Ministry of Education of Republic of Moldova
Technical University of Moldova
CIM Faculty
Anglophone Department

Report

On Logical Programming and Artificial Intelligence
Laboratory Work №1

Performed by: st. gt. FAF-172 **Cretu D.**
Verified by: prof. **Gavrilita M.**

Chisinau, 2021

Laboratory Work Nr. 1

Topic:

Expert system in Python

Tasks:

- Create an expert system for tourists classification and definition;
- Analyze expert system work and define main points of work.

System description:

Task was implemented via two approaches and each one differs from the other.

1. The first approach contains an OOP-based checking system requiring multiple sequenced inputs from the user for facts definition (consider that class is a form for rule and user input of facts). Parent class defines a form of rule facts and defines order of fact estimation by the user. Each child-object is a specific rule with an unique outcome as an answer. System compares the user-specified object with all rules objects field-by-field, collecting "compliance degree" values. User-specified object is considered as related to the rule one if it has the highest compliance degree amongst other "rules".
2. The second approach is a complex rule-based system where facts are statements and rule is a set of relations between facts. User inputs a random sequence of all known facts and then the system checks those facts to match with rules statement-by-statement considering relations between those statements in the rule. If there is a match, then the program gives an answer related to the corresponding rule. If there is no rule matching, then the program gives a "no match" answer.

The first approach does not provide an option for setting relations between facts, making rules work only as a set of facts with which user statements are compared. Another problem is that there is a limited set of facts that can be defined in a rule and in a user statement. Any change in a rule structure will also change the manner of user interaction, and will require additional coding.

The second approach is complex. It requires an effective base for facts definition and relations definitions between those facts (rules definitions). Effective base for rules definitions will make the system adaptable to work with any amount of facts and with any relations between facts. Such a system allows user statements estimations in unlimited and

non sequential manner. Potentially, such a system can be enhanced to receive new rules even from users.

System structure:

Figure 1 will present how the system works from an OOP-based (class based) solution. Considering that this figure will present only the current case structure of such a solution there are also specified roles of each class in the system.

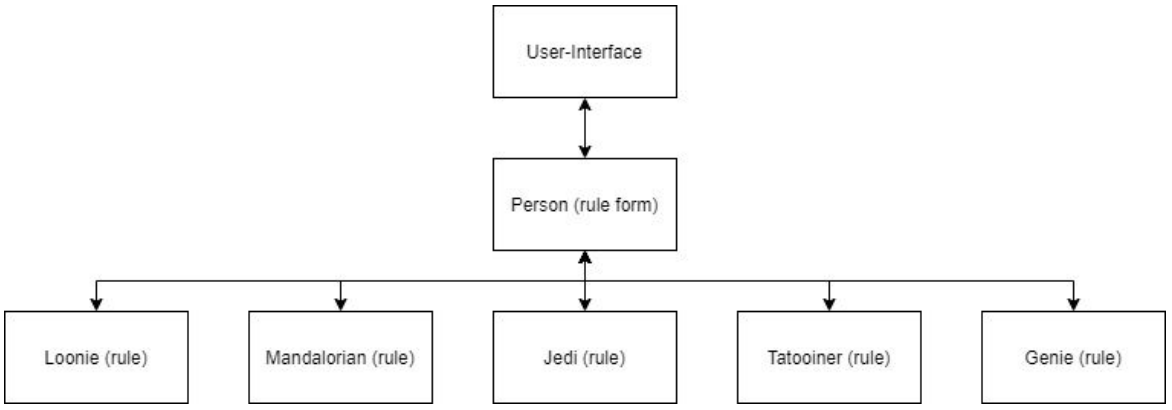
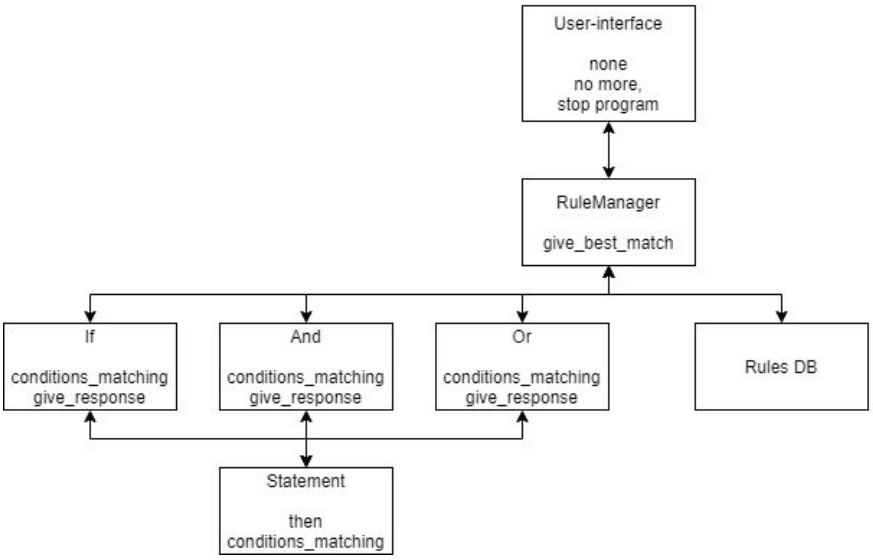


Figure 1: structure of class-based expert system

Below are listed all classes with their roles in this implementation:

- User interface - interface that allows typing known facts in predefined in current case manner, All inputs are then added to the specified object, that will be compared with other objects that represent “rules”;
- Person - parent-class that defines the form of all rules, how user input will be received by the system and establishes how objects will be compared;
- Children-classes - classes defining rules with outcomes. Each class is a rule, where fields are required facts matching with which increases the possibility of matching the rule. In current system there are 5 classes, defining unique rules:
    - Loonie;
    - Mandalorian;
    - Jedi;
    - Tatooiner;
    - Genie

Figure 2 represents a system from an adaptable and rule-based approach.



Below is presented general overview of all classes involved:

- User interface - listens for input from the user, where each input not equal to the command is considered as a statement/fact from the user that each time is inserted in a list of statements relating to some request. Commands make next actions:
    - none/no more - shows that the user stopped definition of statements/facts and request must be checked to match any of the available rules (calls for check);
    - stop program - cancel further execution of program and close it;
- Rule manager - rule manager at the stage of creation gets a database of rules and then redirects requests to autonomous rules modules that analyze if rule is matched and then redirects incoming answers to the user;
    - give_best_match - redirect request to all autonomous rules and receive answers, waiting for successful one;
- If - autonomous rule module consisting of one fact and response to meeting this fact, has two functions as all other rule modules:
    - conditions_matching - check if user request match the rule and give result;
    - give_response - give status of response basing on matching conditions of rule;
- And - autonomous rule module where rule will be matched if all facts of the rule are matched, has the same methods as If module;
- Or - autonomous rule module where rule will be matched if even one fact of the rule has matched, has the same methods as If and And modules;
- Statement - rule base module setting storage for all facts of rule and response of the rule. Has two methods:
    - then - specify answer of the rule if it is a successful match;
    - conditions_matching - abstract method that must be implemented by rule

Code and mentions:

First will be presented as class based solution parts of the program.

Listing 1: Person class initialization with fields specification in OOP-based program approach

```
class Person(object):

    def __init__(self, hair, eyes, ears, headshape, height, mouthSize,
language, skinColor):
        self.hair = hair
        self.eyes = eyes
        self.ears = ears
        self.headshape = headshape
        self.height = height
        self.mouthSize = mouthSize
        self.language = language
        self.skinColor = skinColor
```

Current class contains strict forms of possible rules and possible input from the user. If there will be new characteristics (new type of fact), then the program will require rewriting a big part of code. If type of fact will require change this will also cause code rewriting.

Listing 2: algorithm for checking match to facts in OOP-based program approach

```
if(isinstance(person, Person)):
            if self.hair == person.hair:
                complianceDegree += 1
            if self.eyes == person.eyes:
                complianceDegree += 1
            if self.ears == person.ears:
                complianceDegree += 1
            if self.headshape == person.headshape:
                complianceDegree += 1
            if self.height == person.height:
                complianceDegree += 1
            if self.mouthSize == person.mouthSize:
                complianceDegree += 1
            if self.language == person.language:
                complianceDegree += 1
            if self.skinColor == person.skinColor:
                complianceDegree += 1
            return complianceDegree
```

Each new rule of the system will be introduced as a new child of the parent class with its own facts, answer, but the number of facts and their sequence is strict.

Listing 3: rule estimation conform parent structure in OOP based program approach

```
class Genie(Person):
    def __init__(self):
        super().__init__(
            hair="none",
            eyes="foggy",
            ears="none",
            headshape="round",
            height="small",
            mouthSize="none",
            language="high elfian",
            skinColor="blue foggy"
        )
    def show_class(self):
        return "genie"
```

Another important moment is that all those children must be introduced into the database of rules as it is shown in Listing 4.

Listing 4: database of all rules in OOP-based program approach

```
class PersonalitiesDB:
    def __init__(self):
        self.listOfPersonalities = [Genie(), Mandalorian(), Jedi(),
Tatoiner(), Loonie()]
```

Another important issue is that the program receives input from the user in a defined manner and change of this manner will require code rewriting.

Listing 5: receiving input from user in OOP-based program approach

```
print("Input characteristics of the person using words with small letters
(if there is no concrete characteristic print 'unknown' or 'some'")
hair = input("Hair of the person (none, yellow, brown, black, white): ")
eyes = input("Eyes of the person (foggy, small green, brown, blue, black):
")
ears = input("Ears of the person (none, small round, small, small cubic):
")
headshape = input("Person shape of head (round, cubic): ")
height = input("Height of the person (small, medium, tall): ")
mouthSize = input("Person size of mouth (none, medium, small, big): ")
languages = input("Language that this person speaks the most (high elfian,
universal, force, tatooinean, lunar, universal): ")
skinColor = input("Skin color of the person (blue foggy, white, brown,
black): ")
```

After inserting all those values the program will check those facts to match to any of the rules or to find the most similar match.

Listing 6: rules check in OOP-based program approach

```
for personalityType in personsDB.listOfPersonalities:
    complienceDegree = personalityType.compliance_degree(randomPerson)
    if complienceDegree > bestComplianceValue:
        bestComplianceValue = complienceDegree
        bestComplianceType = personalityType
```

You can see that the current approach looks uneffective, unadaptable, non-usable. That is the reason why during the current work is shown the advantage of the rule-based system approach over OOP-based.

Now will review the rule-based approach of the program (or rule-based program).

Listing 7: statement class defining base principles of rules in rule-based system

```
class Statement(object):
    #   create object of the class
    def __init__(self):
        self.statementsList = []
        self.response = None

    #   set response field and return object
    def then(self, statement):
        self.response = statement
        return self

    #   abstract method that must be implemented by child
    def conditions_matching(self):
        raise NotImplementedError("Please Implement this method")
```

This is the base for rule. Here you can see that each rule contains a list of facts, possible positive answers and how facts relate to each other. In the “And” part of rule each fact must be matched to match the rule and so on, all those relations were mentioned above.

For showing how it can be implemented below is the interpretation of the “Or” rule part.

Listing 8: “Or” rule part code realization

```
class Or(Statement):
    def __init__(self, conditions):
        super().__init__()
        for condition in conditions:
            self.statementsList.append(condition)

    #   check how user conditions match all statements of rule
    def conditions_matching(self, userConditions):

        #   check if there is a list of user conditions
```

```
        if isinstance(userConditions, list):

            #   iterate through all statements and user conditions
            for userCondition in userConditions:
                for statement in self.statementsList:

            #   check if there is inner AND rule and check matching this
rule
                    if isinstance(statement, And):
                        if statement.conditions_matching(userConditions):
                            return True

                    #   if user condition matches any of statements
                    if userCondition == statement:
                        return True

        #   check if one user characteristic is meeting even one statement
        else:
            for statement in self.statementsList:
                if isinstance(statement, And):
                    if statement.conditions_matching(userConditions):
                        return True

                if userConditions == statement:
                    return True

        return False

    #   give response basing on matching result
    def give_response(self, userConditions):
        if self.conditions_matching(userConditions):
            return self.response
        else:
            return "no match"
```

After setting all required logical relations between facts is required to set manager of rules that will pick all further defined by user or experts rules inside database and how to interact with them.

Listing 9: Rule manager

```
class RuleManager(object):
    def __init__(self, listOfRules):
        self.rulesList = listOfRules

    #   return answer basing on user conditions matching any of the
available rules.
```

```
    def give_best_match(self, userConditions):
        response = None
         for rule in self.rulesList:
            if isinstance(rule, And) or isinstance(rule, Or) or
isinstance(rule, If) or isinstance(rule, Statement):
                response = rule.give_response(userConditions)
            else:
                raise TypeError("rules db has a non-rule object")

            if response != "no match":
                return response

        return "no match"
```

Manager is a simple class, considering that all ways of checking facts and how facts are related is defined in logical classes. The last part of setting an expert system is to provide a rules database that the manager will read at the initialization stage.

Listing 10: rules database example

```
rulesDB = [
    And(["has no hair", "has foggy eyes", "has no ears", "has round head",
Or(["speaks force", "speaks high elfian"]), "has blue foggy
body"]).then("genie"),

    And([Or(["has black hair", "has brown hair"]), Or(["has green eyes",
"has brown eyes", "has gray eyes"]), "has small round ears", "has round
head", "speaks universal", "has white skin body"]).then("mandalorian"),

    And([Or(["has black hair", "has brown hair"]), "has blue eyes", "has
small round ears", "has round head", Or(["speaks universal", "speaks
force"]), Or(["has white skin body", "has yellow skin body", "has black
skin body"])]).then("jedi"),

    And(["has yellow hair", "has blue eyes", "has medium cubic ears", "has
cubic head", Or(["speaks universal", "speaks tatooinean"]), "has black skin
body"]).then("tatooiner"),

    And(["has white hair", Or(["has white eyes", "has black eyes"]), "has
medium cubic ears", "has round head", Or(["speaks lunar", "speaks
tatooinean", "speaks force"]), "has white skin body"]).then("loonie")
]
```

This system can be easily modified to give rules to the manager in real time, possibly to manage already existing rules (modification or deleting), add new ones in real time, even give possibility to set new rules by the user. Lastly can be shown user interaction code.

Listing 11: user interface code

```
while True:
    answer = input(">>>\t")
    if answer == "none" or answer == "no more":

        if len(userConditions) == 0:
            print("\tWell, there is no condition from you, I can't even try
to guess who this can be ¯\_(ツ)_/¯")
            answer = None
            continue

        expertAnswer = ruleManager.give_best_match(userConditions)

        if expertAnswer == "no match":
            print("\tThere is no person with such characteristics, maybe
there is an error in characteristics?")
            print("\tMaybe this type of person even was not introduced in
system DB ¯\_(ツ)_/¯")
            answer = None
            userConditions = []
            continue

        print("person with conditions that you typed is ", expertAnswer)
        userConditions = []
        answer = None

    if answer == "stop program":
        print("\tOk, thanks for using this... code (□° ͜ʖ□° )")
        answer = None
        userConditions = None
        break

    userConditions.append(answer)
    answer = None
```

All user statements can be inserted in the random manner and after collecting all of them, the system will check which rule is the closest to all those statements.

Conclusion:

During this laboratory work was learned how to create a personal expert rule-based system. In comparison with standard systems based on comparing different characteristics, rule-based systems are adaptable to any type of rules, statements and can represent any type of relations between those facts. Such rule-based systems are great considering the fast and

easy process of adding new rules even not requiring any new interaction with code. Such an approach makes the system effective.

Current system can be further enhanced to show some additional information, make interaction with the system more systematic, effective, adaptable to user input, add some deviations considering that some facts can be defined not precisely. Some parts of the code can be optimized to make code smaller, more effective.

Rule-based expert systems are great for making simple AI-like systems that will solve some standard scenarios.

Appendix:

OOP-based approach:

Listing 12: PersonConstruct.py

```
class Person(object):

    def __init__(self, hair, eyes, ears, headshape, height, mouthSize,
language, skinColor):
        self.hair = hair
        self.eyes = eyes
        self.ears = ears
        self.headshape = headshape
        self.height = height
        self.mouthSize = mouthSize
        self.language = language
        self.skinColor = skinColor

    def show_info(self):
        print("Here is a person with " + self.hair + " hair, " + self.eyes
+ " eyes, " + self.headshape + " head shape, " +
              self.height + " height, " + self.mouthSize + " mouth size, "
+ self.language + " is language that person speaks, " +
              self.skinColor + " skin of color")

    def compliance_degree(self, person):
        complianceDegree = 0

        if(isinstance(person, Person)):
            if self.hair == person.hair:
                complianceDegree += 1

            if self.eyes == person.eyes:
                complianceDegree += 1

            if self.ears == person.ears:
```

```
                complianceDegree += 1

            if self.headshape == person.headshape:
                complianceDegree += 1

            if self.height == person.height:
                complianceDegree += 1

            if self.mouthSize == person.mouthSize:
                complianceDegree += 1

            if self.language == person.language:
                complianceDegree += 1

            if self.skinColor == person.skinColor:
                complianceDegree += 1

            return complianceDegree

        else:
            raise TypeError("Incorrect type of object passed for person
verification")
```

Listing 13: TouristTypeDB.py

```
from PersonConstruct import Person


#   for better understanding of how personalities are built here is class
with initialization defining arguments
# correlation
class Genie(Person):
    def __init__(self):
        super().__init__(
            hair="none",
            eyes="foggy",
            ears="none",
            headshape="round",
            height="small",
            mouthSize="none",
            language="high elfian",
            skinColor="blue foggy"
        )

    def show_class(self):
        return "genie"
```

```
class Mandalorian(Person):
    def __init__(self):
        super().__init__("black", "small green", "small round", "round",
"medium", "medium", "universal", "white")

    def show_class(self):
        return "mandalorian"

class Jedi(Person):
    def __init__(self):
        super().__init__("brown", "brown", "small round", "round", "tall",
"small", "force", "brown")

    def show_class(self):
        return "jedi"

class Tatoiner(Person):
    def __init__(self):
        super().__init__("yellow", "blue", "small", "cubic", "medium",
"small", "tatooinean", "black")

    def show_class(self):
        return "tatooiner"

class Loonie(Person):
    def __init__(self):
        super().__init__("white", "black", "small cubic", "cubic",
"medium", "big", "lunar", "white")

    def show_class(self):
        return "loonie"


#   this is the most important class that collects all species in one base
and can be accessed for requesting comparison operation
class PersonalitiesDB:
    def __init__(self):
        self.listOfPersonalities = [Genie(), Mandalorian(), Jedi(),
Tatoiner(), Loonie()]
```

Listing 14: test.py

```
from PersonConstruct import Person
from TouristTypeDB import PersonalitiesDB

#   initialize a database of all possible personalities
```

```
personsDB = PersonalitiesDB()

#   input all characteristics of person (small letters)
print("Input characteristics of the person using words with small letters
(if there is no concrete characteristic print 'unknown' or 'some'")
hair = input("Hair of the person (none, yellow, brown, black, white): ")
eyes = input("Eyes of the person (foggy, small green, brown, blue, black):
")
ears = input("Ears of the person (none, small round, small, small cubic):
")
headshape = input("Person shape of head (round, cubic): ")
height = input("Height of the person (small, medium, tall): ")
mouthSize = input("Person size of mouth (none, medium, small, big): ")
languages = input("Language that this person speaks the most (high elfian,
universal, force, tatooinean, lunar, universal): ")
skinColor = input("Skin color of the person (blue foggy, white, brown,
black): ")

randomPerson = Person(hair, eyes, ears, headshape, height, mouthSize,
languages, skinColor)
randomPerson.show_info()

bestComplianceValue = 0
bestComplianceType = None

for personalityType in personsDB.listOfPersonalities:
    complienceDegree = personalityType.compliance_degree(randomPerson)
    if complienceDegree > bestComplianceValue:
        bestComplianceValue = complienceDegree
        bestComplianceType = personalityType

print("it is a " + bestComplianceType.show_class())
```

Rule-based expert system:

Listing 15: RuleSystem.py

```
"""
Parent class with basic methods, fields and abstract method that must be
implemented by children
"""
class Statement(object):
    #   create object of the class
    def __init__(self):
        self.statementsList = []
        self.response = None

    #   set response field and return object
```

```
    def then(self, statement):
        self.response = statement
        return self

    #   abstract method that must be implemented by child
    def conditions_matching(self):
        raise NotImplementedError("Please Implement this method")

"""
Logical IF class that contains one-statement-rule principle of work (WAS
NOT TESTED, USE WITH CAUTION)
"""
class If(Statement):
    def __init__(self, condition):
        super().__init__()
        self.statementsList.append(condition)

    #   function checks if user characteristics meet statement of the rule
    def conditions_matching(self, userConditions):
        for userCondition in userConditions:
            if userCondition == self.statementsList[0]:
                return True

        return False

    #   give response basing on user characteristics and matching to the
rule
    def give_response(self, userConditions):
        if self.conditions_matching(userConditions):
            return self.response
        else:
            return "no match"

"""
Logical AND class that contains multi-statement-rule with possible inner
OR-rules (this class was mainly used in work)
"""
class And(Statement):
    def __init__(self, conditions):
        super().__init__()
        for condition in conditions:
            self.statementsList.append(condition)

    #   function checks is user characteristics meet statements of the rule
    def conditions_matching(self, userConditions):
        #   shows how many matchings to rule statements are found
        matching_value = 0
```

```
        #   check if there is list of user characteristics or only one user
characteristic
        if isinstance(userConditions, list):

            #   come through all charactestics from user and all statements
of rule
            for userCondition in userConditions:
                for statement in self.statementsList:

                    #   if statement is an inner OR-rule then perform check
                    if isinstance(statement, Or):
                        if statement.conditions_matching(userCondition):
                            matching_value += 1

                    #   increment matching value if statement meets user
condition
                    if userCondition == statement:
                        matching_value += 1

            #   check if all rule statements have been matched
            statementsAmount = len(self.statementsList)
            if matching_value == statementsAmount:
                return True

        #   else if there is only one user characteristic
        else:

            #   come through all statements
            for statement in self.statementsList:
                if isinstance(statement, Or):
                    if statement.conditions_matching(userConditions):
                        matching_value += 1

                    if userConditions == statement:
                        matching_value += 1

            #   check if all rule statements have been matched
            statementsAmount = len(self.statementsList)
            if matching_value == statementsAmount:
                return True

        return False

    #   give response basing on meeting statements
    def give_response(self, userConditions):
        if self.conditions_matching(userConditions):
```

```
            return self.response
        else:
            return "no match"

"""
Logical OR class with multi-statement-rule principle of work, where user
conditions must meet even one statement to match all rule
"""
class Or(Statement):
    def __init__(self, conditions):
        super().__init__()
        for condition in conditions:
            self.statementsList.append(condition)

    #   check how user conditions match all statements of rule
    def conditions_matching(self, userConditions):

        #   check if there is a list of user conditions
        if isinstance(userConditions, list):

            #   iterate through all statements and user conditions
            for userCondition in userConditions:
                for statement in self.statementsList:

                    #   check if there is inner AND rule and check matching
this rule
                    if isinstance(statement, And):
                        if statement.conditions_matching(userConditions):
                            return True

                    #   if user condition matches any of statements
                    if userCondition == statement:
                        return True

        #   check if one user characteristic is meeting even one statement
        else:
            for statement in self.statementsList:
                if isinstance(statement, And):
                    if statement.conditions_matching(userConditions):
                        return True

                if userConditions == statement:
                    return True

        return False

    #   give response basing on matching result
```

```
    def give_response(self, userConditions):
        if self.conditions_matching(userConditions):
            return self.response
        else:
            return "no match"
```

Listing 16: RuleHandler.py

```
from RuleSystem import If, And, Or, Statement

"""
Class responsible for taking all rules and performing check if user
conditions meet all requirements of any rule
"""
class RuleManager(object):
    def __init__(self, listOfRules):
        self.rulesList = listOfRules

    #   return answer basing on user conditions matching any of the
available rules.
    def give_best_match(self, userConditions):
        response = None

        for rule in self.rulesList:
            if isinstance(rule, And) or isinstance(rule, Or) or
isinstance(rule, If) or isinstance(rule, Statement):
                response = rule.give_response(userConditions)
            else:
                raise TypeError("rules db has a non-rule object")

            if response != "no match":
                return response

        return "no match"
```

Listing 17: Rules.py

```
from RuleHandler import RuleManager
from RuleSystem import If, And, Or, Statement

#   This is DB that Rule Manager will use to answer on questions following
defined rules. Rules can be changed, added, removed and so on.
rulesDB = [

    #   Example of a rule defining which person is Genie. Object 'And()'
consists of statements that must be met to consider match.
    # Object 'Or()' consists of statements where one or more must be
satisfied to consider match. Function 'then()' consists of
    # answer that object must provide if all conditions match.
```

```
    And(
        ["has no hair", "has foggy eyes", "has no ears", "has round head",
Or(["speaks force", "speaks high elfian"]), "has blue foggy body"]
    ).then("genie"),

    And(
        [Or(["has black hair", "has brown hair"]), Or(["has green eyes",
"has brown eyes", "has gray eyes"]), "has small round ears", "has round
head", "speaks universal", "has white skin body"]
    ).then("mandalorian"),

    And(
        [Or(["has black hair", "has brown hair"]), "has blue eyes", "has
small round ears", "has round head", Or(["speaks universal", "speaks
force"]), Or(["has white skin body", "has yellow skin body", "has black
skin body"])]
    ).then("jedi"),

    And(
        ["has yellow hair", "has blue eyes", "has medium cubic ears", "has
cubic head", Or(["speaks universal", "speaks tatooinean"]), "has black skin
body"]
    ).then("tatooiner"),

    And(
        ["has white hair", Or(["has white eyes", "has black eyes"]), "has
medium cubic ears", "has round head", Or(["speaks lunar", "speaks
tatooinean", "speaks force"]), "has white skin body"]
    ).then("loonie")
]

#   rule manager take DB of rules for further work
ruleManager = RuleManager(rulesDB)

#   here is a list of characteristics as input for checking how program
works and if all is in place. Order of characteristics plays is not
important
possibleGenieConditions = ["has no hair", "has foggy eyes", "has no ears",
"has round head", "speaks high elfian", "has blue foggy body"]
possibleMandalorianConditions = ["has brown hair", "has green eyes", "has
small round ears", "has round head", "speaks universal", "has white skin
body"]
possibleJediConditions = ["has brown hair", "has blue eyes", "has small
round ears", "has round head", "speaks force", "has white skin body"]
possibleTatooinerConditions = ["has yellow hair", "has blue eyes", "has
medium cubic ears", "has cubic head", "speaks tatooinean", "has black skin
body"]
```

```
possibleLoonieConditions = ["has white hair", "has white eyes", "has medium
cubic ears", "has round head", "speaks lunar", "has white skin body"]

#   rule manager show its suggestions over defined above characteristics
and rules
print("Possible genie was considered by system as ",
ruleManager.give_best_match(possibleGenieConditions))
print("Possible mandalorian was considered by system as ",
ruleManager.give_best_match(possibleMandalorianConditions))
print("Possible jedi was considered by system as ",
ruleManager.give_best_match(possibleJediConditions))
print("Possible tatooiner was considered by system as ",
ruleManager.give_best_match(possibleTatooinerConditions))
print("Possible loonie was considered by system as ",
ruleManager.give_best_match(possibleLoonieConditions))

#   here is command line interface for typing user conditions of new person
and getting result from program.
print("\tIf you want to check program, then type in your conditions and
then program will give response basing on your input and rules defined by
experts.")
print("\tType your condition (if there is no new condition, then type
'none')")
answer = None
userConditions = []

#   always listen for input
while True:
    answer = input(">>>\t")

    #   this case means that user either does not know characteristics of
person or ended their input
    if answer == "none" or answer == "no more":

        #   this means there are no characteristics from user
        if len(userConditions) == 0:
            print("\tWell, there is no condition from you, I can't even try
to guess who this can be ¯\_(ツ)_/¯")
            answer = None
            continue

        #   try to give suggestion to user
        expertAnswer = ruleManager.give_best_match(userConditions)

        #   give some additional data to user if there is no match
        if expertAnswer == "no match":
            print("\tThere is no person with such characterstics, maybe
```

```
there is an error in characteristics?")
            print("\tMaybe this type of person even was not introduced in
system DB ¯\_(ツ)_/¯")
            answer = None
            userConditions = []
            continue

        #   if all is ok, then just show answer to user
        print("person with conditions that you typed is ", expertAnswer)
        userConditions = []
        answer = None

    #   in this case just stop program execution
    if answer == "stop program":
        print("\tOk, thanks for using this... code (□° ͜ʖ□° )")
        answer = None
        userConditions = None
        break

    #   if user has not finished typing new characteristics, then append
current to the list of all characteristics
    userConditions.append(answer)
    answer = None
```

Links:

GitHub repository with first lab -

https://github.com/filpatterson/python_fia_codes#rule-based-expert-system