

LOGIC Programming

(pyDatalog in Python)

Dr. Indrajit Pan

Associate Professor, Dept. of IT

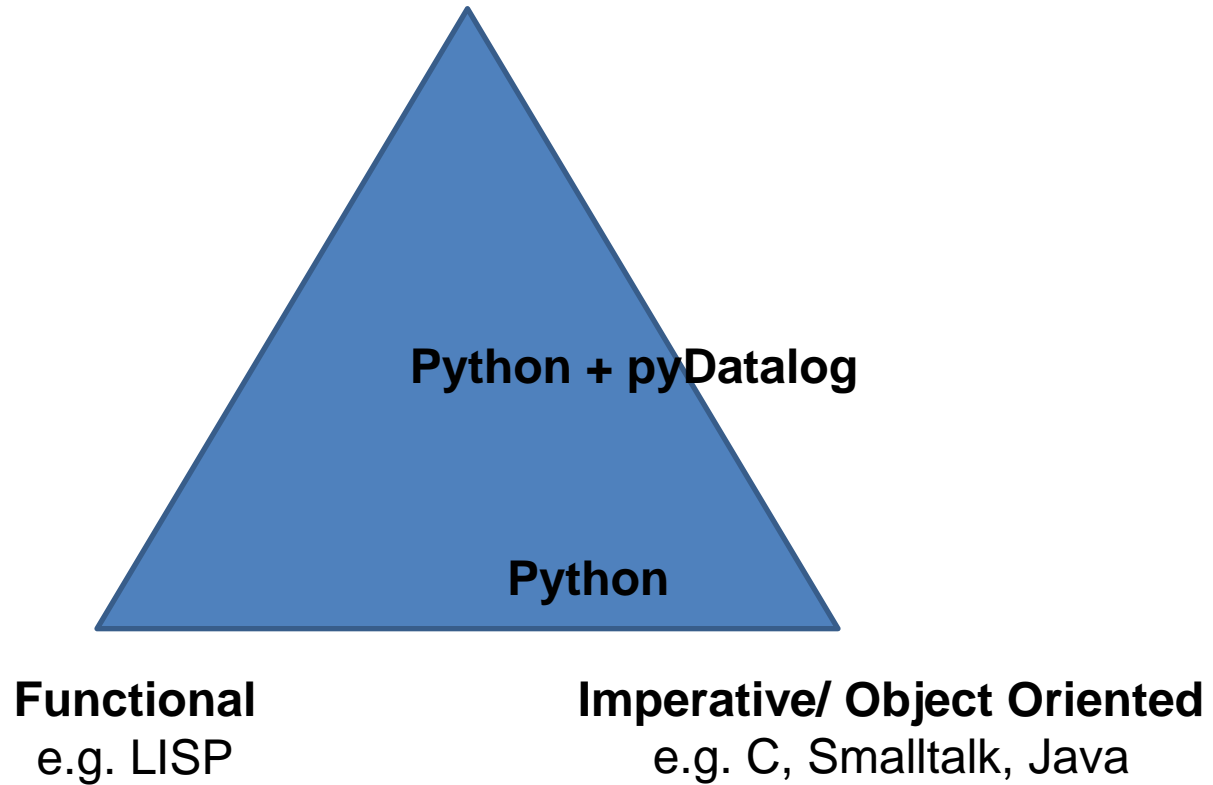
RCCIIT, Kolkata

Logic Programming

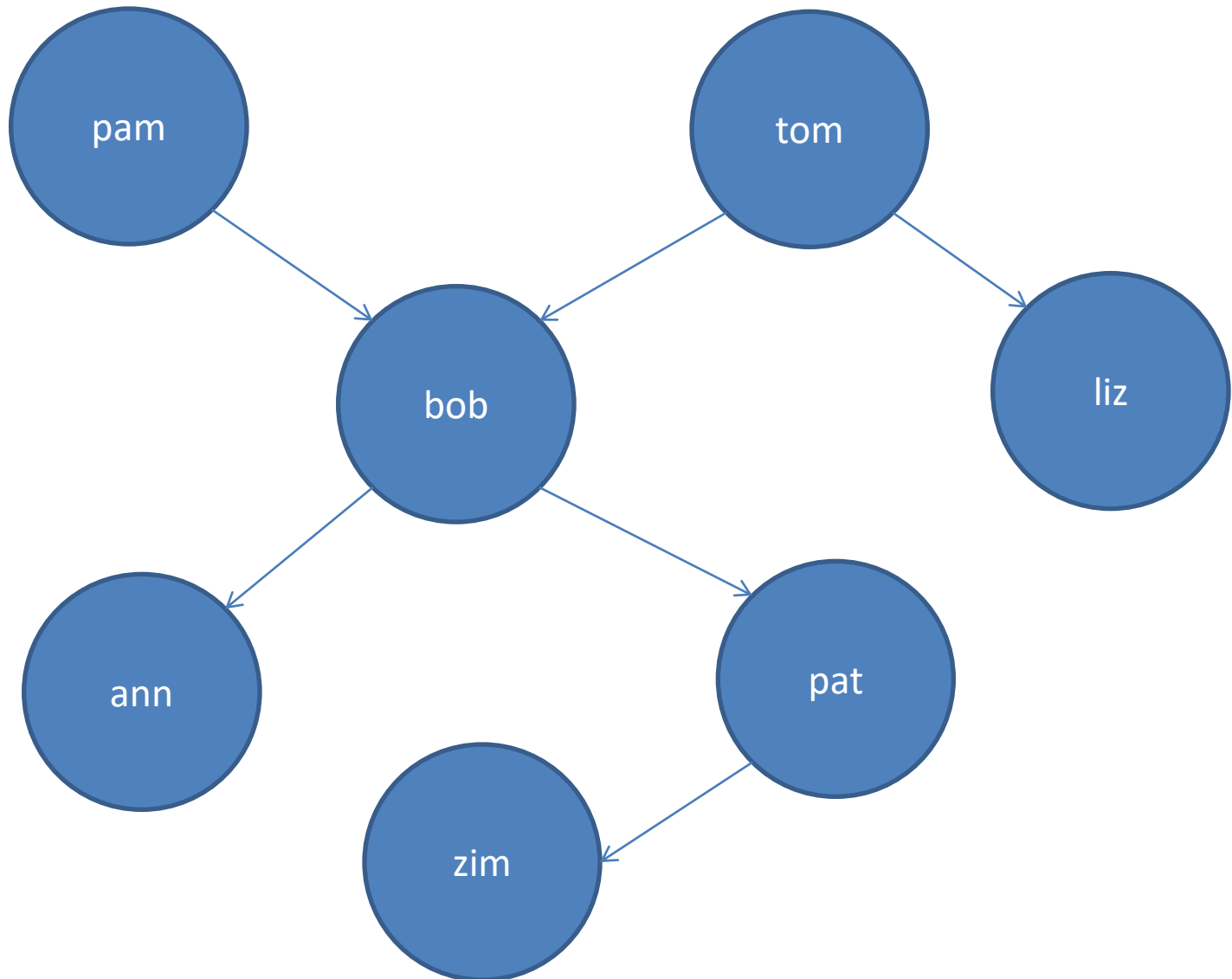
- A logic combines syntax and semantics.
- This also empowers decision making with inference rules
- **Syntax** – Rules about how to form formulas
- **Semantics** – Meaning carried by formulas, mainly terms of logical consequences
- **Inference rules** – Describe correct ways to derive conclusions

Logic Programming

Logic Programming
e.g. PROLOG, SQL



Logic Programming - Example



Logic Programming - Example

- Relational clauses -

- parent(tom, bob)
- parent(pam, bob)
- parent(tom, liz)
- parent(bob, ann)
- parent(bob, pat)
- parent(pat, zim)

Execute –

? parent(pam, bob) → Yes (True)

? parent(pam, pat) → No (False)

? parent(tom, X) → X = bob

(X will be instantiated by bob)

Logic Programming - Example

Grandparent clause

? parent(Y,X) and parent(X, zim)

male(tom)

female(pam)

male(bob)

female(liz)

female(ann)

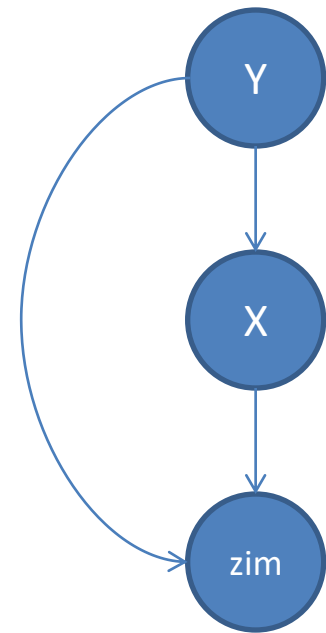
female(pat)

male(zim)

mother(X, Y) :- parent (X,Y) and female (X)

offspring(X,Y) :- parent (Y, X)

sister(X,Y) :- parent(Z,X) and parent(Z,Y) and female(X)



pyDatalog

[Installation Command: `pip install pyDatalog`**]**

- It's an open source project
- pyDatalog adds the logic programming paradigm to Python
- Logic programmers can now use extensive standard libraries of Python
- Datalog is truly declarative language derived from PROLOG

pyDatalog Features

- pyDatalog is a powerful language with very few syntactic elements, mostly coming from Python
- Facts can be inserted in datalog knowledge base and query it
- Logic clauses can also be used to query any relational database
- Attributes of python clauses can be defined through logic clauses and use logic queries on python clauses
- Logic functions can be defined (e.g. $p[X] == Y$) , i.e. logic predicate with unicity
- Arithmetic literals can be used ($X == Y+1$). An expression can contain Python functions and lambda expressions
- A literal can be negated in the body of a clause : $\sim p(X)$

pyDatalog Applications

- Simulating intelligent behaviour
- Querying complex set of related information
- Performing recursive algorithms

pyDatalog – Core Technology

Core technology in pyDatalog	Benefits	Sample applications
The resolution engine determines the best sequence of clauses to use to reach a goal	Spreadsheet-style programming : faster development; fewer bugs, easier to read afterwards	Rule-based models with many input and output, e.g. expert system for many applications, intelligent agent in games or automated assistants
The resolution engine can resolve recursive algorithm	Easy to write queries on hierarchical structure	Reporting with hierarchical reference data such as organisational structure
The same clause can solve various mixes of known and unknown parameters	Maximize code reuse : shorter program, simpler query language than SQL	Cross-database queries, data integration
Intermediate results are memoized	Improved speed by avoiding duplicate work	Business logic in 3-tier architecture

Working with pyDatalog

1. First step is to import pyDatalog

```
from pyDatalog import pyDatalog
```

2. **[Working with Variables]** Now declare variables to be used in upper case letters

```
pyDatalog.create_terms('X,Y')
```

– Variables appear in logic queries, which return a printable result

```
# give me all the X so that X is 1
```

```
print(X==1)
```

Working with pyDatalog

- Queries can contain several variables and several criteria ('&' is read 'and'):

give me all the X and Y so that X is True and Y is False

```
print((X==True) & (Y==False))
```

Note the parenthesis around each equality: they are required to avoid confusion with

(X==(True & Y)==False)

Working with pyDatalog

- Some queries return an empty result :

give me all the X that are both True and False

print((X==True) & (X==False))

- Besides numbers and Booleans, variables can represent strings. Furthermore, queries can contain python expressions:

give me all the X and Y so that X is a name and Y is 'Hello ' followed by the first letter of X

print((X==input('Please enter your name : ')) & (Y=='Hello ' + X[0]))

Working with pyDatalog

- pyDatalog has no symbolic resolver (yet)! If a variable in an expression is not bound, the query returns an empty solution

give me all the X and Y so that Y is 1 and Y is X+1

print((X==1) & (Y==X+1))

- Variables can also represent (nested) tuples, which can participate in an expression and be sliced (0-based).

print((X==(1,2)+(3,)) & (Y==X[2]))

Working with pyDatalog

- To use your own functions in logic expressions, define them in Python, then ask pyDatalog to create logical terms for them:

```
from pyDatalog import pyDatalog
import math
pyDatalog.create_terms('X,Y')
pyDatalog.create_terms('twice')
def twice(a):
    return a+a
print((X==1) & (Y==twice(X)))
```

Note that X must be bound before calling the function

Working with pyDatalog

- Similarly, pyDatalog variables can be passed to functions in the Python standard library:

```
from pyDatalog import pyDatalog
import math
pyDatalog.create_terms('X,Y')
X=9
print(Y==math.sqrt(X))
```


Logic Functions

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z,
    salary,tax_rate,
    tax_rate_for_salary_above, net_salary')
salary['Amit'] = 60
salary['Akash'] = 110
print(salary[X]==Y) # function can be queried
print(salary['Akash']==Y)
    # func. has one value for given argument
print(salary[X]==110) # func. queried by value
```

Observations:

- Logic function name such as 'salary', begins with lower case
- Function defines one value for given argument

Logic Functions

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z,      salary,
                        tax_rate,    tax_rate_for_salary_above,
                        net_salary')
salary['Amit'] = 60
salary['Akash'] = 110
print((salary[X]==Y) & ~(Y==110))
                        #negation of criteria
+ (tax_rate[None]==0.33)
                        #global tax rate
print((Z==salary[X] * (1-tax_rate[None])))
# In this case, X is bound by salary[X], so the expression
can be evaluated.
```

Logic Functions

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z, salary, tax_rate,
    tax_rate_for_salary_above, net_salary')
salary['Amit'] = 60
salary['Akash'] = 110
+(tax_rate[None]==0.33)                #global tax rate
net_salary[X] = salary[X]*(1-tax_rate[None])
    #function can also be defined by a clause
print(net_salary[X]==Y)
# such a function can be queried by value
print(net_salary['Amit']==Y)
print(net_salary[X]>50)
```

Logic Functions

- Let's now define a progressive tax system: the tax rate is 33 % by default, but 50% for salaries above 100
- '<=' is the important token in the statements above : it is read 'if'

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z, salary, tax_rate,
    tax_rate_for_salary_above, net_salary')
salary['Amit'] = 60
salary['Akash'] = 110
(tax_rate_for_salary_above[X] == 0.33) <= (X >= 0)
(tax_rate_for_salary_above[X] == 0.50) <= (X >= 100)
print(tax_rate_for_salary_above[70]==Y)
print
print(tax_rate_for_salary_above[150]==Y)
```

Logic Functions

- Net salary newly defined on the basis of tax rate

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z, salary, tax_rate,
    tax_rate_for_salary_above, net_salary')
salary['Amit'] = 60
salary['Akash'] = 110
(tax_rate_for_salary_above[X] == 0.33) <= (X >= 0)
(tax_rate_for_salary_above[X] == 0.50) <= (X >= 100)
net_salary[X] = salary[X]*(1-
    tax_rate_for_salary_above[salary[X]])
print(net_salary[X]==Y)
```

Program for Factorial

- This short notation, together with the fact that functions can be defined in any order, makes writing a pyDatalog program as easy as creating a spreadsheet.
- To illustrate the point, this definition of Factorial cannot be any clearer !

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('factorial, N')
```

```
factorial[N] = N*factorial[N-1]
factorial[1] = 1
```

```
print(factorial[3]==N)
```

Aggregate functions

- Aggregate functions are a special type of functions. Let's first create the data we need to illustrate them.

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,manager,
    count_of_direct_reports')
+(manager['Mary'] == 'John')
+(manager['Sam'] == 'Mary')
+(manager['Tom'] == 'Mary')
(count_of_direct_reports[X]==len_(Y)) <= (manager[Y]==X)
print(count_of_direct_reports['Mary']==Y)
```

- A basic aggregation is to count the number of results, using len_.
- pyDatalog searches all possible solutions for manager['Mary']==Y, then counts the number of Y.

Aggregate functions

- **len_ (P[X]==len_(Y)) <= body** : P[X] is the count of values of Y (associated to X by the body of the clause)
- **sum_ (P[X]==sum_(Y, for_each=Z)) <= body** : P[X] is the sum of Y for each Z. (Z is used to distinguish possibly identical Y values)
- **min_, max_ (P[X]==min_(Y, order_by=Z)) <= body** : P[X] is the minimum (or maximum) of Y sorted by Z.
- **tuple_ (P[X]==tuple_(Y, order_by=Z)) <= body** : P[X] is a tuple containing all values of Y sorted by Z.
- **concat_ (P[X]==concat_(Y, order_by=Z, sep=',')) <= body** : same as 'sum' but for string. The strings are sorted by Z, and separated by ','.
- **rank_ (P[X]==rank_(group_by=Y, order_by=Z)) <= body** : P[X] is the sequence number of X in the list of Y values when the list is sorted by Z.
- **running_sum_ (P[X]==running_sum_(N, group_by=Y, order_by=Z)) <= body** : P[X] is the sum of the values of N, for each Y that are before or equal to X when Y's are sorted by Z.

Literals and Set

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z, works_in,
    department_size, manager, indirect_manager,
    count_of_indirect_reports')

# facts to the set
+ works_in('Mary', 'Production')
+ works_in('Sam', 'Marketing')

+ works_in('John', 'Production')
+ works_in('John', 'Marketing')

# give me all the X that work in Marketing
print(works_in(X, 'Marketing'))
```

Literals and Set (lg12.py)

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z, works_in, department_size, manager,
    indirect_manager, count_of_indirect_reports')

# facts to the set
+ works_in('Mary', 'Production')
+ works_in('Sam', 'Marketing')
+ works_in('John', 'Production')
+ works_in('John', 'Marketing')
+ (manager['Mary'] == 'John')
+ (manager['Sam'] == 'Mary')
+ (manager['Tom'] == 'Mary')

# give me all the X that work in Marketing
print(works_in(X, 'Marketing'))

# one of the indirect manager of X is Y, if the (direct) manager of X is Y

indirect_manager(X,Y) <= (manager[X] == Y)
# another indirect manager of X is Y, if there is a Z so that the manager of
  X is Z, and an indirect manager of Z is Y
indirect_manager(X,Y) <= (manager[X] == Z) & indirect_manager(Z,Y)
print(indirect_manager('Sam',Y))
```

Literals and Set (lg13.py)

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z, works_in, department_size,
    manager, indirect_manager, count_of_indirect_reports')
+ works_in('Mary', 'Production')
+ works_in('Sam', 'Marketing')
+ works_in('John', 'Production')
+ works_in('John', 'Marketing')
+ (manager['Mary'] == 'John')
+ (manager['Sam'] == 'Mary')
+ (manager['Tom'] == 'Mary')
print(works_in(X, 'Marketing'))
indirect_manager(X,Y) <= (manager[X] == Y)
indirect_manager(X,Y) <= (manager[X] == Z) &
    indirect_manager(Z,Y)
print(indirect_manager('Sam',Y))
(count_of_indirect_reports[X]==len_(Y)) <=
    indirect_manager(Y,X)
print(count_of_indirect_reports['John']==Y)
```

Recursive Algorithms

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z,link,can_reach')

# there is a link between node 1 and node 2
+link(1,2)
+link(2,3)
+link(2,4)
+link(2,5)
+link(5,6)
+link(6,7)
+link(7,2)
# links are bi-directional
link(X,Y) <= link(Y,X)
# can Y be reached from X ?
can_reach(X,Y) <= link(X,Y) # direct link via Z
can_reach(X,Y) <= link(X,Z) & can_reach(Z,Y) & (X!=Y)

print (can_reach(1,Y))
```

Logic Programming – Example 1

- Relational clauses -

parent(tom, bob)

parent(pam, bob)

parent(tom, liz)

parent(bob, ann)

parent(bob, pat)

parent(pat, zim)

male(tom)

female(pam)

male(bob)

female(liz)

female(ann)

female(pat)

male(zim)

Logic Programming – Example 1

Rule Set:

grandparent(X,Y) :- parent(X,Z) and parent(Z,Y)

mother(X, Y) :- parent (X,Y) and female (X) and (X!=Y)

offspring(X,Y) :- parent (Y, X)

sister(X,Y) :- parent(Z,X) and parent(Z,Y) and female(X)

Queries to be performed:

Is pam parent of bob?

Whose parent is tom?

Who is grandparent of zim?

Show details of sisters.

Show all offspring relations.

Answer (lg16.py)

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z,parent,grandparent,mother,male,female,of
    fspring,sister')

# Relational facts
+parent('tom', 'bob')
+parent('pam', 'bob')
+parent('tom', 'liz')
+parent('bob', 'ann')
+parent('bob', 'pat')
+parent('pat', 'zim')
+male('tom')
+female('pam')
+male('bob')
+female('liz')
+female('ann')
+female('pat')
+male('zim')
```

Answer - Continued

```
# inference rules
grandparent(X,Y) <= parent(X,Z) & parent(Z,Y)
mother(X,Y) <= parent (X,Y) & female (X)
offspring(X,Y) <= parent (Y,X)
sister(X,Y) <= parent(Z,X) & parent(Z,Y) & female(X) & (X!=Y)

# Queries
print("Is pam parent of bob?")
print(parent(pam,bob) )
print("-----")
print("Whose parent is tom?")
print(parent(tom,X) )
print("-----")
print("Who is grandparent of zim?")
print(grandparent(Y,zim) )
print("-----")
print("Show details of sisters.")
print(sister(X,Y) )
print("-----")
print("Show all offspring relations.")
print(offspring(X,Y) )
```


Logic Programming - Example

Rule Set:

grandparent(X,Y) :- parent(X,Z) and parent(Z,Y)

mother(X, Y) :- parent (X,Y) and female (X) and (X!=Y)

offspring(X,Y) :- parent (Y, X)

sister(X,Y) :- parent(Z,X) and parent(Z,Y) and female(X)

Queries to be performed:

Is pam parent of bob?

Whose parent is tom?

Who is grandparent of zim?

Show details of sisters.

Show all offspring relations.

Logic Programming – Example 2

(Resolution)

Sentence:

All lecturers are determined. Anyone who is determined and intelligent will give good service. Mary is an intelligent lecturer. ***Show that*** Mary will give good service

Logic Programming – Example 2

(Resolution)

Rules in FOL	Resolution in FOL
[R1] $\forall x: \text{lecturer}(x) \rightarrow \text{determined}(x)$	good_service (Mary)
[R2] $\forall x: \text{determined}(x) \wedge \text{intelligent}(x) \rightarrow \text{good_service}(x)$	
[R3] $\text{intelligent}(\text{Mary}) \wedge \text{lecturer}(\text{Mary})$	

Logic Programming – Example 2

(Resolution) - [lecturer.py]

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,determined,lecturer,intelligent,good_service')
# Relational facts
+lecturer('Mary')
+intelligent('Mary')
# inference rules
determined(X) <= lecturer(X)
good_service(X) <= determined(X) & intelligent (X)
# Queries
print("\nDoes Mary render good service?\n")
print(good_service(X))
print("\nResolution -----")
print(good_service('Mary'))
```

Logic Programming – Example 3

(Resolution)

Sentence:

Ravi likes all kinds of food. Apple and chicken are food. Anything anyone eats and not killed is food. Ajay eats peanut and still alive. ***Show that Ravi likes peanut.***

Logic Programming – Example 3

(Resolution)

Rules in FOL	Resolution in FOL
[R1] $\forall x: \text{food}(x) \rightarrow \text{likes}(\text{Ravi}, x)$	likes (Ravi, peanut)
[R2] $\text{food}(\text{Apple}) \wedge \text{food}(\text{chicken})$	
[R3] $\forall x, \forall y: \text{eats}(y, x) \wedge \text{notkilled}(y) \rightarrow \text{food}(x)$	
[R4] $\text{eats}(\text{Ajay}, \text{peanut}) \wedge \text{alive}(\text{Ajay})$	
[R5] $\forall x: \text{alive}(y) \rightarrow \text{notkilled}(y)$ [derived rule]	

Logic Programming – Example 3

(Resolution) - [food.py]

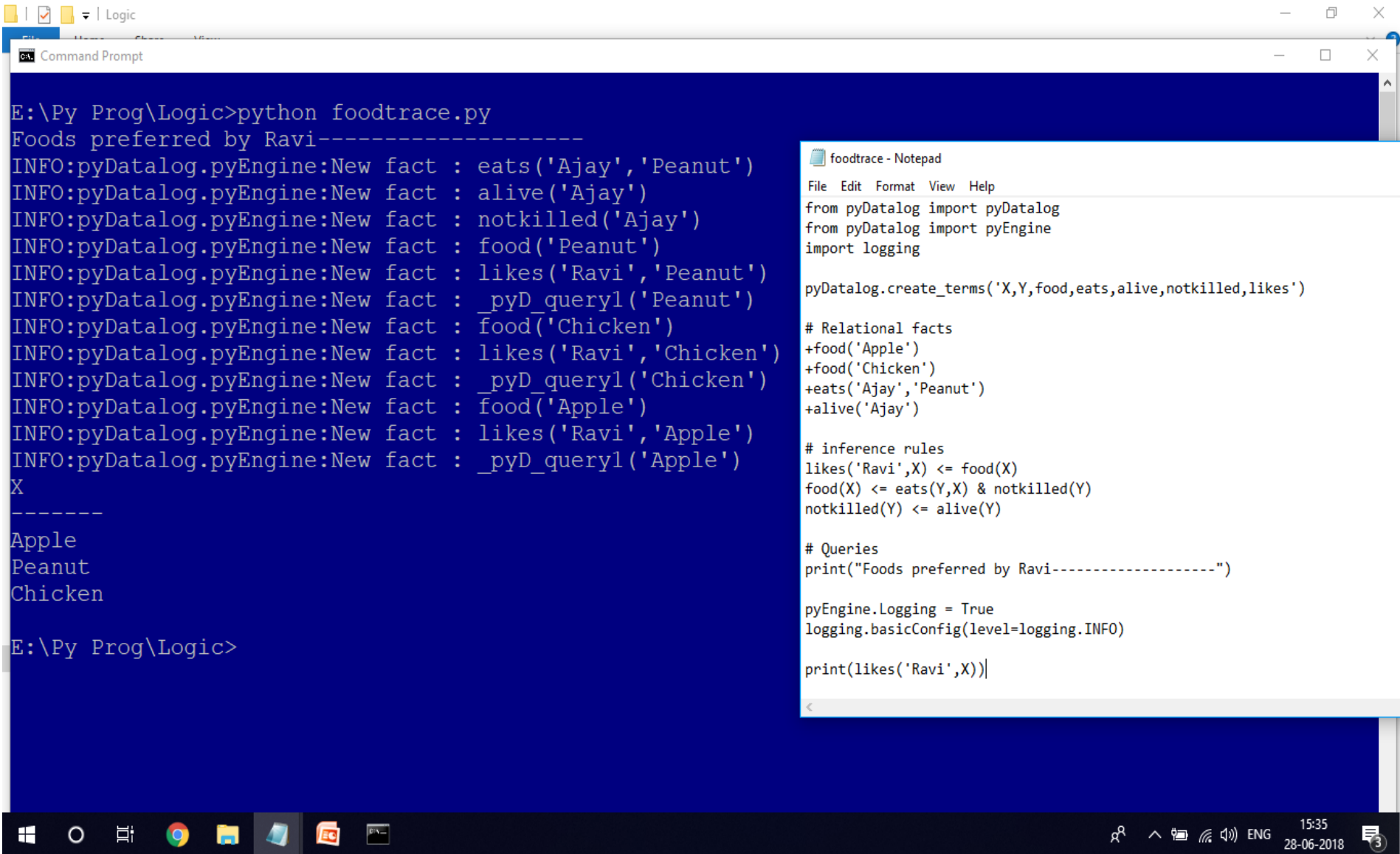
```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,food,eats,alive,notkilled,likes')
# Relational facts
+food('Apple')
+food('Chicken')
+eats('Ajay','Peanut')
+alive('Ajay')
# inference rules
likes('Ravi',X) <= food(X)
food(X) <= eats(Y,X) & notkilled(Y)
notkilled(Y) <= alive(Y)
eats('Rita',X) <= eats('Ajay',X)
```

Continued .. – Example 3

(Resolution) - [food.py]

```
# Queries
print("Does Ravi like peanut?")
print(likes('Ravi',X) & (X=='Peanut'))
print("\nResolution -----")
print(likes('Ravi','Peanut'))
```


Logic Programming - [Trace of Resolution]



```
E:\Py Prog\Logic>python foodtrace.py
Foods preferred by Ravi-----
INFO:pyDatalog.pyEngine:New fact : eats('Ajay','Peanut')
INFO:pyDatalog.pyEngine:New fact : alive('Ajay')
INFO:pyDatalog.pyEngine:New fact : notkilled('Ajay')
INFO:pyDatalog.pyEngine:New fact : food('Peanut')
INFO:pyDatalog.pyEngine:New fact : likes('Ravi','Peanut')
INFO:pyDatalog.pyEngine:New fact : _pyD_query1('Peanut')
INFO:pyDatalog.pyEngine:New fact : food('Chicken')
INFO:pyDatalog.pyEngine:New fact : likes('Ravi','Chicken')
INFO:pyDatalog.pyEngine:New fact : _pyD_query1('Chicken')
INFO:pyDatalog.pyEngine:New fact : food('Apple')
INFO:pyDatalog.pyEngine:New fact : likes('Ravi','Apple')
INFO:pyDatalog.pyEngine:New fact : _pyD_query1('Apple')
X
-----
Apple
Peanut
Chicken

E:\Py Prog\Logic>
```

```
foodtrace - Notepad
File Edit Format View Help
from pyDatalog import pyDatalog
from pyDatalog import pyEngine
import logging

pyDatalog.create_terms('X,Y,food,eats,alive,notkilled,likes')

# Relational facts
+food('Apple')
+food('Chicken')
+eats('Ajay','Peanut')
+alive('Ajay')

# inference rules
likes('Ravi',X) <= food(X)
food(X) <= eats(Y,X) & notkilled(Y)
notkilled(Y) <= alive(Y)

# Queries
print("Foods preferred by Ravi-----")

pyEngine.Logging = True
logging.basicConfig(level=logging.INFO)

print(likes('Ravi',X))
```