# Contents

## 1 User Guide

1.1 Introduction .................................................................................................................. 3
  1.1.1 Philosophy ................................................................................................................. 3
  1.1.2 Features ..................................................................................................................... 3
  1.1.3 Difference between CLIPS and PyKnow ................................................................. 3

1.2 Installation ....................................................................................................................... 3
  1.2.1 From PyPI .................................................................................................................. 3
  1.2.2 Getting the source code ............................................................................................ 4

1.3 The Basics ......................................................................................................................... 4
  1.3.1 Facts .......................................................................................................................... 4
  1.3.2 Rules .......................................................................................................................... 5
  1.3.3 DefFacts ..................................................................................................................... 6
  1.3.4 KnowledgeEngine ..................................................................................................... 7

1.4 Reference ........................................................................................................................ 9
  1.4.1 Rule ............................................................................................................................ 11
  1.4.2 Conditional Elements: Composing Patterns Together ................................................. 13
  1.4.3 Field Constraints: FC for sort .................................................................................... 15
  1.4.4 Composing FCs: &, | and ~ ....................................................................................... 15
  1.4.5 Variable Binding: The << Operator ............................................................................ 16
  1.4.6 MATCH object ........................................................................................................... 17
  1.4.7 AS object ................................................................................................................... 17
  1.4.8 Nested matching ....................................................................................................... 17
  1.4.9 Mutable objects ....................................................................................................... 18

1.5 Cookbook ........................................................................................................................ 19

## 2 API Documentation

2.1 Modules documentation .................................................................................................. 21
  2.1.1 pyknow ...................................................................................................................... 21
  2.1.2 pyknow.abstract ...................................................................................................... 21
  2.1.3 pyknow.activation ................................................................................................... 21
  2.1.4 pyknow.agenda ........................................................................................................ 22
  2.1.5 pyknow.conditionalelement ..................................................................................... 22
  2.1.6 pyknow.engine .......................................................................................................... 22
  2.1.7 pyknow.factlist ......................................................................................................... 23
  2.1.8 pyknow.fact .............................................................................................................. 24
  2.1.9 pyknow.fieldconstraint ............................................................................................ 25
3 Release History
3.1 1.7.0 ......................................................... 33
3.2 1.6.0 ......................................................... 33
3.3 1.5.0 ......................................................... 33
3.4 1.4.0 ......................................................... 33
3.5 1.3.0 ......................................................... 34
3.6 1.2.0 ......................................................... 34
3.7 1.1.1 ......................................................... 34
3.8 1.1.0 ......................................................... 34
3.9 1.0.1 ......................................................... 34
3.10 1.0.0 ....................................................... 34
3.11 <1.0.0 ....................................................... 35

4 Source documentation

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

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

```bash
>>> 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.
1.1 Introduction

1.1.1 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.

1.1.2 Features

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

1.1.3 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.

1.2 Installation

1.2.1 From PyPI

To install PyKnow, run this command in your terminal:
1.2.2 Getting the source code

PyKnow is developed on Github.
You can clone the repository using the git command:

```bash
$ 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:

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

1.3 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.

1.3.1 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.

```python
>>> f = Fact(a=1, b=2)
```

```python
>>> f['a']
1
```

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

```python
>>> Fact(a=1, b=2)  # Order is arbitrary :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.

```python
>>> f = Fact('x', 'y', 'z')
```

```python
>>> f[0]
'x'
```

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

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

```python
>>> f[1]
'y'
```

```python
>>> f['b']
2
```

5. You can subclass Fact to express different kinds of data or extend it with your custom functionality.
```python
class Alert(Fact):
    """The alert level.""
    pass

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

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

```python
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)
```

6. *Fact* fields can be validated automatically for you if you define them using *Field*. *Field* uses the *Schema* library internally for data validation. Also, a field can be declared *mandatory* or have a *default*.

### 1.3.2 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.

```python
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'
    """
```
You can use logic operators to express complex LHS conditions.

```python
@Rule(
    AND(
        OR(User('admin'),
        User('root')),
        NOT(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.

**Note:** For a list of more complex operators you can check the *pyknow.operator* module.

### 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.

```python
>>> 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 "../pyknow/engine.py", line 210, in declare
    self.__declare(*facts)
  File "../pyknow/engine.py", line 191, in __declare
    "Declared facts cannot contain conditional elements"
TypeError: Declared facts cannot contain conditional elements
```

### 1.3.3 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.

```python
@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.

New in version 1.7.0: The `reset()` method accepts any number of keyword parameters whose gets passed to `DefFacts` decorated methods if those methods present the same parameters.

### 1.3.4 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: 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=MATCH.name),
          Fact(location=MATCH.location))
    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!
```

```bash
$ 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).

```python
>>> 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 2:** Both, the index and the fact can be used with retract

```python
>>> 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.

```python
>>> 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-0> Fact(color='red')
<f-2> Fact(color='yellow', blink=True)

Engine execution procedure

This is the usual process to execute a KnowledgeEngine.

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.

Cycle of execution

In a conventional programming style, the starting point, the stopping point, and the sequence of operations are defined explicitly by the programmer. With PyKnow, the program flow does not need to be defined quite so explicitly. The knowledge (Rules) and the data (Facts) are separated, and the KnowledgeEngine is used to apply the knowledge to the data.

The basic execution cycle is as follows:

1. If the rule firing limit has been reached the execution is halted.
2. The top rule on the agenda is selected for execution. If there are no rules on the agenda, the execution is halted.
3. The RHS actions of the selected rule are executed (the method is called). As a result, rules may be activated or deactivated. Activated rules (those rules whose conditions are currently satisfied) are placed on the agenda. The placement on the agenda is determined by the salience of the rule and the current conflict resolution strategy. Deactivated rules are removed from the agenda.

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.

1.4 Reference

The following diagram shows all the system components and the relationships among them.
1.4.1 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.

```python
@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 3: r1 has precedence over r2

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

(continues on next page)
1.4.2 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 4: Match if two facts are declared, one matching Fact(1) and other matching Fact(2)

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

**OR**

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

Listing 5: Match if a fact matching Fact(1) exists and/or a fact matching Fact(2) exists

```python
@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 6: Match if no fact match with Fact(1)

```python
@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 7: Match for all numbers \(a, b, c\) where \(a > b > c\)

```python
@Rule(Number(MATCH.a),
      Number(MATCH.b),
      TEST(lambda a, b: a > b),
      Number(MATCH.c),
      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 8: Match once when one or more Color exists

```python
@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 occurrence of another specified CE.

Listing 9: Match when for every Student fact there is a Reading, Writing and Arithmetic fact with the same name.

```python
@Rule(FORALL(Student(MATCH.name),
            Reading(MATCH.name),
            Writing(MATCH.name),
            Arithmetic(MATCH.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.

1.4.3 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 10: Match if the first element is exactly 3

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

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

W (Wildcard Field Constraint)

This element matches with any value.

Listing 11: Match if some fact is declared with the key mykey.

```python
@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 12: Match if some fact is declared whose first parameter is an instance of int

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

1.4.4 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 13: Match if key $x$ of Point is a value between 0 and 255.

```python
@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 14: Match if name is either Alice or Bob.

```python
@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 15: Match if name is not Charlie.

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

1.4.5 Variable Binding: The $\ll$ Operator

Any pattern and some FCs can be binded to a name using the $\ll$ operator.

Listing 16: The first value of the matching fact will be binded to the name value and passed to the function when fired.

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

Deprecated since version 1.2.0: Use MATCH object instead.

Listing 17: The whole matching fact will be binded to $f1$ and passed to the function when fired.

```python
@Rule('f1' << Fact())
def _f1():
    pass
```

Deprecated since version 1.2.0: Use AS object instead.
1.4.6 MATCH object

The MATCH objects helps generating more readable name bindings. It is syntactic sugar for a Wildcard Field Constraint binded to a name. For example:

```python
@Rule(Fact(MATCH.myvalue))
def _(myvalue):
    pass
```

Is exactly the same as:

```python
@Rule(Fact("myvalue" << W()))
def _(myvalue):
    pass
```

1.4.7 AS object

The AS object like the MATCH object is syntactic sugar for generating bindable names. In this case any attribute requested to the AS object will return a string with the same name.

```python
@Rule(AS.myfact << Fact(W()))
def _(myfact):
    pass
```

Is exactly the same as:

```python
@Rule("myfact" << Fact(W()))
def _(myfact):
    pass
```

Warning: This behavior will vary in future releases of PyKnow and the string flavour of the operator may disappear.

1.4.8 Nested matching

New in version 1.3.0.

Nested matching is useful to match against Fact values which contains nested structures like dicts or lists.

```python
>>> Fact(name="scissors", against={"scissors": 0, "rock": -1, "paper": 1})
>>> Fact(name="paper", against={"scissors": -1, "rock": 1, "paper": 0})
>>> Fact(name="rock", against={"scissors": 1, "rock": 0, "paper": -1})
```

Nested matching take the form field__subkey=value. (That’s a double-underscore). For example:

```python
>>> @Rule(Fact(name=MATCH.name, against__scissors=1, against__paper=-1))
... def what_wins_to_scissors_and_losses_to_paper(self, name):
...     print(name)
```

Is possible to match against an arbitrary deep structure following the same method.
In this example we can check for collision between a ship and its parent with the following rule:

```python
>>> @Rule(Ship(data__name=MATCH.name1,
... data__position__x=MATCH.x,
... data__position__y=MATCH.y,
... data__parent__name=MATCH.name2,
... data__parent__position__x=MATCH.x,
... data__parent__position__y=MATCH.y))
... def collision_detected(self, name1, name2, **_):
...     print("COLLISION!", name1, name2)
```

If the nested data structure contains list, tuples or any other sequence you can use numeric indexes as needed.

```python
>>> Ship(data={
...     "name": "SmallShip",
...     "position": {
...         "x": 300,
...         "y": 200},
...     "parent": {
...         "name": "BigShip",
...         "position": {
...             "x": 150,
...             "y": 300}}})
>>> Ship(data={
...     "name": "SmallShip",
...     "position": {
...         "x": 300,
...         "y": 200},
...     "enemies": [
...         {"name": "Destroyer"},
...         {"name": "BigShip"}]
... })
>>> @Rule(Ship(data__enemies__0__name="Destroyer"))
... def next_enemy_is_destroyer(self):
...     print("Bye bye!")
```

## 1.4.9 Mutable objects

PyKnow’s matching algorithm depends on the values of the declared facts being immutable. When a `Fact` is created, all its values are transformed to an immutable type if they are not. For this matter the method `pyknow.utils.freeze` is used internally.

```python
>>> class MutableTest(KnowledgeEngine):
...     @Rule(Fact(v1=MATCH.v1, v2=MATCH.v2, v3=MATCH.v3))
...     def is_immutable(self, v1, v2, v3):
...         print(type(v1), "is Immutable!")
...         print(type(v2), "is Immutable!")
...         print(type(v3), "is Immutable!")
... >>> ke = MutableTest()
... >>> ke.reset()```
>>> ke.declare(Fact(v1={"a": 1, "b": 2}, v2=[1, 2, 3], v3={1, 2, 3}))
>>> ke.run()
frozendict is Immutable
frozenlist is Immutable
frozenset is Immutable

Note: You can import frozendict and frozenlist from pyknow.utils module. However frozenset is a Python built-in type.

Register your own mutable freezer

If you need to include your own custom mutable types as fact values you have to register a specialized type freezer for your custom type.

>>> from pyknow.utils import freeze
>>> @freeze.register(MyType)
... def freeze_mytype(obj):
...     return ...

# My frozen version of my type

Unfreeze frozen objects

To easily unfreeze the frozen objects pyknow.utils contains an unfreeze method.

>>> class MutableTest(KnowledgeEngine):
...     @Rule(Fact(v1=MATCH.v1, v2=MATCH.v2, v3=MATCH.v3))
...     def is_immutable(self, v1, v2, v3):
...         print(type(unfreeze(v1)), "is Mutable!")
...         print(type(unfreeze(v2)), "is Mutable!")
...         print(type(unfreeze(v3)), "is Mutable!")

>>> ke = MutableTest()
>>> ke.reset()
>>> ke.declare(Fact(v1={"a": 1, "b": 2}, v2=[1, 2, 3], v3={1, 2, 3}))
>>> ke.run()
dict is Mutable
list is Mutable
gset is Mutable

Note: The same freeze registration procedure shown above also applies to unfreeze.

1.5 Cookbook
2.1 Modules documentation

2.1.1 pyknow

2.1.2 pyknow.abstract

```python
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.
```

```python
class pyknow.abstract.Strategy(*args, **kwargs):
    Bases: object
    update_agenda(agenda, added, removed)
```

2.1.3 pyknow.activation

Activations represent rules that matches against a specific factlist.

```python
class pyknow.activation.Activation(rule, facts, context=None):
    Bases: object
    Activation object
```
2.1.4 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.

2.1.5 pyknow.conditionalelement

class pyknow.conditionalelement.AND

class pyknow.conditionalelement.OR

class pyknow.conditionalelement.NOT

class pyknow.conditionalelement.TEST

class pyknow.conditionalelement.EXISTS

class pyknow.conditionalelement.FORALL

2.1.6 pyknow.engine

   pyknow engine represents CLIPS modules

class pyknow.engine.KnowledgeEngine
   Bases: object

   This represents a clips' module, which 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 knowledge-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 immutable 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:

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

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

Any keyword argument passed to reset will be passed to @DefFacts which have those arguments on their
signature.

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

2.1.7 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), which 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 provided 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 provided does not exist

### 2.1.8 pyknow.fact

**class** pyknow.fact.BaseField
Bases: object
- **validate**(data)
  Raise an exception on invalid data.

**class** pyknow.fact.Fact(*args, **kwargs)
Bases: pyknow.conditionalelement.OperableCE, pyknow.patternBindable, dict
Base Fact class
- **as_dict**()
  Return a dictionary containing this Fact data.
- **copy**()
  Return a copy of this Fact.
- **classmethod** from_iter(pairs)
- **has_field_constraints**()
- **has_nested_accessor**()
- **static** is_special(key)
update \([E], **F) \rightarrow \text{None}. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

validate ()

class pyknow.fact.Field (schema_definition, mandatory=False, default=<object object>)
Bases: pyknow.fact.BaseField

NODEFAULT = <object object>

validate (data)
Raise an exception on invalid data.

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

class pyknow.fact.Validable
Bases: type

2.1.9 pyknow.fieldconstraint

class pyknow.fieldconstraint.L
Bases: pyknow.patternBindable.pyknow.fieldconstraint.FieldConstraint

Literal Field Constraint

value

class pyknow.fieldconstraint.W
Bases: pyknow.patternBindable.pyknow.fieldconstraint.FieldConstraint

Wildcard Field Constraint

class pyknow.fieldconstraint.P
Bases: pyknow.patternBindable.pyknow.fieldconstraint.FieldConstraint

Predicate Field Constraint

match

2.1.10 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, so we pass all the arguments along.

new_conditions (*args)
Generate a new rule with the same attributes but with the given conditions.
2.1.11 pyknow.strategies

class pyknow.strategies.DepthStrategy(*args, **kwargs)
    Bases: pyknow.abstract.Strategy
    get_key

2.1.12 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.

2.1.13 pyknow.matchers

2.1.14 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()
        Reset the matcher memory.
2.1.15 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.

2.1.16 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

2.1.17 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)

2.1.18 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.

2.1.19 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
    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)
    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)
    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 initialization 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.
• Mapping (dict):
  – With content: The test pass. In addition the pairs key-value will be added to the token context.

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)

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.OldinaryMatchNode(matcher)

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)

Check some conditions over a token context.

2.1.20 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

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 invalid**(data, context=None)
Shortcut to create an INVALID Token.

**is_valid()**
Test if this Token is VALID.

**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 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.

---

### 2.1.21 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.
5. 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)

---

### 2.1.22 pyknow.operator

The operator module contains a set of predicate functions constructors based on the P() field constraint.
These operators can be composed together and binded like normal Field Constraints.
Example:
>>> WIDTH = 640
>>> HEIGHT = 480
>>> 
>>> class Player(Fact):
...     pass
...
>>> @Rule(
...     Player(
...         x=MATCH.x & GE(0) & LE(WIDTH),
...         y=MATCH.y & BETWEEN(0, HEIGHT),
...         name=MATCH.name & (CALL.startswith("@") | CALL.endswith("_ADM")))
...     )
...     def admin_in_invisible_area(self, x, y, name):
...         pass
...

pyknow.operator.TRUTH = P(<class 'bool'>,)
    Return True if obj is true, and False otherwise. This is equivalent to using the bool constructor.

pyknow.operator.LT(b)
    Less than operator.

pyknow.operator.LE(b)
    Less than or equal operator.

pyknow.operator.EQ(b)
    Equal operator.

pyknow.operator.NE(b)
    Not equal operator.

pyknow.operator.GE(b)
    Greater than or equal operator.

pyknow.operator.GT(b)
    Greater than operator.

pyknow.operator.IS(b)
    Tests object identity.

pyknow.operator.IS_NOT(b)
    Tests object identity.

pyknow.operator.CONTAINS(b)
    Return the outcome of the test b in a

pyknow.operator.BETWEEN(a, b)
    The BETWEEN operator selects values within a given range. The BETWEEN operator is inclusive: begin and end values are included.

pyknow.operator.REGEX(pattern, flags=0)
    Regular expression matching.

pyknow.operator.LIKE(pattern)
    Unix shell-style wildcards. Case-sensitive

pyknow.operator.ILIKE(pattern)
    Unix shell-style wildcards. Case-insensitive
CHAPTER 3

Release History

3.1 1.7.0

- Implemented the template system.
- Replaced warnings by watchers messages.
- Fixed freeze() with frozen objects.
- Fixed unfreeze() with unfrozen objects.
- Parametrized DefFacts via reset() kwargs.

3.2 1.6.0

- Improved overall performance.

3.3 1.5.0

- Added Python version 3.7 to tox.ini.
- Monkey and bananas example.
- Fixed bug, numeric index args gets repeated in a weird way introduced in 1.4.0.
- Pass only the defined args in absence of kwargs.

3.4 1.4.0

- Order integer facts keys after making a copy.
- as_dict method for Fact.
- freeze and unfreeze method documentation.
- unfreeze method in pyknow.utils.
- Zebra example from Clips.

### 3.5 1.3.0

- pyknow.operator module.
- Nested matching.
- Added Talk ‘Sistemas Expertos en Python con PyKnow - PyConES 2017’ to docs folder.

### 3.6 1.2.0

- Freeze fact values as the default behavior to address Issue #9.
- Added pyknow.utils.anyof to mitigate Issue #7.
- Raise RuntimeError if a fact value is modified after declare().
- Added MATCH and AS objects.

### 3.7 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.

### 3.8 1.1.0

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

### 3.9 1.0.1

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

### 3.10 1.0.0

- RETE matching algorithm.
- Better Rule decorator system.
- Facts are dictionaries.
- Documentation.
3.11 <1.0.0

- Unstable API.
- Wrong matching algorithm.
- Bad performance
- PLEASE DON’T USE THIS.
CHAPTER 4

Source documentation

• genindex
• modindex
• search
p
pyknow, 21
pyknow.abstract, 21
pyknow.activation, 21
pyknow.agenda, 22
pyknow.conditionalelement, 22
pyknow.engine, 22
pyknow.fact, 24
pyknow.factlist, 23
pyknow.fieldconstraint, 25
pyknow.matchers, 26
pyknow.matchers.rete, 26
pyknow.matchers.rete.abstract, 27
pyknow.matchers.rete.check, 27
pyknow.matchers.rete.dnf, 28
pyknow.matchers.rete.mixins, 28
pyknow.matchers.rete.nodes, 29
pyknow.matchers.rete.token, 30
pyknow.matchers.rete.utils, 31
pyknow.operator, 31
pyknow.rule, 25
pyknow.strategies, 26
pyknow.watchers, 26
Index

A
activate() (pyknow.matchers.rete.abstract.OneInputNode method), 27
activate_left() (pyknow.matchers.rete.abstract.TwoInputNode method), 27
activate_right() (pyknow.matchers.rete.abstract.TwoInputNode method), 27
Activation (class in pyknow.activation), 21
add() (pyknow.matchers.rete.nodes.BusNode method), 29
add_child() (pyknow.matchers.rete.abstract.Node method), 27
add_child() (pyknow.matchers.rete.mixins.AnyChild method), 28
Agenda (class in pyknow.agenda), 22
AND (class in pyknow.conditionalelement), 22
AnyChild (class in pyknow.matchers.rete.mixins), 28
as_dict() (pyknow.fact.Fact method), 24

B
BaseField (class in pyknow.fact), 24
BETWEEN() (in module pyknow.operator), 32
build_alpha_part() (pyknow.matchers.rete.ReteMatcher static method), 26
build_beta_part() (pyknow.matchers.rete.ReteMatcher static method), 26
build_network() (pyknow.matchers.rete.ReteMatcher method), 26
BusNode (class in pyknow.matchers.rete.nodes), 29

C
callback (pyknow.matchers.rete.mixins.ChildNode attribute), 28
changes (pyknow.factlist.FactList attribute), 24
changes() (pyknow.abstract.Matcher attribute), 24
changes() (pyknow.matchers.rete.ReteMatcher method), 26
Check (class in pyknow.matchers.rete.abstract), 27
check (pyknow.matchers.rete.check.CheckFunction attribute), 27
CheckFunction (class in pyknow.matchers.rete.check), 27
ChildNode (class in pyknow.matchers.rete.mixins), 28
ConflictSetNode (class in pyknow.matchers.rete.nodes), 29
CONTAINS() (in module pyknow.operator), 32
copy() (pyknow.fact.Fact method), 24
copy() (pyknow.matchers.rete.token.Token method), 30

D
declare() (pyknow.engine.KnowledgeEngine method), 22
declare() (pyknow.matchers.rete.CheckFunction attribute), 27
declare() (pyknow.matchers.rete.FactList method), 24
DepthStrategy (class in pyknow.strategies), 26
dnf() (in module pyknow.matchers.rete.dnf), 28
duplicate() (pyknow.engine.KnowledgeEngine method), 23

E
eq() (in module pyknow.operator), 32
EXISTS (class in pyknow.conditionalelement), 22
expected (pyknow.matchers.rete.check.CheckFunction attribute), 27
extract_facts() (in module pyknow.matchers.rete.utils), 31

F
Fact (class in pyknow.fact), 24
FactCapture (class in pyknow.matchers.rete.check), 27
FactList (class in pyknow.matchers.rete.FactList), 23
FeatureCheck (class in pyknow.matchers.rete.check), 27
FeatureTesterNode (class in pyknow.matchers.rete.nodes), 29
Field (class in pyknow.fact), 25
FORALL (class in pyknow.conditionalelement), 22
from_iter() (pyknow.matchers.rete.Fact class method), 24

G
GE() (in module pyknow.operator), 32
generate_checks() (in module pyknow.matchers.rete.utils), 31
get_activations() (pyknow.engine.KnowledgeEngine method), 23
| get_activations() (pyknow.matchers.rete.nodes.ConflictSetNode method), 29 |
| get_check_function() (pyknow.matchers.rete.check.FeatureCheck method), 28 |
| get_deffacts() (pyknow.engine.KnowledgeEngine method), 23 |
| get_key (pyknow.strategies.DepthStrategy attribute), 26 |
| get_next() (pyknow.agenda.Agenda method), 22 |
| get_rules() (pyknow.engine.KnowledgeEngine method), 23 |
| GT() (in module pyknow.operator), 32 |
| halt() (pyknow.engine.KnowledgeEngine method), 23 |
| has_field_constraints() (pyknow.fact.Fact method), 24 |
| has_nested_accessor() (pyknow.fact.Fact method), 24 |
| HasMatcher (class in pyknow.matchers.rete.mixins), 28 |
| ILIKE() (in module pyknow.operator), 32 |
| InitialFact (class in pyknow.fact), 25 |
| INVALID (pyknow.matchers.rete.token.Token.TagType attribute), 30 |
| invalid() (pyknow.matchers.rete.token.Token class method), 31 |
| IS() (in module pyknow.operator), 32 |
| IS_NOT() (in module pyknow.operator), 32 |
| is_special() (pyknow.fact.Fact static method), 24 |
| is_valid() (pyknow.matchers.rete.token.Token method), 31 |
| key_a (pyknow.matchers.rete.check.CheckFunction attribute), 27 |
| key_b (pyknow.matchers.rete.check.CheckFunction attribute), 27 |
| KnowledgeEngine (class in pyknow.engine), 22 |
| L (class in pyknow.fieldconstraint), 25 |
| LE() (in module pyknow.operator), 32 |
| LIKE() (in module pyknow.operator), 32 |
| LT() (in module pyknow.operator), 32 |
| match (pyknow.fieldconstraint.P attribute), 25 |
| Matcher (class in pyknow.abstract), 21 |
| modify() (pyknow.engine.KnowledgeEngine method), 23 |
| NE() (in module pyknow.operator), 32 |
| new_conditions() (pyknow.rule.Rule method), 25 |
| N (class in pyknow.matchers.rete.abstract), 27 |
| NODEFAULT (pyknow.fact.Field attribute), 25 |
| NoMemory (class in pyknow.matchers.rete.mixins), 29 |
| NOT (class in pyknow.conditionalelement), 22 |
| NotNode (class in pyknow.matchers.rete.nodes), 30 |
| O (class in pyknow.conditionalelement), 22 |
| OrdinaryMatchNode (class in pyknow.matchers.rete.nodes), 29 |
| P (class in pyknow.fieldconstraint), 25 |
| prepare_rule() (in module pyknow.matchers.rete.utils), 31 |
| prepare_ruleset() (pyknow.matchers.rete.ReteMatcher static method), 26 |
| print_network() (pyknow.matchers.rete.ReteMatcher method), 26 |
| pyknow (module), 21 |
| pyknow.abstract (module), 21 |
| pyknow.activation (module), 21 |
| pyknow.agenda (module), 22 |
| pyknow.conditionalelement (module), 22 |
| pyknow.engine (module), 22 |
| pyknow.fact (module), 24 |
| pyknow.factlist (module), 23 |
| pyknow.fieldconstraint (module), 25 |
| pyknow.matchers (module), 26 |
| pyknow.matchers.rete (module), 26 |
| pyknow.matchers.rete.abstract (module), 27 |
| pyknow.matchers.rete.check (module), 27 |
| pyknow.matchers.rete.dnf (module), 28 |
| pyknow.matchers.rete.mixins (module), 28 |
| pyknow.matchers.rete.nodes (module), 29 |
| pyknow.matchers.rete.token (module), 30 |
| pyknow.matchers.rete.utils (module), 31 |
| pyknow.operator (module), 31 |
| pyknow.rule (module), 25 |
| pyknow.strategies (module), 26 |
| pyknow.watchers (module), 26 |
| REGEX() (in module pyknow.operator), 32 |
| remove() (pyknow.matchers.rete.nodes.BusNode method), 29 |
| reset() (pyknow.abstract.Matcher method), 21 |
| reset() (pyknow.engine.KnowledgeEngine method), 23 |
| reset() (pyknow.matchers.rete.abstract.Node method), 27 |
| reset() (pyknow.matchers.rete.ReteMatcher method), 26 |
| ReteMatcher (class in pyknow.matchers.rete), 26 |
retract() (pyknow.engine.KnowledgeEngine method), 23
retract() (pyknow.factlist.FactList method), 24
Rule (class in pyknow.rule), 25
run() (pyknow.engine.KnowledgeEngine method), 23

S
SameContextCheck (class in pyknow.matchers.rete.check), 28
show_network() (pyknow.matchers.rete.ReteMatcher method), 26
Strategy (class in pyknow.abstract), 21

T
TEST (class in pyknow.conditionalelement), 22
to_info() (pyknow.matchers.rete.token.Token method), 31
to_invalid_token() (pyknow.matchers.rete.token.TokenInfo method), 31
to_valid_token() (pyknow.matchers.rete.token.TokenInfo method), 31
Token (class in pyknow.matchers.rete.token), 30
Token.TagType (class in pyknow.matchers.rete.token), 30
TokenInfo (class in pyknow.matchers.rete.token), 31
TRUTH (in module pyknow.operator), 32
TwoInputNode (class in pyknow.matchers.rete.abstract), 27
TypeCheck (class in pyknow.matchers.rete.check), 28

U
unpack_exp() (in module pyknow.matchers.rete.dnf), 28
unwatch() (in module pyknow.watchers), 26
update() (pyknow.fact.Fact method), 24
update_agenda() (pyknow.abstract.Strategy method), 21

V
VALID (pyknow.matchers.rete.token.Token.TagType attribute), 30
valid() (pyknow.matchers.rete.token.Token class method), 31
Validable (class in pyknow.fact), 25
validate() (pyknow.fact.BaseField method), 24
validate() (pyknow.fact.Fact method), 25
validate() (pyknow.fact.Field method), 25
value (pyknow.fieldconstraint.L attribute), 25

W
W (class in pyknow.fieldconstraint), 25
watch() (in module pyknow.watchers), 26
WhereCheck (class in pyknow.matchers.rete.check), 28
WhereNode (class in pyknow.matchers.rete.nodes), 30
wire_rule() (in module pyknow.matchers.rete.utils), 31