback*`)`

Bases: `tuple`

Used to store node/callback pair in nodes children set.

**`callback`**

Alias for field number 1

**`node`**

Alias for field number 0

**class** `pyknow.matchers.rete.mixins.`**`HasMatcher`**`(`*matcher*`)`

Bases: `object`

This node need a match callable as parameter.

**class** `pyknow.matchers.rete.mixins.`**`NoMemory`**

Bases: `object`

The node has no memory so we have nothing to do.

## pyknow.matchers.rete.nodes

RETE nodes implementation.

This are the node types needed by this RETE implementation. Some node types (like ‘The One-input Node for Testing Variable Bindings) are not needed in this implementation.

**class** `pyknow.matchers.rete.nodes.`**`BusNode`**

Bases: *`pyknow.matchers.rete.mixins.AnyChild`*, *`pyknow.matchers.rete.mixins.NoMemory`*, *`pyknow.matchers.rete.abstract.Node`*

The Bus Node.

The node that reports working memory changes to the rest of the network.

This node cannot be activated in the same manner as the other nodes. No tokens can be sent to it since this is the node where the first tokens are built.

**`add`**`(`*fact*`)`

Create a VALID token and send it to all children.

**remove** (*fact*)
Create an INVALID token and send it to all children.

**class** pyknow.matchers.rete.nodes.**ConflictSetNode** (*rule*)
Bases: *pyknow.matchers.rete.mixins.AnyChild*, *pyknow.matchers.rete.abstract.OneInputNode*

Conflict Set Change Node.

This node is the final step in the network. Any token activating this node will produce an activation (VALID token) or deactivation (INVALID token) of the internal *rule* with the token context and facts.

**get_activations** ()
Return a list of activations.

**class** pyknow.matchers.rete.nodes.**FeatureTesterNode** (*matcher*)
Bases: *pyknow.matchers.rete.mixins.AnyChild*, *pyknow.matchers.rete.mixins.HasMatcher*, *pyknow.matchers.rete.mixins.NoMemory*, *pyknow.matchers.rete.abstract.OneInputNode*

Feature Tester Node.

This node implementation represents two different nodes in the original paper: *The One-input Node for Testing Constant Features* and *The One-input Node for Testing Variable Bindings*.

The trick here is this node receives a callable object at initilization time and uses it for testing the received tokens on activation. The given callable can return one of the following things:

- Boolean:
  - *True*: The test pass. The token will be sent to the children nodes.
  - *False*: The test failed. Do nothing.
- Mapping (dict):
  - With content: The test pass. In addition the pairs key-value will be added to the token context.
  - Empty: The test failed. Do nothing.

The only exception here is when the callable returns a mapping with some key and some value, and the current context of the token also have an entry for this key but with a different value. In this case the test do not pass.

**class** pyknow.matchers.rete.nodes.**NotNode** (*matcher*)
Bases: *pyknow.matchers.rete.mixins.AnyChild*, *pyknow.matchers.rete.mixins.HasMatcher*, *pyknow.matchers.rete.abstract.TwoInputNode*

Not Node.

This is a special kind of node representing the absence of some fact/condition.

This node is similar to *OrdinaryMatchNode* in the sense it has two input ports and try to match tokens arriving in both of them. But pass VALID tokens to the children when no matches are found and INVALID tokens when they are.

**class** pyknow.matchers.rete.nodes.**OrdinaryMatchNode** (*matcher*)
Bases: *pyknow.matchers.rete.mixins.AnyChild*, *pyknow.matchers.rete.mixins.HasMatcher*, *pyknow.matchers.rete.abstract.TwoInputNode*

Ordinary Two-input Node.

This kind of node receive tokens at two ports (left and right) and try to match them.

The matching function is a callable given as a parameter to __init__ and stored internally. This functions will receive two contexts, one from the left and other from the right, and decides if they match together (returning True or False).

Matching pairs will be combined in one token containing facts from both and a combined context. This combined tokens will be sent to all children.

**class** `pyknow.matchers.rete.nodes.`**`WhereNode`**(*matcher*)

Bases: *`pyknow.matchers.rete.mixins.AnyChild`*, *`pyknow.matchers.rete.mixins.HasMatcher`*, *`pyknow.matchers.rete.mixins.NoMemory`*, *`pyknow.matchers.rete.abstract.OneInputNode`*

Check some conditions over a token context.

## pyknow.matchers.rete.token

Token object and related objects needed by the RETE algorithm.

**class** `pyknow.matchers.rete.token.`**`Token`**

Bases: `pyknow.matchers.rete.token._Token`

Token, as described by RETE but with context.

**class** **`TagType`**

Bases: `enum.Enum`

Types of Token TAG data.

**`INVALID`** **= False**

**`VALID`** **= True**

`Token.`**`copy`**()

Make a new instance of this Token.

This method makes a copy of the mutable part of the token before making the instance.

**classmethod** `Token.`**`invalid`**(*data*, *context=None*)

Shortcut to create an INVALID Token.

`Token.`**`is_valid`**()

Test if this Token is VALID.

`Token.`**`to_info`**()

Create and return an instance of TokenInfo with this token.

This is useful, for example, to use this token information as a dictionary key.

**classmethod** `Token.`**`valid`**(*data*, *context=None*)

Shortcut to create a VALID Token.

**class** `pyknow.matchers.rete.token.`**`TokenInfo`**

Bases: `pyknow.matchers.rete.token._TokenInfo`

Tag agnostig version of Token with inmutable data.

**`to_invalid_token`**()

Create an INVALID token using this data.

**`to_valid_token`**()

Create a VALID token using this data.

## pyknow.matchers.rete.utils

`pyknow.matchers.rete.utils.`**`extract_facts`**`(`*rule*`)`
Given a rule, return a set containing all rule LHS facts.

`pyknow.matchers.rete.utils.`**`generate_checks`**`(`*fact*`)`
Given a fact, generate a list of Check objects for checking it.

`pyknow.matchers.rete.utils.`**`prepare_rule`**`(`*exp*`)`
Given a rule, build a new one suitable for RETE network generation.

Meaning:

1. Rule is in disjuntive normal form (DNF).
2. If the *rule* is empty is filled with an *InitialFact*.
3. If the *rule* starts with a *NOT*, an *InitialFact* is prepended.
4. If any AND starts with a *NOT*, an *InitialFact* is prepended.

#. If the *rule* is an OR condition, each NOT inside will be converted to AND(InitialFact(), NOT(...))

`pyknow.matchers.rete.utils.`**`wire_rule`**`(`*rule*, *alpha_terminals*, *lhs=None*`)`

# CHAPTER 3

# Release History

## 1.1.1

- Removing the borg optimization for P field constraints.
- Use the hash of the check in the sorting of the nodes to always generate the same alpha part of the network.

## 1.1.0

- Allow any kind of callable in Predicate Field Constraints (P()).

## 1.0.1

- DNF of OR clause inside AND or Rule was implemented wrong.

## 1.0.0

- RETE matching algorithm.
- Better Rule decorator system.
- Facts are dictionaries.
- Documentation.

## <1.0.0

- Unestable API.

- Wrong matching algorithm.
- Bad performance
- PLEASE DON'T USE THIS.

CHAPTER 4

# Source documentation

- genindex
- modindex
- search

# Python Module Index

# Index