Artificial Intelligence

Lab 3

Expert Systems

Agenda

- Introduction to Expert System.
- Structure of Expert System.
- Forward Chaining and Backward Chaining.
- What is PyKnow?
- Milestone 1.

New Course-sitesEnrollment

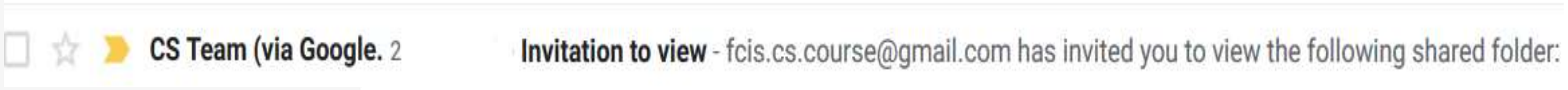
There is a problem in the configuration of the current course on CourseSites. Kindly, re-enroll to this new course-sites link.

Assignment 1 is now available on coursesites

Due Date: 13 March 2019 11:59PM.

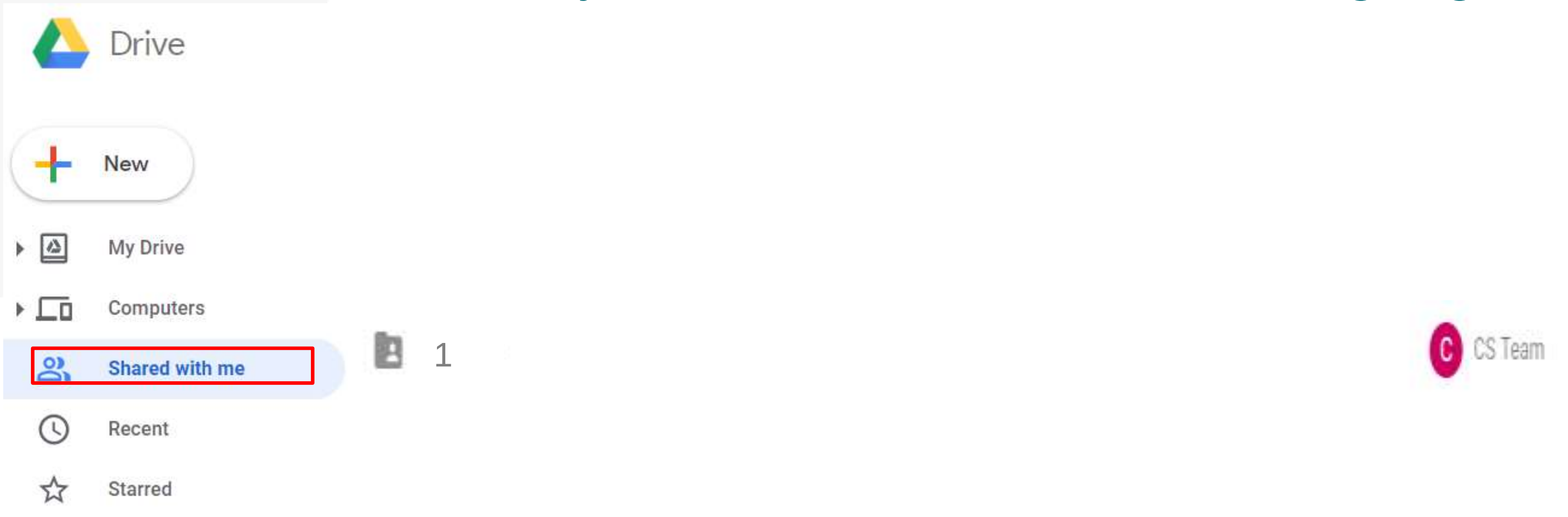
Delivery of Any Milestone

- Each team has received a mail from "CS Team" : fcis.cs.team@gmail.com notifying you that you have a shared folder named with your team number.



CS Team (via Google. 2 **Invitation to view** - fcis.cs.course@gmail.com has invited you to view the following shared folder:

- Also, You can reach your shared folder from drive.google.com



Delivery of Any Milestone

- Each Group has an access on a shared folder on google drive named with 'Team Number'.
- Don't RENAME that folder.
- Make sure to upload all folders with no renaming or change in hierarchy (complete project) without deleting any file before the deadline (Don’t delete old delivered milestones).
- Don't upload any compressed folders (Compressed folder will not be evaluated).
- Create ONLY ONE Project Named ‘AI-Package’

Delivery of Any Milestone

- For each milestone, you will create a new folder inside your project named with lab name.
  For example for first milestone, add a folder named ‘ExpertSystems’.
- Make sure all folders and files are synced before deadline, No EXCUSES for syncing issues will be accepted.
- No Manual/Mail delivery for any milestone.
- If you face any issues, you have time to ask your TA before the deadline.
- Don’t forget to share the folder with your team-mates and don’t remove CS Team.

Expert Systems

- Expert Systems solve problems that are normally solved by human experts.
- An expert system is a computer program that represents and reasons with knowledge of some specialist subject with a view to solving problems or giving an advice.
- To solve expert-level problems, expert systems will need efficient access to a substantial domain knowledge base which must be built as efficiently as possible.
- They also need to exploit one or more reasoning mechanisms to apply their knowledge to the problems they are given.
- Knowledge is represented by rules.

Expert Systems

- An expert system is divided into two subsystems:
  - The inference engine
  - The knowledge base.

- The knowledge base represents facts and rules.

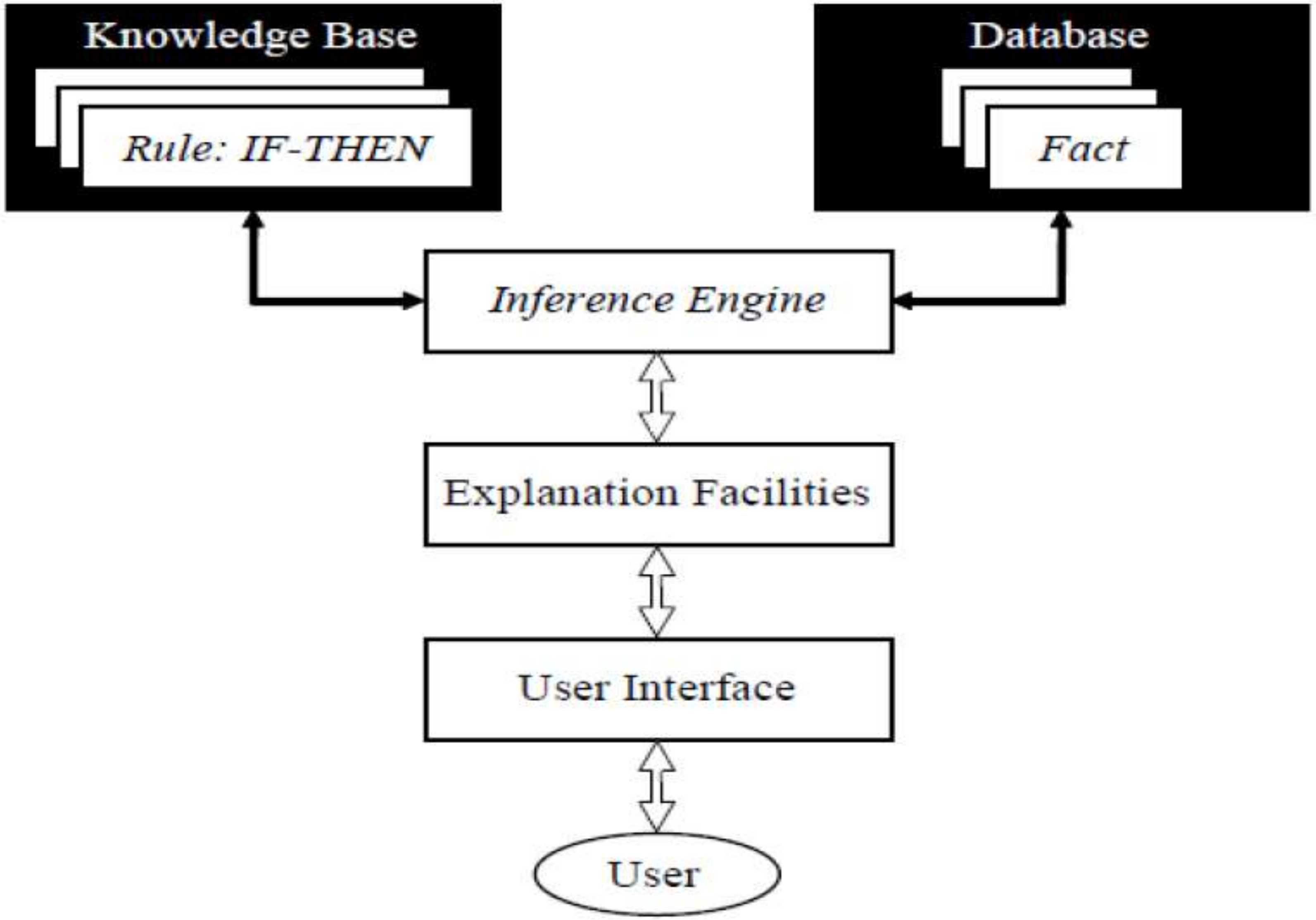
- The inference engine applies the rules to the known facts to deduce new facts.

Knowledge Representation - Rules

- The term rule in AI, which is the most commonly used type of knowledge representation.
- Rules can be defined as an IF-THEN structure.
- The given facts or information in the IF part to some action in Then part.
    - IF < condition > Then < action >.
- A rule can have multiple conditions joined by keyword AND, OR, or

  combination of them.
    - IF < condition1 > AND < condition2 > OR < condition3 > Then < action >.

Basic Structure of Expert Systems



Basic Structure of Expert Systems

- The knowledge base contains the domain knowledge that is useful for problem solving.

- In rule-based expert system, the knowledge is represented by a set of rules.

- Each rule specifies a relation, recommendation, strategy or heuristic and has the IF (condition) THEN action structure.

- When the condition part of a rule is said to fire the action part is executed.

Basic Structure of Expert Systems

The inference engine is a component of the system that applies logical rules to the knowledge base to deduce new information.

The inference engine carries out the reasoning whereby the expert system reaches the solution.

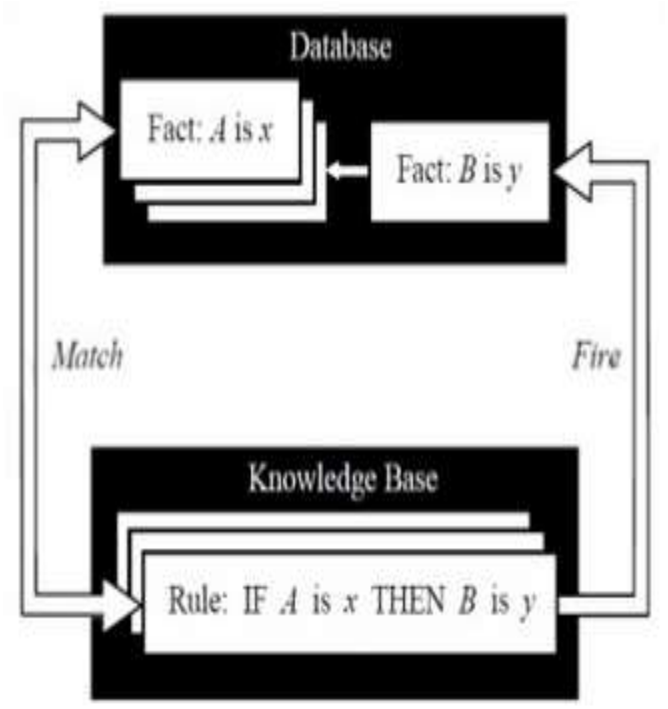
The explanation facilities enables the user to ask the expert system how a particular conclusion is reached and why a specific fact is needed.

Forward Chaining & Backward Chaining

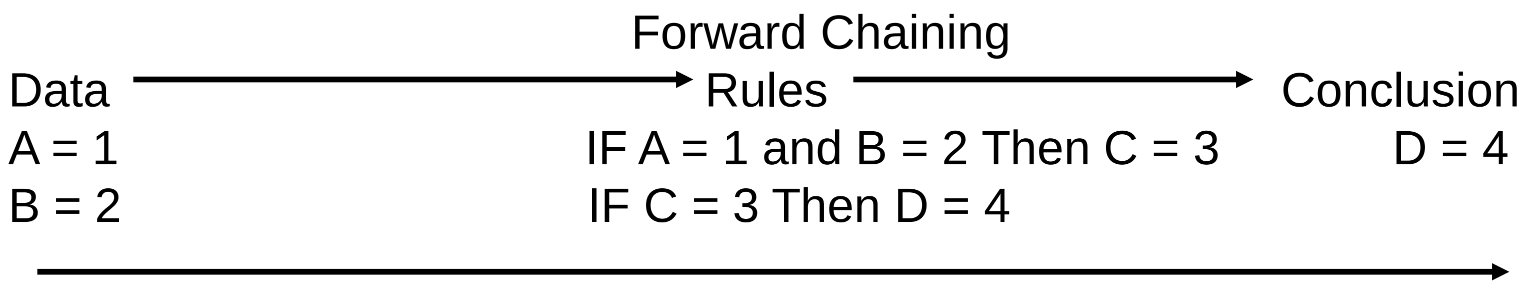
The Inference engine compares each rule stored in knowledge base with facts contained in the database.

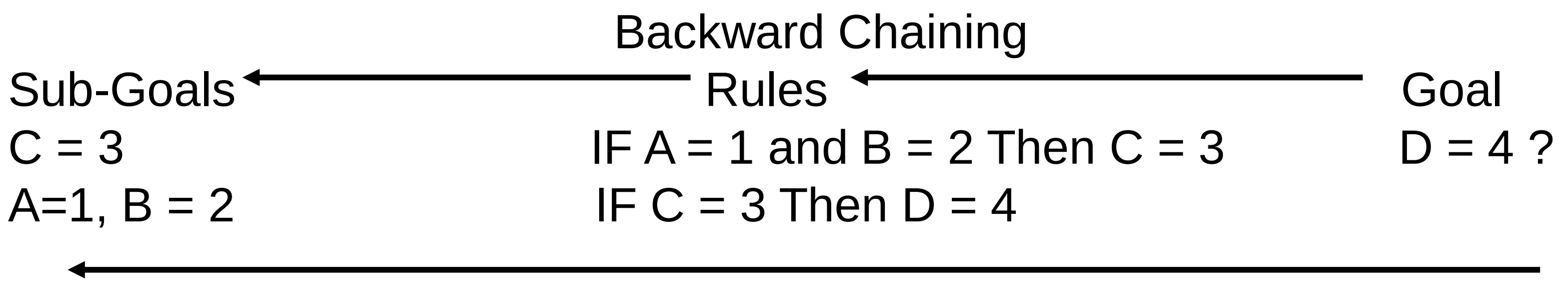
When the IF (condition) part of rule matches a fact, the rule is fired and its THEN (action) part is executed.

The matching of the rule IF parts to the facts produces inference chains. The inference chain indicates how an expert system applies the rules to reach conclusion.



Forward Chaining & Backward Chaining





Forward Chaining

- It is also known as data driven inference technique.
- Forward chaining matches the set of conditions and infer results from these conditions.
- Basically, forward chaining starts from a data and aims for any conclusion.
- It is bottom up reasoning also a breadth first search.
- It continues until no more rules can be applied or some cycle limit is met.

Forward Chaining Cont.

- Each time any rule can be executed only once.
- When rule is fired, It adds a new fact in the database.
- The match-fire cycle stops when no further rules can be fired.
- The forward chaining is a technique for gathering information and the inferring for it whatever can be inferred.

Forward Chaining Cont.

Example:

Suppose a new pet, Fritz, is delivered in a box along with two facts about Fritz:

- **Fritz bites**
- **Fritz eats flies**

The goal is to conclude the color of a pet named Fritz, based on a rule base containing the following four rules:

- **If X bites and X eats flies - Then X is a frog**
- **If X chirps and X sings - Then X is a canary**
- **If X is a frog - Then X is green**
- **If X is a canary - Then X is yellow**

Forward Chaining Cont.

With forward reasoning, an inference engine can derive that Fritz is green in two steps.

Remember Rule #1: **If X bites and X eats flies – Then X is a frog.**

1. Since the base facts indicate that "Fritz bites" and "Fritz eats flies", the condition of rule #1 is satisfied by substituting Fritz for X, and the inference engine concludes:

Fritz is a frog.

Remember Rule #3:**If X is a frog – Then X is green.**

2. The condition of rule #3 is then satisfied by substituting Fritz for X, and the inference engine concludes:

Fritz is green.

Backward Chaining

- It is also called as **goal driven inference** technique.
- It is a backward search from goal to the conditions used to get the goal.
- Basically it starts from a goal and aims for necessary data.
- It is top down reasoning or a depth first search.
- It processes operations in a backward direction from end to start, it will stop when the matching initial condition is met.

Backward Chaining Cont.

Example:

Suppose a new pet, Fritz, is delivered in a box along with two facts about Fritz:

- Fritz bites
- Fritz eats flies

The goal is to decide whether Fritz is green?!, based on a rule base containing the following four rules:

- If X bites and X eats flies - Then X is a frog
- If X chirps and X sings - Then X is a canary
- If X is a frog - Then X is green
- If X is a canary - Then X is yellow

Backward Chaining Cont.

With backward reasoning, an inference engine can determine whether Fritz is green in four steps.

1. Fritz is substituted for X in rule #3 to see if its consequent matches the goal, so rule #3 becomes:

   If Fritz is a frog - Then Fritz is green.

Since the consequent matches the goal ("Fritz is green"), the rules engine now needs to see if the ("Fritz is a frog") can be proved. The condition therefore becomes the new sub-goal: **Fritz is a frog**

2. Again substituting Fritz for X, rule #1 becomes:

   If Fritz bites and Fritz eats flies - Then Fritz is a frog.

Since the consequent matches the current goal ("Fritz is a frog"), the inference engine now needs to see if the ("Fritz bites and eats flies") can be proved. The condition therefore becomes the new sub-goals: **Fritz bites and Fritz eats flies.**

Backward Chaining Cont.

3. Since this goal is a conjunction of two statements, the inference engine breaks it into two sub-goals, both of which must be proved:
   - **Fritz bites**
   - **Fritz eats flies**

4. To prove both of these sub-goals, the inference engine sees that both of these sub-goals were given as initial facts. Therefore, the conjunction is true: Fritz bites and Fritz eats flies

- therefore the antecedent of rule #1 is true and the consequent must be true: Fritz is a frog

- therefore the antecedent of rule #3 is true and the consequent must be true: Fritz is green

PyKnow==1.7.0

- PyKnow is a Python library for building expert systems
- Matcher based on the forward chaining (the Rete algorithm).
- You can find the documentation here.
- To install PyKnow, run this command in your terminal:
  - pip install pyknow

The basics:

- Facts
- DefFacts
- Rules
- Knowledge Engine

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. 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’
```

Facts Cont.

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

```
from pyknow import Fact
class Alert(Fact):
    """The alert."""
    pass
class Status(Fact):
    """The system status."""
    pass

fact1 = Alert(message = 'This is an alert')
fact2 = Status(state = 'critical')
print(fact1['message']) #This is an alert
print (fact2['state']) #Critical
```

Rules

- Rules have two components, LHS (left-hand-side) and RHS (right-hand-side).
- The LHS describes 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.
- They are written inside Knowledge Engine Class.

```
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
```

Rules Cont.

```
class MyFact(Fact):
    pass


@Rule(MyFact(type = 'animal', family='felinae'))
def match_with_cats():
    """ Match with every `Fact` which:
    * f['type'] == 'animal'
    * f['family'] == 'felinae'"""
    print("Meow!")
```

Rule Conditional Elements

Conditional Elements: creates a composed conditions containing all Facts passed as arguments.

AND :

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

OR :

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

NOT :

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

Rules Cont.

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()
```

Rules 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 ==.

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

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

W (Wildcard Field Constraint): This element matches with any value (Not NULL).

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

Composing FCs

All FC can be composed together using the composition operators.

- ANDFC() a.k.a. &
- ORFC() a.k.a |
- NOTFC() a.k.a ~

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

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

MATCH object

The MATCH objects helps generating more readable name bindings.

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

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

AS Object

The AS object like the MATCH object i generating bind-able variables.

Myfact is an Object of myFact(Fact)

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

    pass
```

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

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.
- All DefFacts inside a KnowledgeEngine will be called every time the reset method is called.
- The decorated method MUST be generators.

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

Declare

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

```
engine = KnowledgeEngine()
engine.reset()
engine.declare(Fact(score=5))
print(engine.facts)
```

```
<f-0> InitialFact()
<f-1> Fact(score=5)
```

Difference between *DefFacts* and *declare Facts*

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.

Knowledge Engine

- 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.

Engine execution procedure

- This is the usual process to execute a KnowledgeEngine.
- The class must be instantiated.
- 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.
- The run method must be called. This starts the cycle of execution.

```
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())), F
    def ask_location(self):
        self.declare(Fact(locatio

    @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!
```

Check if the
value f

Name is

Declare New fact with name equals user input value

Match
any v

Matches Rule location with any value for location

```
What's your name? Roberto
Where are you? Madrid
Hi Roberto! How is the weather in Madrid?
```

Install Pyknow offline

1. Extract Pyknow.rar
2. Open CMD and redirect to the path of PyKnow folder
   - For example:
     ```
     CD/d C:\user\desktop\Pyknow
     ```
3. Execute that command:
   ```
   python setup.py install
   ```

Hands On

Write an expert system that helps a Robot crosses the street based on the light.
If the color of light is red, it won't walk.

If the color of light is green, it will walk.

If the color of light is yellow or blinking-yellow, then it will be cautious.

Solution

Milestone 1

Expert Systems milestone deadline: 16 march 2019.

It will be published on course-sites.

General instructions:

- Regarding your AI-Package:
- Initially create a new project named 'AI-Package'.
- Add a new folder named 'ExpertSystems'.
- Add only one new '.PY' file for writing your code.
- For documentation of PyKnow check this link.

Regarding your submission file:

- Submit **only running** code that you have tested before.
- Your assignment should be written in ONE ".py" file, this file should include the solution of ALL the problems and a main function that calls them.
- Compressed files (.zip/.rar) are **not allowed.**
- The Submission of team work package is only through **your shared folder on google drive.**
- **Don't delete** any previous milestones.

Questions?