experta Documentation

Release unknown

Roberto Abdelkader Martínez Pérez

Jan 23, 2023

Contents

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

```
from random import choice
from experta 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"])
```

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

CHAPTER 1

Migrating from Pyknow

Experta is a Pyknow fork. Just replace any *pyknow* references in your code/examples to *experta* and everything should work the same.

CHAPTER 2

Examples

You can find some more examples on GitHub.

CHAPTER 3

User Guide

3.1 Introduction

3.1.1 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.

3.1.2 Features

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

3.1.3 Difference between CLIPS and Experta

1. CLIPS is a programming language, Experta 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, Experta 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.

3.2 Installation

3.2.1 From PyPI

To install Experta, run this command in your terminal:

```
$ pip install experta
```

3.2.2 Getting the source code

Experta is developed on Github.

You can clone the repository using the git command:

```
$ git clone https://github.com/nilp0inter/experta.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
```

3.3 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.

3.3.1 Facts

Facts are the basic unit of information of Experta. 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 experta 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*.

3.3.2 Rules

In Experta 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'
```

(continues on next page)

(continued from previous page)

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

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

```
@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 `experta.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**, Experta 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/experta/experta/engine.py", line 210, in declare
    self.__declare(*facts)
  File "/home/experta/experta/engine.py", line 191, in __declare
    "Declared facts cannot contain conditional elements")
TypeError: Declared facts cannot contain conditional elements
```

3.3.3 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.

```
@DefFacts()
def needed_data():
    yield Fact(best_color="red")
```

(continues on next page)

(continued from previous page)

```
    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.

3.3.4 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

```
from experta 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!
```

```
$ 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 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)
```

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 Experta, 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.

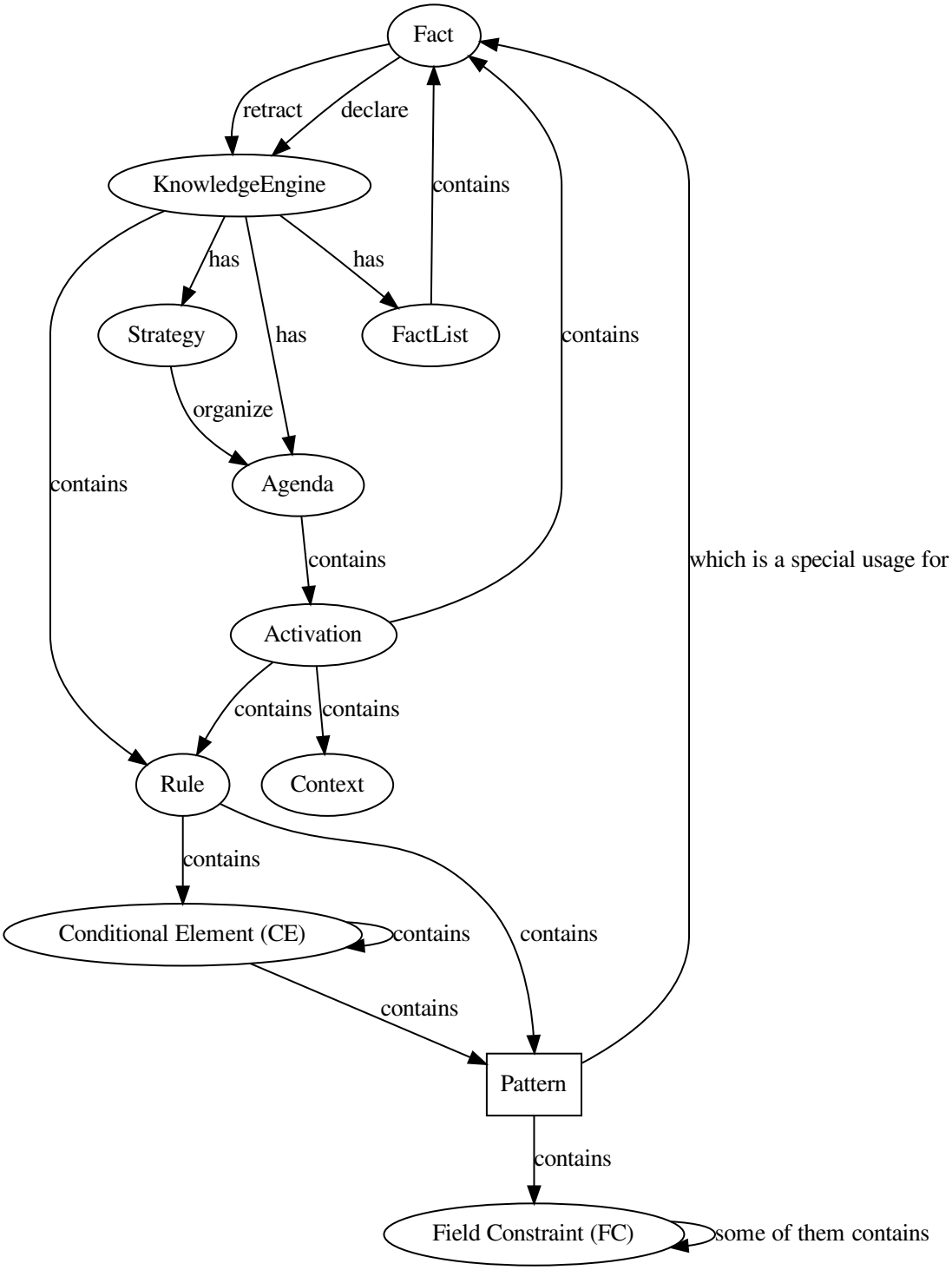
Difference between *DefFacts* and *declare*

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*.

3.4 Reference

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

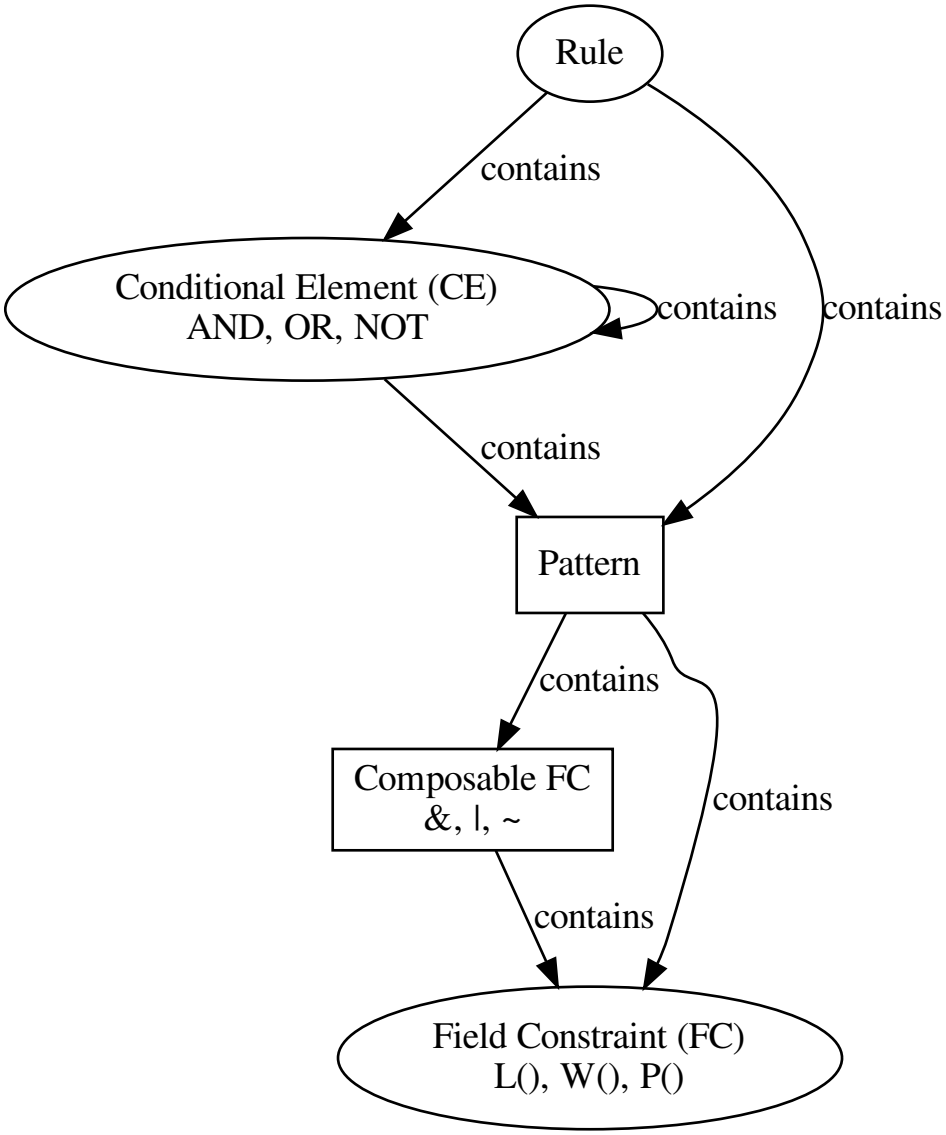

Fact
retract
declare
KnowledgeEngine
contains
has
has
Strategy
has
FactList
contains
organize
contains
Agenda
contains
which is a special usage for
Activation
contains
contains
Rule
Context
contains
Conditional Element (CE)
contains
contains
contains
Pattern
contains
Field Constraint (FC)
some of them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

3.4.1 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.

```
@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*

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

(continues on next page)

(continued from previous page)

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

3.4.2 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)

```
@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 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 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 7: Match for all numbers *a*, *b*, *c* where a > b > c

```
@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

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

```
@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.

3.4.3 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*

```
@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 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 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
```

3.4.4 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.

```
@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*.

```
@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*.

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

3.4.5 Variable Binding: The << Operator

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

Listing 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
```

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.

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

Deprecated since version 1.2.0: Use *AS* object instead.

3.4.6 MATCH object

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

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

Is exactly the same as:

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

3.4.7 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.

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

Is exactly the same as:

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

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

3.4.8 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.

```
>>> 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:

```
>>> @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.

```
>>> class Ship(Fact):
...     pass
...
>>> Ship(data={
...      "name": "SmallShip",
...      "position": {
...          "x": 300,
...          "y": 200},
...      "parent": {
...          "name": "BigShip",
...          "position": {
...              "x": 150,
...              "y": 300}}})
```

In this example we can check for collision between a ship and its parent with the following rule:

```
>>> @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.

```
>>> 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 byee!")
```

3.4.9 Mutable objects

Experta'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 *experta.utils.freeze* is used internally.

```
>>> 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()
```

(continues on next page)

(continued from previous page)

```
>>> 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 *experta.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 experta.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 *experta.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
set is Mutable
>>>
```

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

3.5 Cookbook

CHAPTER 4

API Documentation

4.1 Modules documentation

CHAPTER 5

Release History

CHAPTER 6

CHANGELOG

6.1 1.9.4

- Resolved #10. Corner case in DepthStrategy in which the first activation of the agenda didn't get removed.

6.2 1.9.3

- Resolved #11 that was introduced with #7, that caused the same behavior under different conditions.

6.3 1.9.2

- Resolved #7. In some situations last activation was removed right after entering the agenda.

6.4 1.9.1

- Resolved #3. Rules should not be called with already retracted facts.

6.5 1.9.0

- Drop Python 3.4 support.
- Use setup.cfg to save all package metadata.

6.6 1.8.0-1.8.2

- Rebranded to Experta.

6.7 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.

6.8 1.6.0

- Improved overall performance.

6.9 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.

6.10 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.

6.11 1.3.0

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

6.12 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.

6.13 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.

6.14 1.1.0

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

6.15 1.0.1

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

6.16 1.0.0

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

6.17 <1.0.0

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

CHAPTER 7

Source documentation

- genindex
- modindex
- search