PyKE
Python Knowledge Engine

Intelligent Systems II - 2022/2023

Filipe Gonçalves, 98083
Gonçalo Machado, 98359

TABLE OF CONTENTS

01
Introduction
Contextualization and basic ideas

02
Knowledge Bases
PyKE's definition of knowledge bases and how to create them

03
Tutorial
Some examples and proper explanation

04
Why PyKE ?
PyKE vs Prolog

01

Introduction

What is PyKE ?
What should we know about PyKE ?

What is PyKE ?

The **Python Knowledge Engine (PyKE)** is a powerful knowledge management system based on the Python programming language.

PyKE uses **Knowledge Interchange Format (KIF)**, which is a knowledge representation language that is a standard way of expressing knowledge in a formal way.

PyKE includes both a **forward-chaining** and **backward-chaining** inference engines.

PyKE is an open source software, but has received no updates after 2014, and is compatible only with Python 2.6, 2.7 and 3.1.

Pyke Concepts

Statements/Facts: These are pieces of information that are used to demonstrate how entities are related to each other.

family.son_of(Bruce, Thomas, Norma)

Patterns and Matching: Patterns are the arguments of statements. They can be:

- **Literal patterns** - Data values that only match themselves (E.g.: Bruce)
- **Pattern variables** - Match anything including other pattern variables, have values bounded to them and always start with $. (E.g.: $x, $son)
- **Anonymous variables** - Special pattern variables that don't have values bounded to them and start with _. (E.g.: $_)
- **Tuple patterns** - which are a series of patterns surrounded by parenthesis. (E.g.: (Bruce, $father, $_))

Pyke Concepts

Rules: These are “if-then” relations, in which we define conditions where statements are true based on facts.

Rules can be:

- **Forward-Chaining**: These rules are used to infer new facts from existing ones and from other facts created by forward chaining.
- **Backwards-Chaining**: These rules are used to prove or disprove a statement, using the known facts and rules.

Knowledge Base: It’s how **PyKE** organizes and stores knowledge, which is then used to determine the validity of statements.

02

Knowledge Bases

What are the different types of Knowledge Bases ?
How do we create them ?

Knowledge Bases - Fact Bases

A knowledge base that contains simple lists of facts is a **Fact Base**.

These determine if a statement is **True** by checking if it exists in the list of known facts.

There are 2 types of facts:

- **Universal Facts**: These facts are never deleted and are typically added once at program startup.
- **Case Specific Facts**: These facts are temporary facts that are usually generated by **Forward-Chaining** rules, and are deleted when an engine reset is done.

Knowledge Bases - Fact Bases

Example - family.kfb

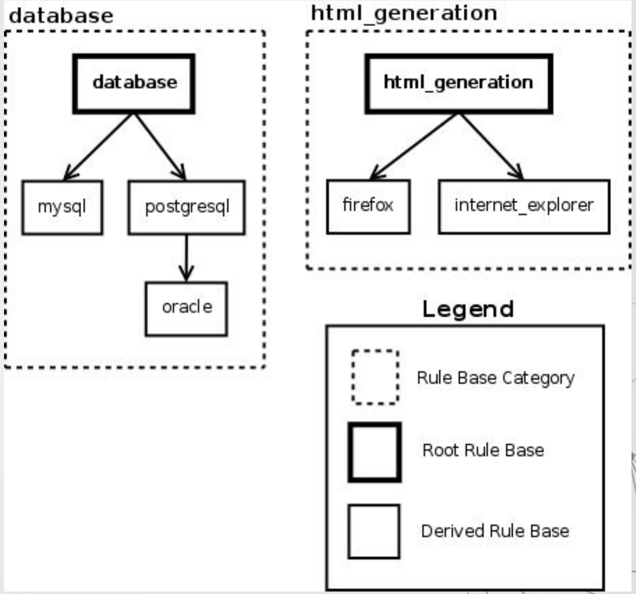
```
# family.daughter_of(daughter, father, mother)
# family.son_of(son, father, mother)

son_of(arthur2, arthur1, bertha_o)
daughter_of(helen, arthur1, bertha_o)
daughter_of(roberta, arthur1, bertha_o)

daughter_of(gladis, john, bertha_c)
daughter_of(sarah_r, john, bertha_c)
daughter_of(alice, marshall1, bertha_c)
son_of(edmond, marshall1, bertha_c)
daughter_of(kathleen, marshall1, bertha_c)
daughter_of(darleen, marshall1, bertha_c)
daughter_of(birdie, marshall1, bertha_c)
son_of(marshall2, marshall1, bertha_c)
```

Knowledge Bases - Rule Bases

A collection of rules is called a **Rule Base**. A single rule base may contain both **Forward-Chaining** and **Backwards-Chaining** rules.

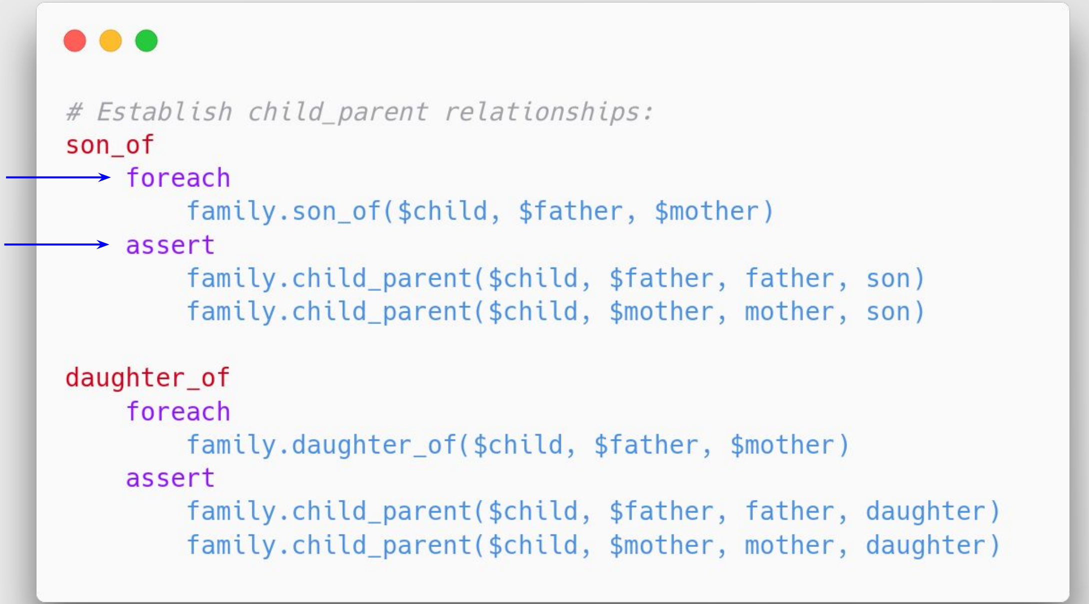
Rule Bases have 3 main capabilities:

- **Activation:** A rule base is only available in a program when explicitly activated.
- **Categories**: Every rule base belongs to a category and each category can only have one active rule base.
- **Inheritance**: Rule bases within the same category can share rules by using a **root Rule Base**.



Knowledge Bases - Rule Bases

Example - family.krb



```
# Establish child_parent relationships:
son_of
    foreach
        family.son_of($child, $father, $mother)
    assert
        family.child_parent($child, $father, father, son)
        family.child_parent($child, $mother, mother, son)

daughter_of
    foreach
        family.daughter_of($child, $father, $mother)
    assert
        family.child_parent($child, $father, father, daughter)
        family.child_parent($child, $mother, mother, daughter)
```

If

then

Knowledge Bases - Question Bases and Special Bases

A **Question Base** is composed of questions of various types (yes/no, multiple choice, true/false) and may be parameterized, where different parameter values are treated as different questions.

Questions can also have reviews attached to them to give feedback to the user depending on the answer given.

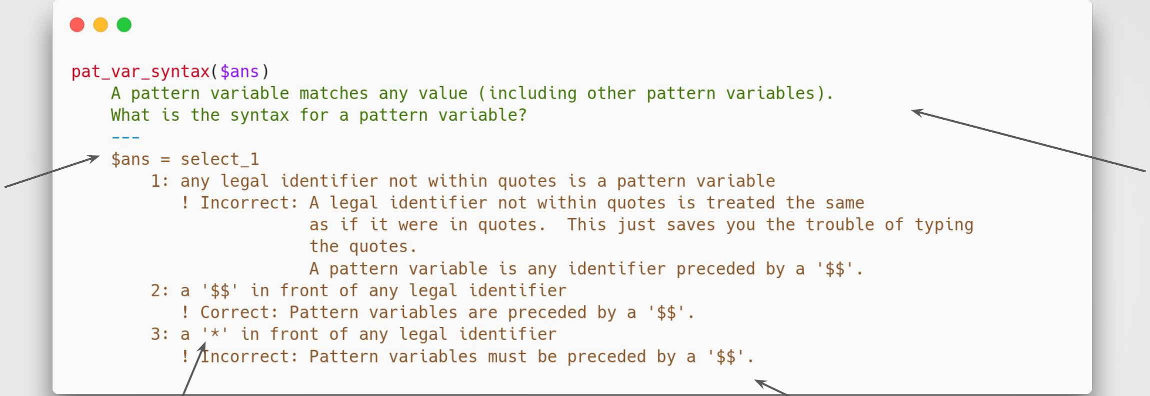
A **Special Base** is a unique knowledge base that contains helpers that determine the validity of a statement in interesting ways. These helpers are:

- **Claim_goal**: This works like the Prolog cut operator(!).
- **Check_command**, **command**, **General_command**:These are used to interact with the host computer that is running the **PyKE** program and they each give different kinds of output.

Knowledge Bases - Questions

Example

questions.kqb



```
pat_var_syntax($ans)
    A pattern variable matches any value (including other pattern variables).
    What is the syntax for a pattern variable?
    ---
    $ans = select_1
        1: any legal identifier not within quotes is a pattern variable
           ! Incorrect: A legal identifier not within quotes is treated the same
                        as if it were in quotes.  This just saves you the trouble of typing
                        the quotes.
                        A pattern variable is any identifier preceded by a '$$'.
        2: a '$$' in front of any legal identifier
           ! Correct: Pattern variables are preceded by a '$$'.
        3: a '*' in front of any legal identifier
           ! Incorrect: Pattern variables must be preceded by a '$$'.
```

Text for the question

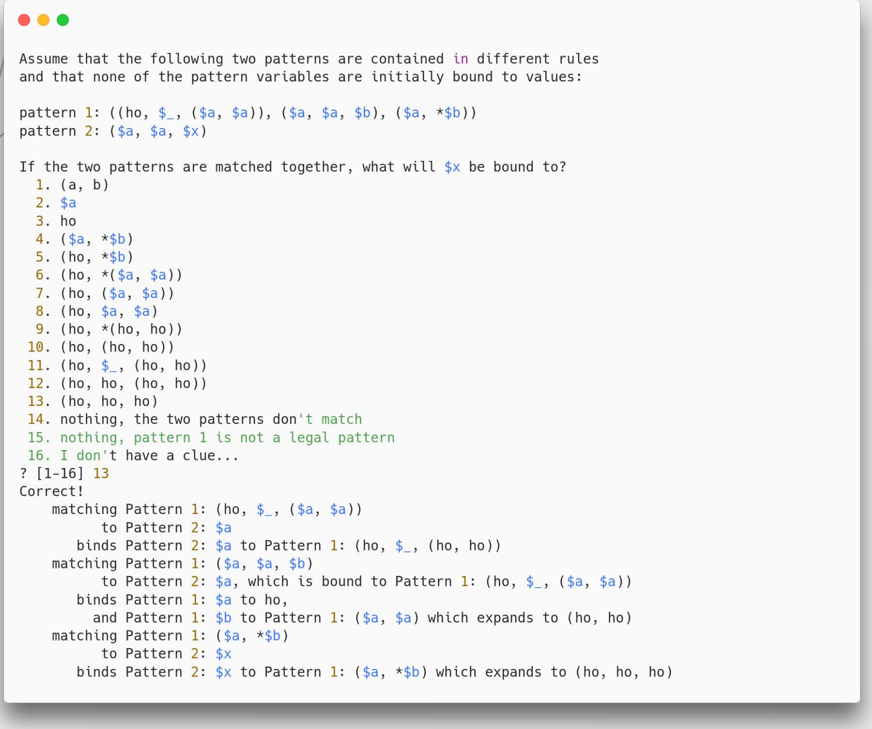
“Switch case” for the answer

Text for the response

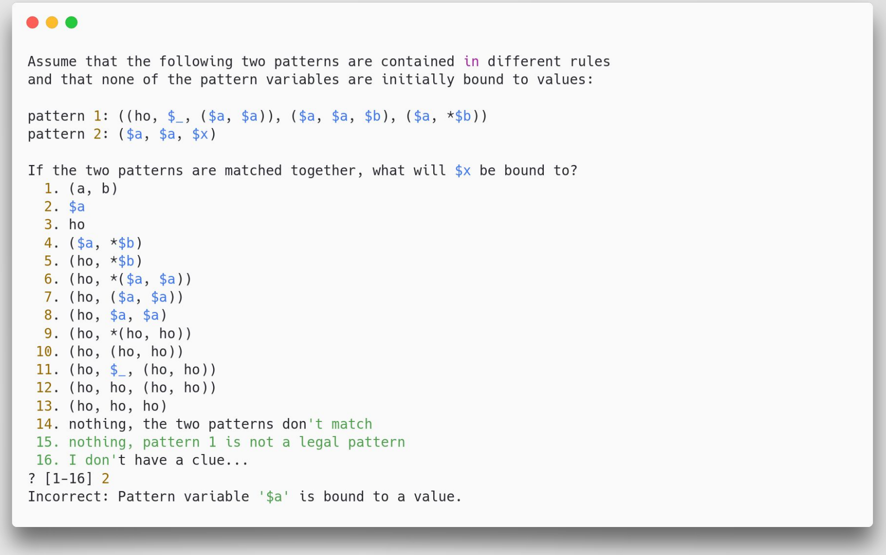
Text for the response feedback

Knowledge Bases - Questions

Exam Example



```
Assume that the following two patterns are contained in different rules
and that none of the pattern variables are initially bound to values:

pattern 1: ((ho, $_, ($a, $a)), ($a, $a, $b), ($a, *$b))
pattern 2: ($a, $a, $x)

If the two patterns are matched together, what will $x be bound to?
  1. (a, b)
  2. $a
  3. ho
  4. ($a, *$b)
  5. (ho, *$b)
  6. (ho, *($a, $a))
  7. (ho, ($a, $a))
  8. (ho, $a, $a)
  9. (ho, *(ho, ho))
 10. (ho, (ho, ho))
 11. (ho, $_, (ho, ho))
 12. (ho, ho, (ho, ho))
 13. (ho, ho, ho)
 14. nothing, the two patterns don't match
 15. nothing, pattern 1 is not a legal pattern
 16. I don't have a clue...
? [1-16] 13
Correct!
    matching Pattern 1: (ho, $_, ($a, $a))
          to Pattern 2: $a
       binds Pattern 2: $a to Pattern 1: (ho, $_, (ho, ho))
    matching Pattern 1: ($a, $a, $b)
          to Pattern 2: $a, which is bound to Pattern 1: (ho, $_, ($a, $a))
       binds Pattern 1: $a to ho,
         and Pattern 1: $b to Pattern 1: ($a, $a) which expands to (ho, ho)
    matching Pattern 1: ($a, *$b)
          to Pattern 2: $x
       binds Pattern 2: $x to Pattern 1: ($a, *$b) which expands to (ho, ho, ho)
```



```
Assume that the following two patterns are contained in different rules
and that none of the pattern variables are initially bound to values:

pattern 1: ((ho, $_, ($a, $a)), ($a, $a, $b), ($a, *$b))
pattern 2: ($a, $a, $x)

If the two patterns are matched together, what will $x be bound to?
  1. (a, b)
  2. $a
  3. ho
  4. ($a, *$b)
  5. (ho, *$b)
  6. (ho, *($a, $a))
  7. (ho, ($a, $a))
  8. (ho, $a, $a)
  9. (ho, *(ho, ho))
 10. (ho, (ho, ho))
 11. (ho, $_, (ho, ho))
 12. (ho, ho, (ho, ho))
 13. (ho, ho, ho)
 14. nothing, the two patterns don't match
 15. nothing, pattern 1 is not a legal pattern
 16. I don't have a clue...
? [1-16] 2
Incorrect: Pattern variable '$a' is bound to a value.
```

Different answers give different feedback

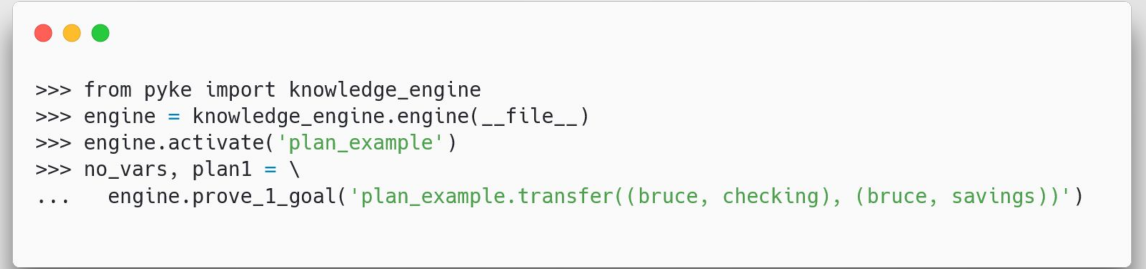
Plans

One of PyKE’s interesting features is the ability to perform **automatic program generation**, which PyKE does using backward-chaining rules that have Python code attached to them.

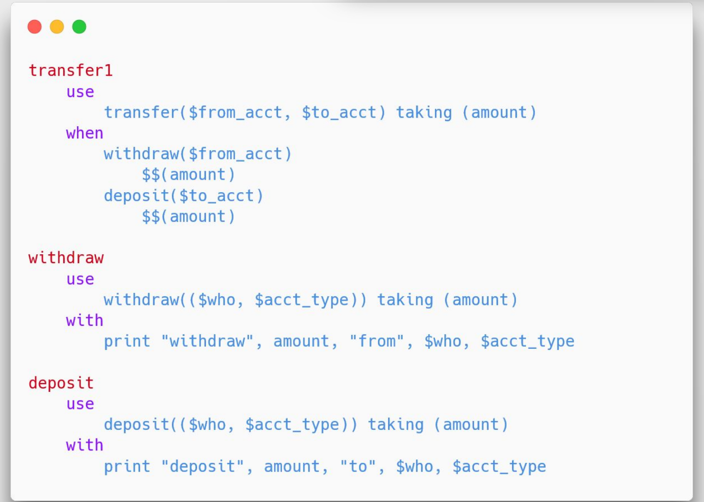
When PyKE proves a goal, it will generate a custom function, also called a **plan**, which can be saved and reused with different variables.

This allows to run different plans depending on the use-case without needing to change the underlying Python code.

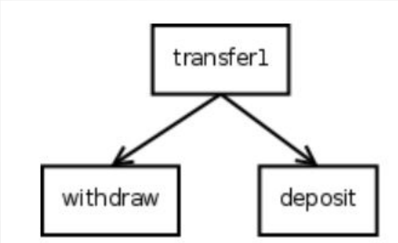
Plans

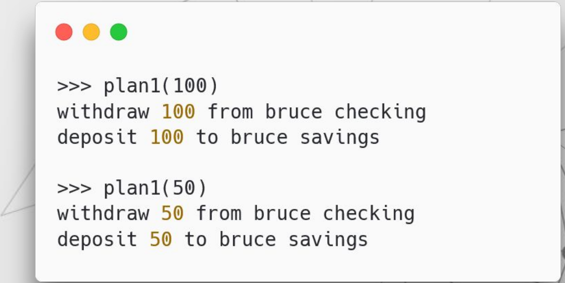


```
>>> from pyke import knowledge_engine
>>> engine = knowledge_engine.engine(__file__)
>>> engine.activate('plan_example')
>>> no_vars, plan1 = \
...   engine.prove_1_goal('plan_example.transfer((bruce, checking), (bruce, savings))')
```



```
transfer1
    use
        transfer($from_acct, $to_acct) taking (amount)
    when
        withdraw($from_acct)
            $$(amount)
        deposit($to_acct)
            $$(amount)

withdraw
    use
        withdraw(($who, $acct_type)) taking (amount)
    with
        print "withdraw", amount, "from", $who, $acct_type

deposit
    use
        deposit(($who, $acct_type)) taking (amount)
    with
        print "deposit", amount, "to", $who, $acct_type
```





```
>>> plan1(100)
withdraw 100 from bruce checking
deposit 100 to bruce savings

>>> plan1(50)
withdraw 50 from bruce checking
deposit 50 to bruce savings
```

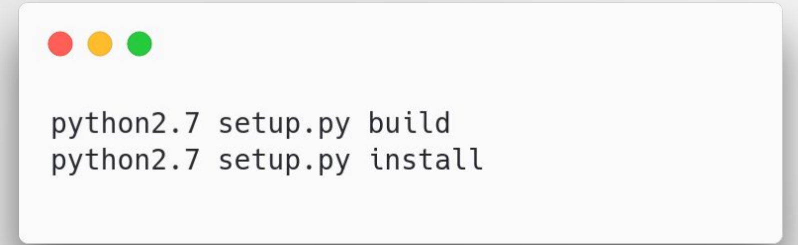
03

Tutorial

How do we create a knowledge base ?
How can we add new rules and query them ?
How can we do forward or backward chaining ?

Tutorial

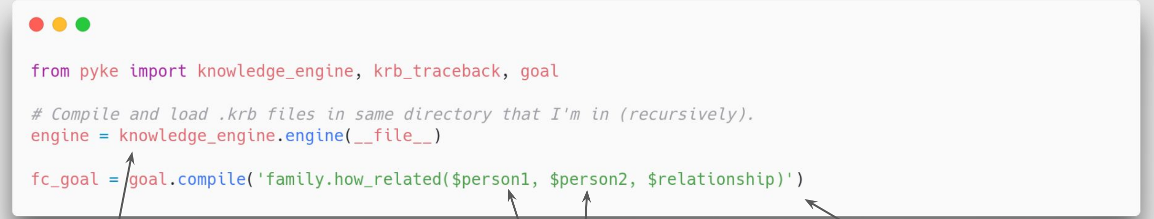
Installation



```
python2.7 setup.py build
python2.7 setup.py install
```

Tutorial

Create a Knowledge Base and infer a Goal

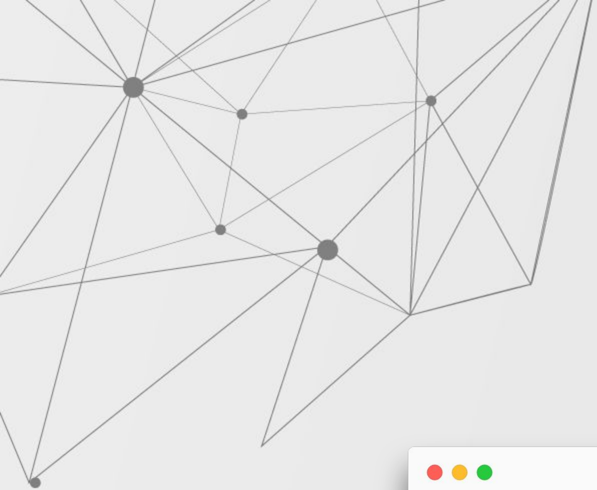


```
from pyke import knowledge_engine, krb_traceback, goal

# Compile and load .krb files in same directory that I'm in (recursively).
engine = knowledge_engine.engine(__file__)

fc_goal = goal.compile('family.how_related($person1, $person2, $relationship)')
```

Create an engine from a file

Specify the goal of our test function

We want to identify the relationships
between *person1* and *person2*

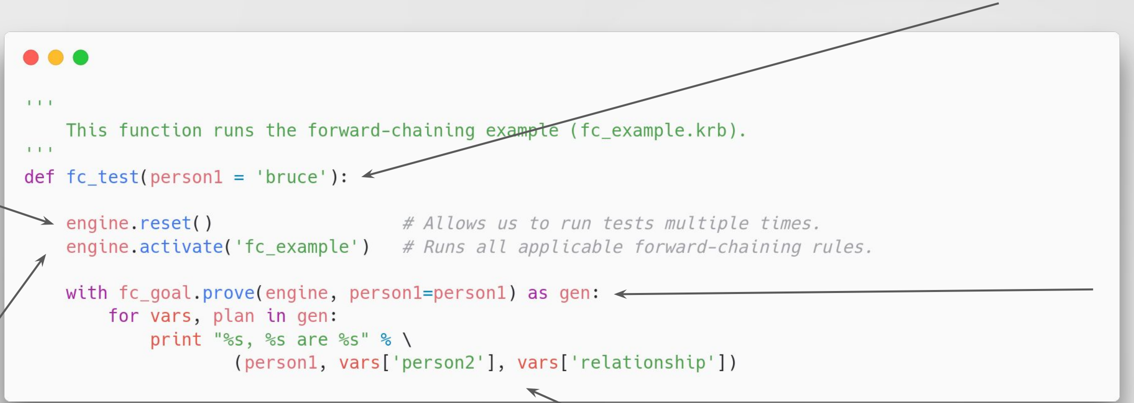


Tutorial

Forward Chaining

Person we want to find the relationships with

Resets the engine as we create new facts each time we query



```
'''
    This function runs the forward-chaining example (fc_example.krb).
'''
def fc_test(person1 = 'bruce'):

    engine.reset()                  # Allows us to run tests multiple times.
    engine.activate('fc_example')   # Runs all applicable forward-chaining rules.

    with fc_goal.prove(engine, person1=person1) as gen:
        for vars, plan in gen:
            print "%s, %s are %s" % \
                    (person1, vars['person2'], vars['relationship'])
```

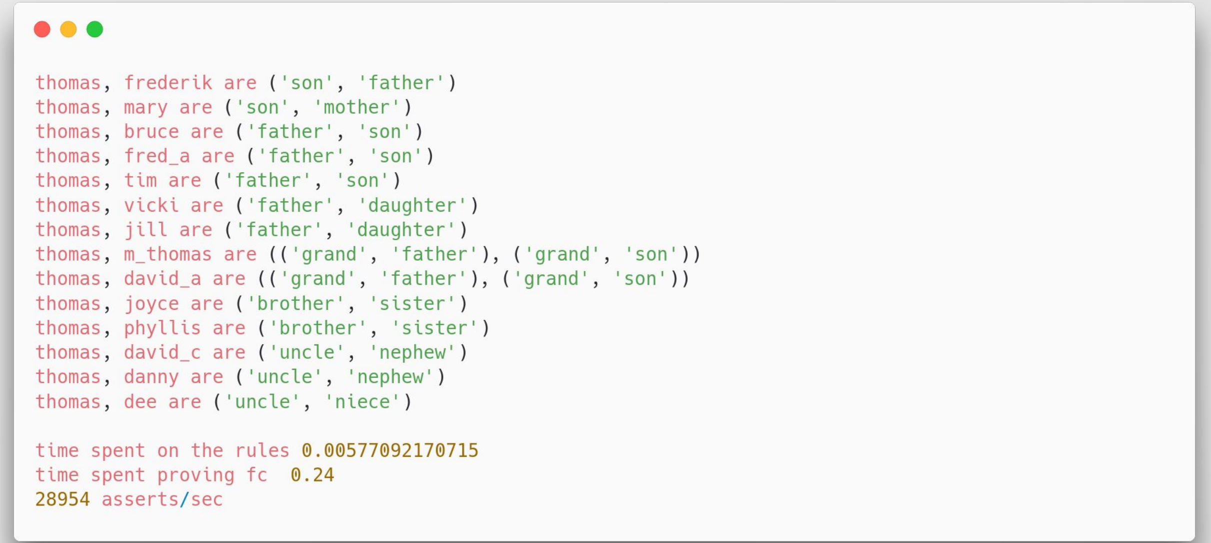
We prove our goal with the person selected

Gets the rules in the file expressed

Print every query result

Tutorial

Forward Chaining



```
thomas, frederik are ('son', 'father')
thomas, mary are ('son', 'mother')
thomas, bruce are ('father', 'son')
thomas, fred_a are ('father', 'son')
thomas, tim are ('father', 'son')
thomas, vicki are ('father', 'daughter')
thomas, jill are ('father', 'daughter')
thomas, m_thomas are (('grand', 'father'), ('grand', 'son'))
thomas, david_a are (('grand', 'father'), ('grand', 'son'))
thomas, joyce are ('brother', 'sister')
thomas, phyllis are ('brother', 'sister')
thomas, david_c are ('uncle', 'nephew')
thomas, danny are ('uncle', 'nephew')
thomas, dee are ('uncle', 'niece')

time spent on the rules 0.00577092170715
time spent proving fc  0.24
28954 asserts/sec
```

Tutorial

Backward Chaining

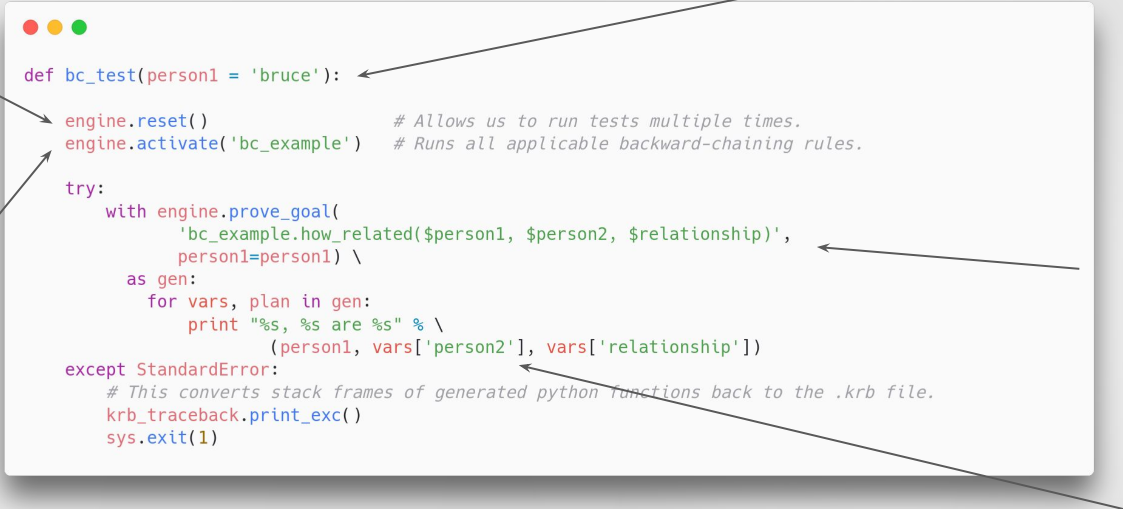
Resets the engine as we create new facts each time we query

Gets the rules in the file expressed

Person we want to find the relationships with

We prove our goal with the person selected

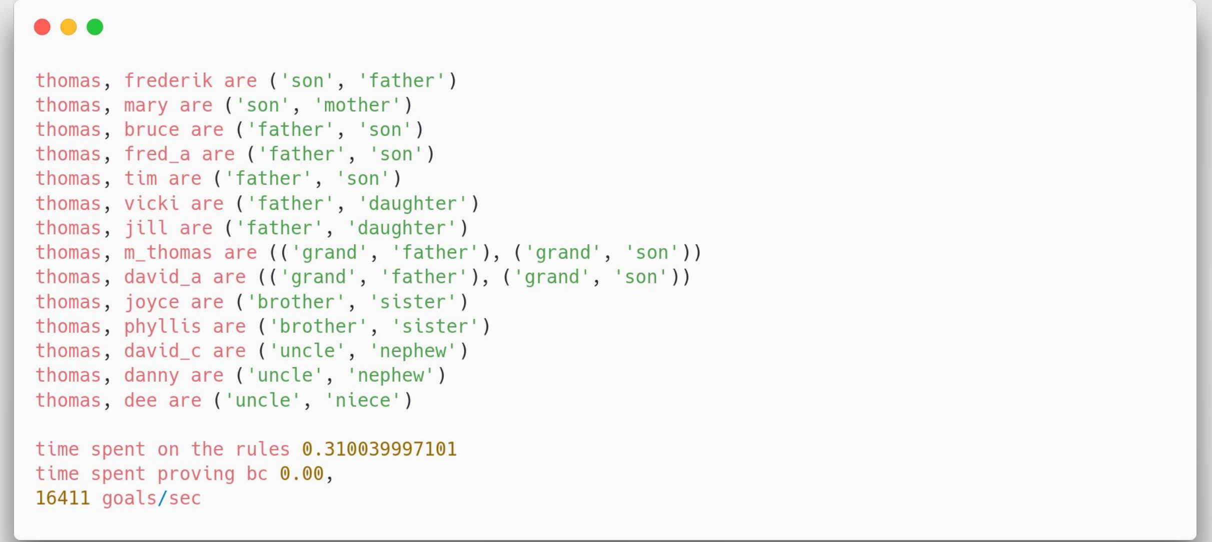
Print every query result



```
def bc_test(person1 = 'bruce'):

    engine.reset()                  # Allows us to run tests multiple times.
    engine.activate('bc_example')   # Runs all applicable backward-chaining rules.

    try:
        with engine.prove_goal(
               'bc_example.how_related($person1, $person2, $relationship)',
               person1=person1) \
          as gen:
            for vars, plan in gen:
                print "%s, %s are %s" % \
                        (person1, vars['person2'], vars['relationship'])
    except StandardError:
        # This converts stack frames of generated python functions back to the .krb file.
        krb_traceback.print_exc()
        sys.exit(1)
```

Tutorial

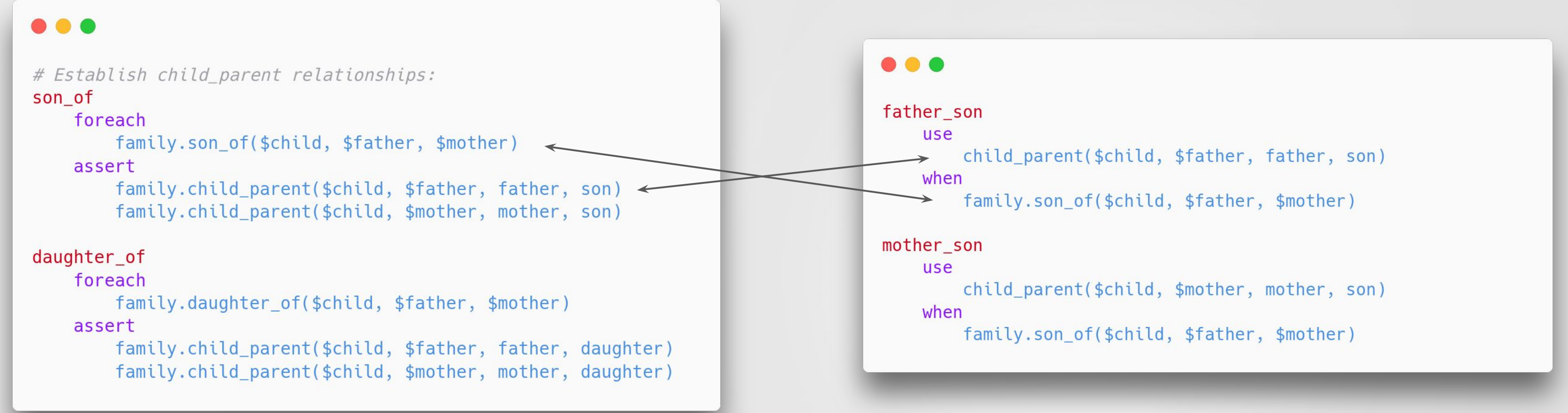
Backward Chaining



```
thomas, frederik are ('son', 'father')
thomas, mary are ('son', 'mother')
thomas, bruce are ('father', 'son')
thomas, fred_a are ('father', 'son')
thomas, tim are ('father', 'son')
thomas, vicki are ('father', 'daughter')
thomas, jill are ('father', 'daughter')
thomas, m_thomas are (('grand', 'father'), ('grand', 'son'))
thomas, david_a are (('grand', 'father'), ('grand', 'son'))
thomas, joyce are ('brother', 'sister')
thomas, phyllis are ('brother', 'sister')
thomas, david_c are ('uncle', 'nephew')
thomas, danny are ('uncle', 'nephew')
thomas, dee are ('uncle', 'niece')

time spent on the rules 0.310039997101
time spent proving bc 0.00,
16411 goals/sec
```

Tutorial

Forward Chaining VS Backward Chaining



```
# Establish child_parent relationships:
son_of
    foreach
        family.son_of($child, $father, $mother)
    assert
        family.child_parent($child, $father, father, son)
        family.child_parent($child, $mother, mother, son)

daughter_of
    foreach
        family.daughter_of($child, $father, $mother)
    assert
        family.child_parent($child, $father, father, daughter)
        family.child_parent($child, $mother, mother, daughter)
```

```
father_son
    use
        child_parent($child, $father, father, son)
    when
        family.son_of($child, $father, $mother)

mother_son
    use
        child_parent($child, $mother, mother, son)
    when
        family.son_of($child, $father, $mother)
```

foreach … assert …

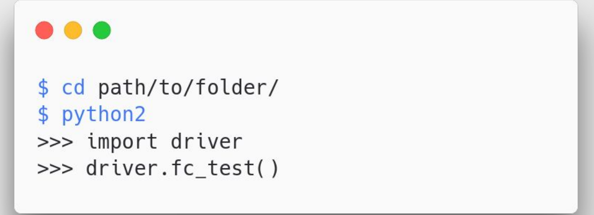
If … then …

use … when …

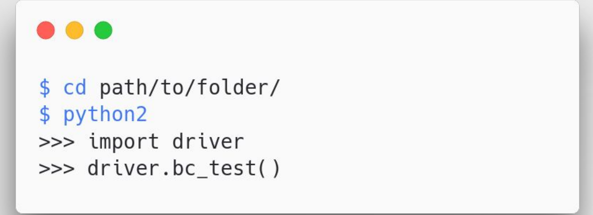
then … if ….

Tutorial

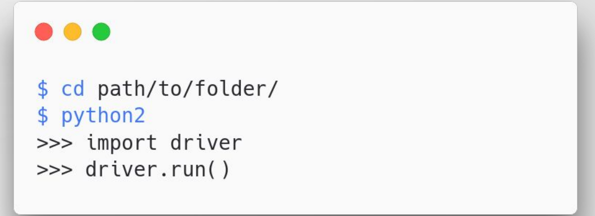
How to Run



```
$ cd path/to/folder/
$ python2
>>> import driver
>>> driver.fc_test()
```



```
$ cd path/to/folder/
$ python2
>>> import driver
>>> driver.bc_test()
```



```
$ cd path/to/folder/
$ python2
>>> import driver
>>> driver.run()
```

04

Why PyKE

Why should we use PyKE ?
What does it give us that Prolog does not ?

Why PyKE ?

It can be used to build **intelligent systems** that can:

- **Automate decision-making processes**, such as in healthcare and finance
- **Provide advice and guidance** in specific domains, such as medicine and law
- **Decide and support a user** to make informed decisions by providing them with relevant information
- Communicate with users using **natural language**, using a set of provided tools for working with natural language processing (**NLP**)
- Train data using **Machine Learning** models and learn from it, which can be used in many fields, such as healthcare, finance and marketing

PyKE vs Prolog

Similarities:

- They are both logic programming languages
- They are both based on **facts** and **rules**
- They both support back-tracking

Differences:

- **Pyke** is **Python** based while **Prolog** is a complete language
- **Pyke** is more suited to build expert systems and knowledge-based applications, while **Prolog** is more suited to natural language processing and symbolic reasoning

PyKE GitHub

THANKS

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**.

Please keep this slide for attribution.