

Artificial Intelligence

Algorithms and Applications with Python

Lectorial 03



Dr. Dominik Jung
dominik.jung42@gmail.com



python



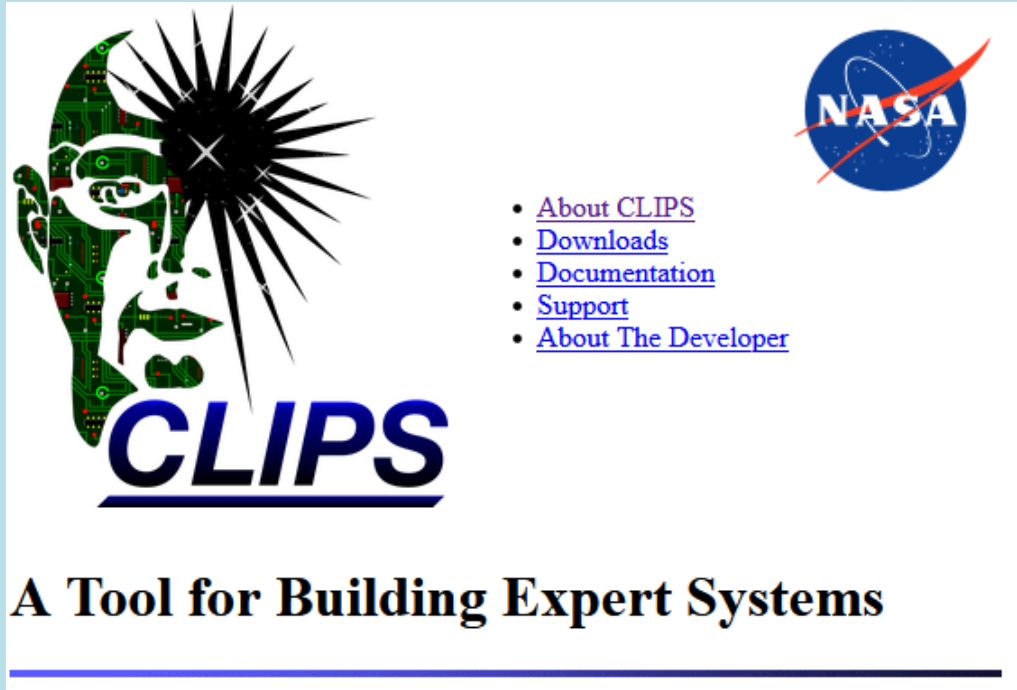
Agenda

1.1 Basics and Repetition

1.2 Building a Rule-Based Agent for Credit Scoring

- This is a lectorial: I will explain/repeat the most important concepts and then you try to solve the programming task by your own
- You are explicitly encouraged to solve this task in groups. And I will help you and give suggestions. However, there is no perfect solution, you will get a possible solution.
- If the task is too hard for you at the moment – relaxe 😊. Just look at the task again at a later point in the course.

C Language Integrated Production System (CLIPS)



CLIPS A Tool for Building Expert
Systems: www.clipsrules.net

- CLIPS is probably the most widely used expert system tool
- Designed using the C language at the NASA/Johnson Space center
- Multiparadigm programming language (rule-based, object-oriented, procedural)
- Many Python alternatives available, PyKnow/Experta are the most popular. However, they do not reach the power of pure CLIPS (performance, expressions etc.)

Image sources: ↗ [Wikipedia](#) (2020) by National Aeronautics and Space Administration; ↗ www.clipsrules.net (2020)

CLIPS Applications in Science and Engineering

NTRS - NASA Technical Reports Server

Satellite simulations utilizing CLIPS
Simulations provide necessary testbeds for system designs. Currently we are developing software whose main requirement is to produce CLIPS executable simulation code of a user prespecified system. This process minimizes the amount of engineering effort required to specify a system thereby reducing cost and providing the capability to quickly revise system definitions. Modeling satellite systems is the primary objective toward which testing has, and is, being conducted using satellite specifications. This paper describes the...

Document ID: 19960002917
Document Type: Conference Paper
Authors: Pauls, Barbara (Rockwell International Corp. Seal Beach, CA, United States)
Sherman, Mark (Rockwell International Corp. Seal Beach, CA, United States)
Date Acquired: September 6, 2013
Publication Date: August 15, 1990
Publication Information: Publication: NASA. Johnson Space Center, First CLIPS Conference Proceedings, Volume 2
Subject Category: COMPUTER PROGRAMMING AND SOFTWARE
Distribution Limits: Public
Copyright: Work of the US Gov. Public Use Permitted.

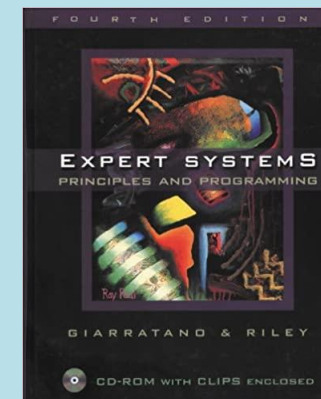
Fuzzy expert systems using CLIPS
This paper describes a CLIPS-based fuzzy expert system development environment called FCLIPS and illustrates its application to the simulated cart-pole balancing problem. FCLIPS is a straightforward extension of CLIPS without any alteration to the CLIPS internal structures. It makes use of the object-oriented and module features in CLIPS version 6.0 for the implementation of fuzzy logic concepts. Systems of varying degrees of mixed Boolean and fuzzy rules can be implemented in CLIPS. Design and implementation issues of...

Document ID: 19950013220
Document Type: Conference Paper
Authors: Le, Thach C. (Aerospace Corp. Los Angeles, CA, United States)
Date Acquired: September 6, 2013
Publication Information: Publication: NASA. Johnson Space Center, Third CLIPS Conference Proceedings, Volume 1
Subject Category: COMPUTER PROGRAMMING AND SOFTWARE
Distribution Limits: Public
Copyright: Work of the US Gov. Public Use Permitted.

NASA Technical Reports:

<https://ntrs.nasa.gov/search?q=CLIPS>

- Many NASA realworld scientific applications of CLIPS expert systems in Engineering and Science available on the NASA Technical Report Server



Giarratano, J. C., & Riley, G. (1989). *Expert systems: principles and programming*. Brooks/Cole Publishing Co..

Age of Empires II Uses CLIPS for AI Opponents

AGE
OF
EMPIRES
II
CONQUERORS
EXPANSION



Image sources: Microsoft Game Studios's Age of Empires II, Screenshot by my own

Experta (Python CLIPS Implementation)



The screenshot shows the PyPI page for the 'experta' package. The header is blue with the package name 'experta 1.9.4' and a 'pip install experta' button. A green badge indicates it's the 'Latest version' and it was released on Nov 16, 2019. The main content area has a left sidebar with navigation links: 'Project description' (selected), 'Release history', and 'Download files'. The main content area has a 'Project description' section with a code snippet and a 'Project links' section with a 'Homepage' link. A callout box with an arrow points to the 'Project description' section.

Search projects

Help Sponsor Log in Register

experta 1.9.4

pip install experta

Released: Nov 16, 2019

Expert Systems for Python

Navigation

- Project description
- Release history
- Download files

Project links

- Homepage

Statistics

GitHub statistics:

- Stars: 57

Project description

pypi v1.9.4 version build failing docs passing codecov 93%

Experta is a Python library for building expert systems strongly inspired by [CLIPS](#).

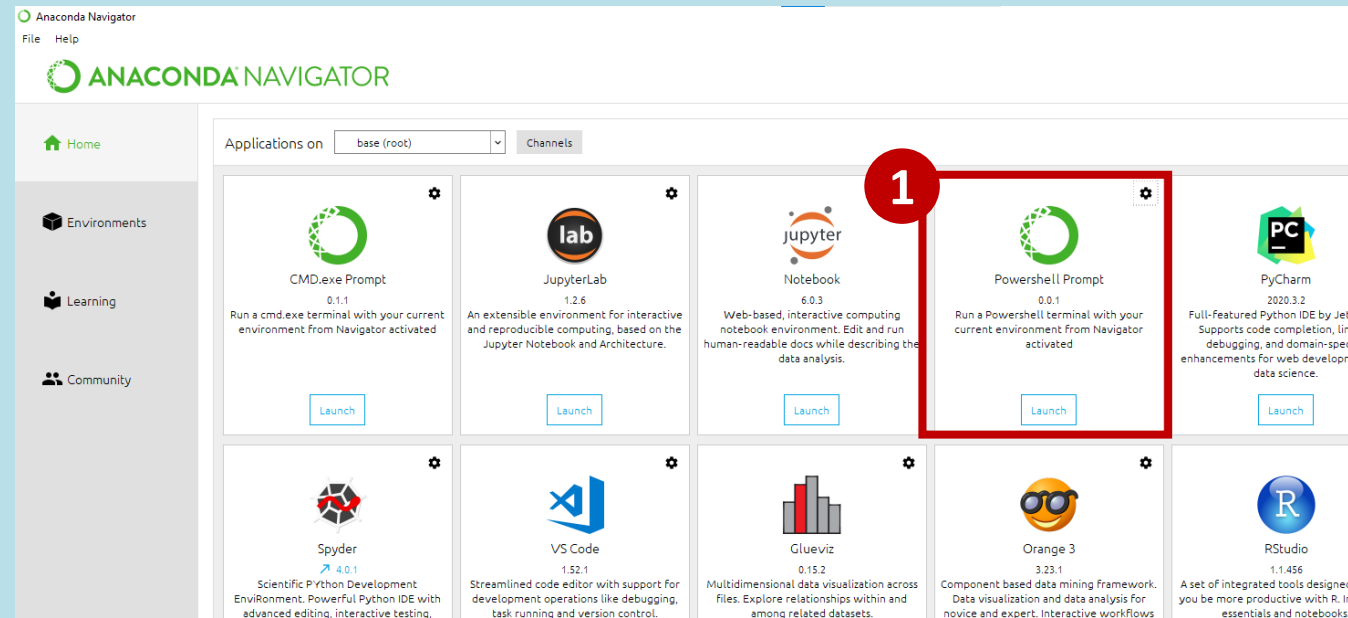
```
from random import choice
from experta import *

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

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

Experta:
<https://pypi.org/project/experta/>

Install Experta



- Experta is not available in the default anaconda repository, hence we have to install it from pypi
- Start your anaconda shell

2

- And type in the following pip install command:

```
pip install experta
```


- Facts are the basic unit of information. They are used by our inference algorithm to reason about the problem

```
>>> my_car = Fact(model="911 Turbo", horsepower=572)
>>> print(my_car["model"])

911 Turbo
```

- The `Fact` class is a subclass of `dictionnaires`, we already know from lecture 3
- The order of arguments is arbitrary in `facts`. Hence, the arguments can be created without keys, or mixed with key-values.

- You can define facts and use them later in your code

```
>>> class Status(Fact):  
>>>     pass  
  
>>> my_fact = Status(color = "red")  
>>> print(my_fact)  
  
Status(color='red')
```

Define Default Facts with DefFacts

- Most of the time our knowledge-based system needs a set of facts to work with. For that purpose we can use the `DefFacts` decorator

```
@DefFacts()  
def needed_data():  
    yield Fact(car_color="red")  
    yield Fact(price=170000)
```

- All `DefFacts` will be executed when we call the `reset()` method

Implement Rules

- We can also implement the two components of rules

$$\overbrace{\text{if my_car} = 911}^{\text{left-hand-side}} \rightarrow \overbrace{\text{me} = \text{happy}}^{\text{right-hand-side}}$$

```
class my_fact(Fact):  
    pass
```

```
@Rule(my_fact())
```

This is the left-hand-side of the rule

```
def matchWithEveryMyFact():  
    pass
```

This is the right-hand-side of the rule

- The left-hand-side of the rule describes the conditions on which the rule should be executed (fired)
- The right-hand-side describe the set of actions to perform, when the rule is fired

- We can implement different field conditions in our rules

```
@Rule(Fact(car = L(911) | L(718)))  
def foo():  
    pass
```

```
@Rule(Fact(name = W()))  
def foo():  
    pass
```

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

- Literal Field Constraint
- Check if the car element is exactly „911“ or „718“
- Wildcard Field Constraint
- Check if there is a fact with the key „name“
- Predicate Field Constraint
- Apply a callable to the fact-extracted value

Conditional Elements and Complex Rules I

- In most cases we want to express more complicated rules. We can do that with different conditions

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

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

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

- In an AND pattern, all of the passed conditions must match
- In an OR pattern, any of the pattern will make the rule match
- With NOT, we can express the absence of a condition

- In most cases, we want to express more complicated rules. We can do that with different conditions

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

```
@Rule(FORALL(Student(W("name")),  
             Exam(W("name")))  
  
def all_students_passed():  
    pass
```

- Check if one or more facts matches this pattern
- Will match only once while one or more matching facts exists
- In FORALL pattern, we can check if a group of specified conditions is satisfied for every occurrence of another specified condition

Working with Variable Bindings

- You can also bind variables to a name with the << operator.
- For instance, we can bind the first value of the matching fact to a name, e.g. „value“ and pass it to the function when fired:

```
@Rule(Fact(MATCH.value))  
def foo(value):  
    pass
```

```
@Rule(Fact("value" << W()))  
def foo(value):  
    pass
```

- Or we can do it for the whole matching fact

```
@Rule(As.my_fact Fact(W()))  
def foo(my_fact):  
    pass
```

```
@Rule(Fact("my_fact" << Fact()))  
def foo(my_fact):  
    pass
```

The Inference or KnowledgeEngine

- The KnowledgeEngine is the main part of your knowledge-base system:

```
class helloWorld(KnowledgeEngine):
    @DefFacts()
    def _initial_action(self):
        yield Fact(action="say_hello")

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

    # ... more rules here ...

    @Rule(Fact(action="say_hello"), Fact(name="name" << W()))
    def greet(self, name):
        print("Hi ", name)
```



```
>>> engine = helloWorld()  
>>> engine.reset()  
>>> engine.run()
```

```
Hey, what's your name? Dominik  
Hi Dominik
```

- In a productive system, we initialize it and populate it step by step with further facts, then we run it for inference



You can find the solution of each lection online in the git repository if you need some starting help!

Case

Your next job is in the *Financial Services Department*. You were ordered to build a credit scoring agent to automate most of the manual assessments of creditworthiness. Based on some first interviews you have noted the following comments:

Extract expert interview:

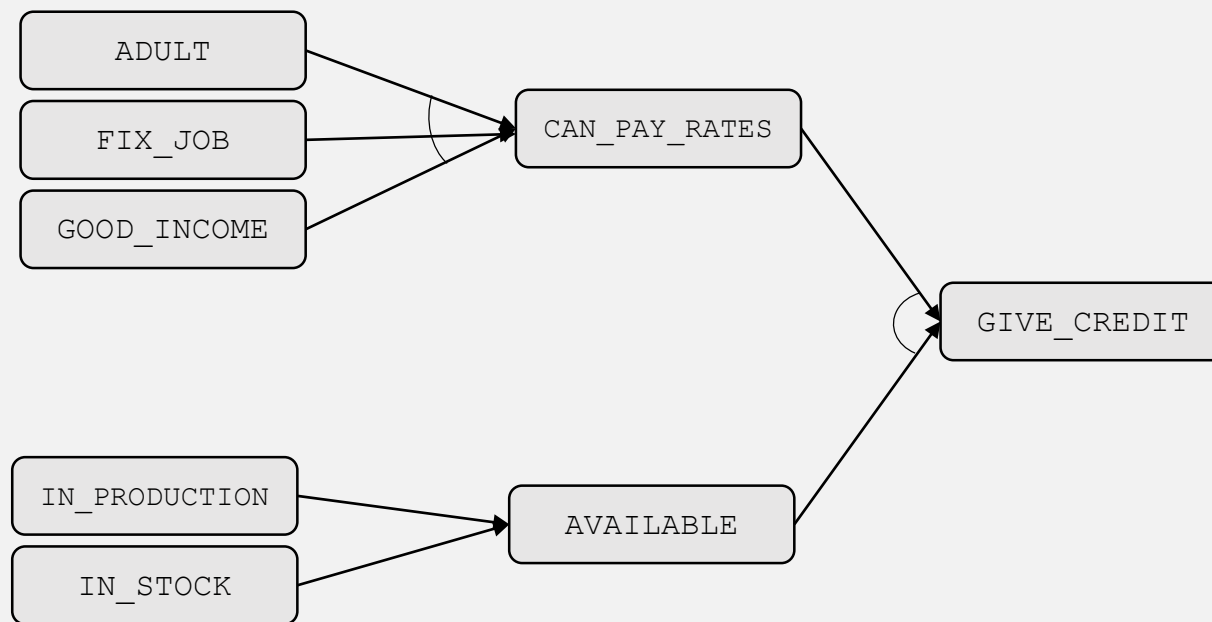
- *"In general we have customer, order or warehouse information that influence our car credit process. Our input fields in the credit system are credit and financial information and the ordered car model."*
- *"A customer can only get a car credit if he is adult, has a fix job and good income. We need this information to assess if he can pay the future credit rates of the car credit."*
- *"If a customer can pay the future car credit rates and the car is available we can give the credit"*
- *"A car can be in stock or we have to produce it. If it can be produced or if it's in stock the car is available for sale"*
- *"Due to the great success of our cars, we have currently no cars in stock. And we can only produce Taycans and 911."*

Please create a rule-based agent to automate the credit scoring process. Start with drawing a first sketch of the rule-network, and write down the input variables.

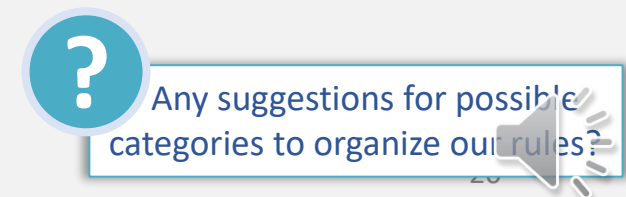
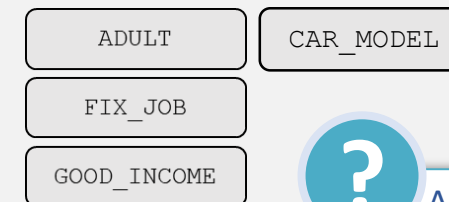
Then implement a first prototype with Python and Experta. You can start with the agent template for this lection or build your own agent from scratch (Lectorial 3 - Rule-based Agent Template.py).

1. Draw a Sketch of the Rule-Network

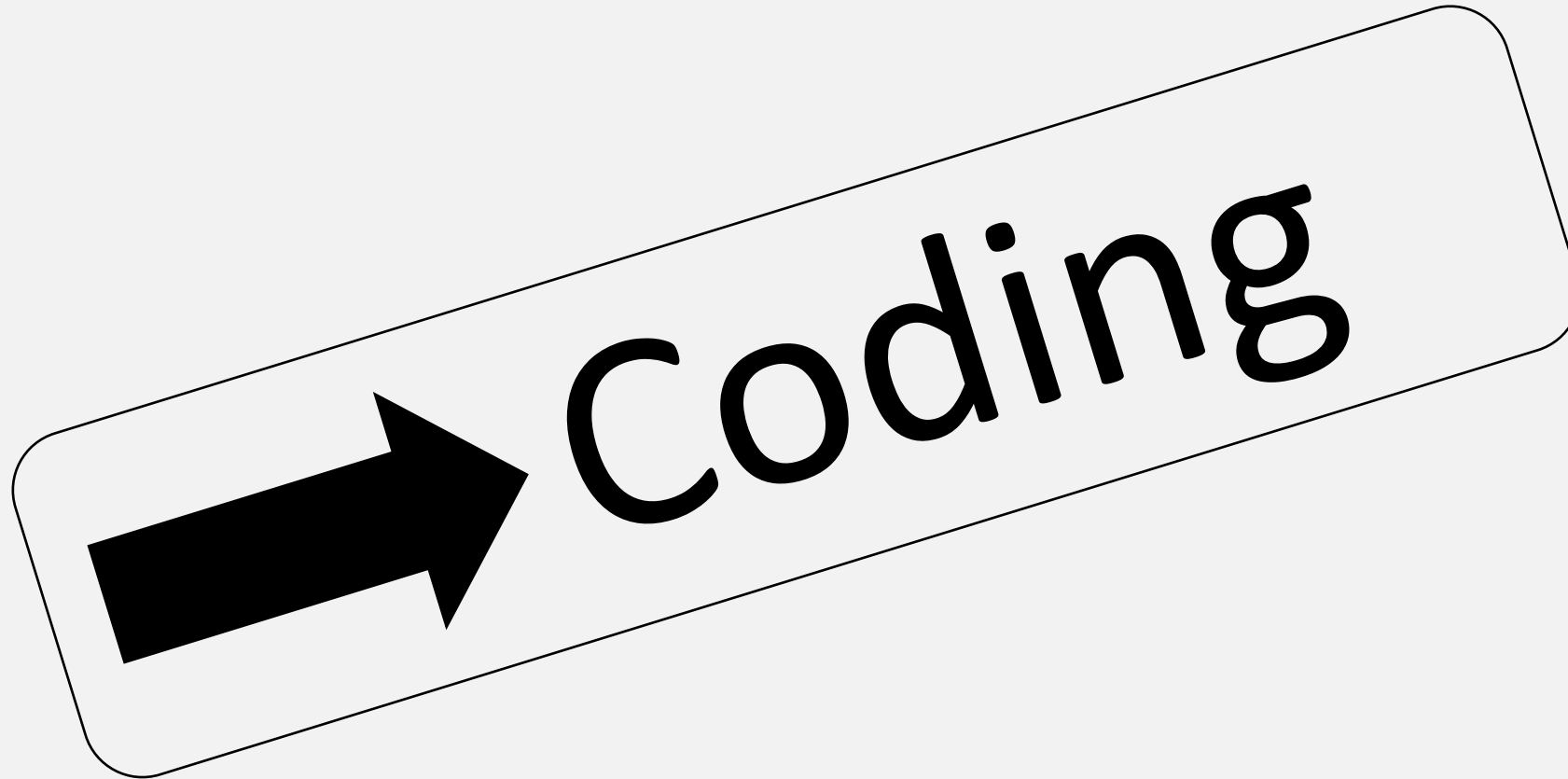
1. *"In general we have customer, order or warehouse information that influence our car credit process. Our input fields in the credit system are credit and financial information and the ordered car model."*
2. *"A customer can only get a car credit if he is adult, has a fix job and good income. We need this information to assess if he can pay the future credit rates of the car credit."*
3. *"If a customer can pay the future car credit rates and the car is available we can give the credit"*
4. *"A car can be in stock or we have to produce it. If it can be produced or if it's in stock the car is available for sale"*
5. *"Due to the great success of our cars, we have currently no cars in stock. And we can only produce Taycans and 911."*



- Rule-networks give an good overview of the different relationships in our knowledge base
- We will use this rule-network for the next step, the agent implementation
- Input variables:



Further questions?



2. Structure of the Agent

```
%% import python libs
from experta import *

%% Knowledge engine and rule base
class CreditScoring(KnowledgeEngine):
    @DefFacts()
    def _initial_action(self):
        pass

    @Rule(..)

%% Implement scoring agent
class CreditScoringAgent():
    def inference(self):
        pass

%% Run agent
agent = CreditScoringAgent()
agent.inference()
```

- In a first step we define the rule-base
- In a second step, the agent logic is implemented

2.1 Implement Fact Classes

- In a first step, we load the template and implement our three kind of fact classes (alternatively you can store them directly as `Fact()`)

```
## import python libs
from experta import *

## Rule-based system
class Customer(Fact):
    """ All info about the customer's credibility """
    pass

class Order(Fact):
    """ Order related information like model etc. """
    pass

class Warehouse(Fact):
    """ Company information about the car production """
    pass
```



These categories are just suggestions, you can also use other categories or just work with facts!

2.2 Implement Rules

- In the next step, let us take a look at the rules and fact definitions

```
class CreditScoring(KnowledgeEngine):  
    @DefFacts()  
    ?  
  
    @Rule()  
    def _(self):  
        pass
```



What is the difference between DefFacts and Rule? Where would you store which kind of information?

```
class CreditScoring(KnowledgeEngine):  
    @DefFacts()  
        def _initial_action(self):  
            yield Warehouse(in_stock=False)
```

“Due to the great success of our cars, we have currently no cars in stock”

2.2 Implement Rules (Agent logic)

```
class CreditScoring(KnowledgeEngine):
```

```
...
```

Rule 1: A customer can only get a car credit if he is adult, has a fix job and good income. We need this information to assess if he can pay the future credit rates of the car credit

Rule 2: ... And we can only produce Taycans and 911."

Rule 3: "A car can be in stock or we have to produce it. If it can be produced or if it's in stock the car is available for sale"

Rule 4: "If a customer can pay the future car credit rates and the car is available we can give the credit"

- "In general we have customer, order or warehouse information that influence our car credit process"
- "A customer can only get a car credit if he is adult, has a fix job and good income. We need this information to assess if he can pay the future credit rates of the car credit."
- "If a customer can pay the future car credit rates and the car is available we can give the credit"
- "A car can be in stock or we have to produce it. If it can be produced or if it's in stock the car is available for sale"
- "Due to the great success of our cars, we have currently no cars in stock. And we can only produce Taycans and 911."

2.2 Implement Rules (Agent logic)

```
class CreditScoring(KnowledgeEngine):
    ...

    @Rule(AND(Customer(adult = True),
                  Customer(fix_job = True),
                  Customer(good_income = True)))
    def is_creditworthy(self):
        self.declare(Customer(creditworthy=True))

    @Rule(OR(Order(model = L("911") | L("Taycan"))))
    def can_be_produced(self):
        print("Model can be produced")
        self.declare(Warehouse(producable=True))

    @Rule(OR(Warehouse(producable = True),
              Warehouse(in_stock = True)))
    def is_available(self):
        print("Car is available")
        self.declare(Warehouse(available=True))

    @Rule(AND(Customer(creditworthy = True),
              Warehouse(available = True)))
    def sell_car(self):
        print("Car can be sold")
```

- “In general we have customer, order or warehouse information that influence our car credit process”
- “A customer can only get a car credit if he is adult, has a fix job and good income. We need this information to assess if he can pay the future credit rates of the car credit.”
- “If a customer can pay the future car credit rates and the car is available we can give the credit”
- “A car can be in stock or we have to produce it. If it can be produced or if it’s in stock the car is available for sale”
- “Due to the great success of our cars, we have currently no cars in stock. And we can only produce Taycans and 911.”

2.3 Implement the Agent

- Finally, we can add the inference code to our agent logic

```
%% Implement scoring agent
class CreditScoringAgent():
    def inference(self):
        # We assume there is a database interface, where the agent can load the data
        engine = CreditScoring()
        engine.reset()

        # Example data from the data base
        engine.declare(Customer(adult=True, fix_job = True, good_income = True), Order(model
            = "911"))
        engine.run()

agent = CreditScoringAgent()
agent.inference()
```

- We can now add a database interface and let our agent score our customers during night time.

Case

Now get more familiar with rule-based programming and add some more rules to your rule-base, for that purpose expand the rule-based agent by your own:

- Play with some other variables and check if the agent works correctly
- Implement the rules from the credit scoring example from lecture 5 to expand the rule-base
- Take a look at the Experta documentation and implement further more sophisticated rules

Just
{ Keep }
Coding