Sistemas Expertos en Python con PyKnow

Roberto Abdelkader Martínez Pérez

robertomartinezp@gmail.com
@nilp0inter

Introducción:
¿Qué es un sistema experto?

Sistemas Expertos

Un sistema experto es un programa que contiene el “conocimiento” en un área específica de un experto.

Están diseñados para resolver problemas complejos, razonando la información mediante reglas en lugar de mediante la ejecución de procedimientos.

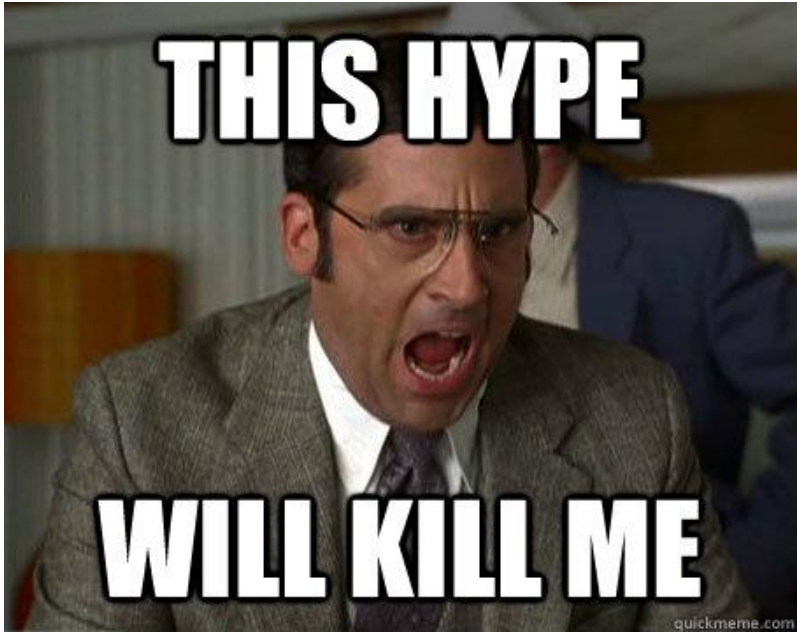
Introducción:
Breve historia de los sistemas expertos

Contexto histórico

- El objetivo inicial de los científicos era desarrollar programas capaces de *pensar* de alguna manera
- En la década de los 60 comenzaron el desarrollo de distintos GPS (General Problem Solvers); los cuales no consiguieron ser aplicados con éxito a problemas reales
- Poco a poco se llegó a la conclusión de que la *inteligencia* del sistema era proporcional al *conocimiento específico* que contenía

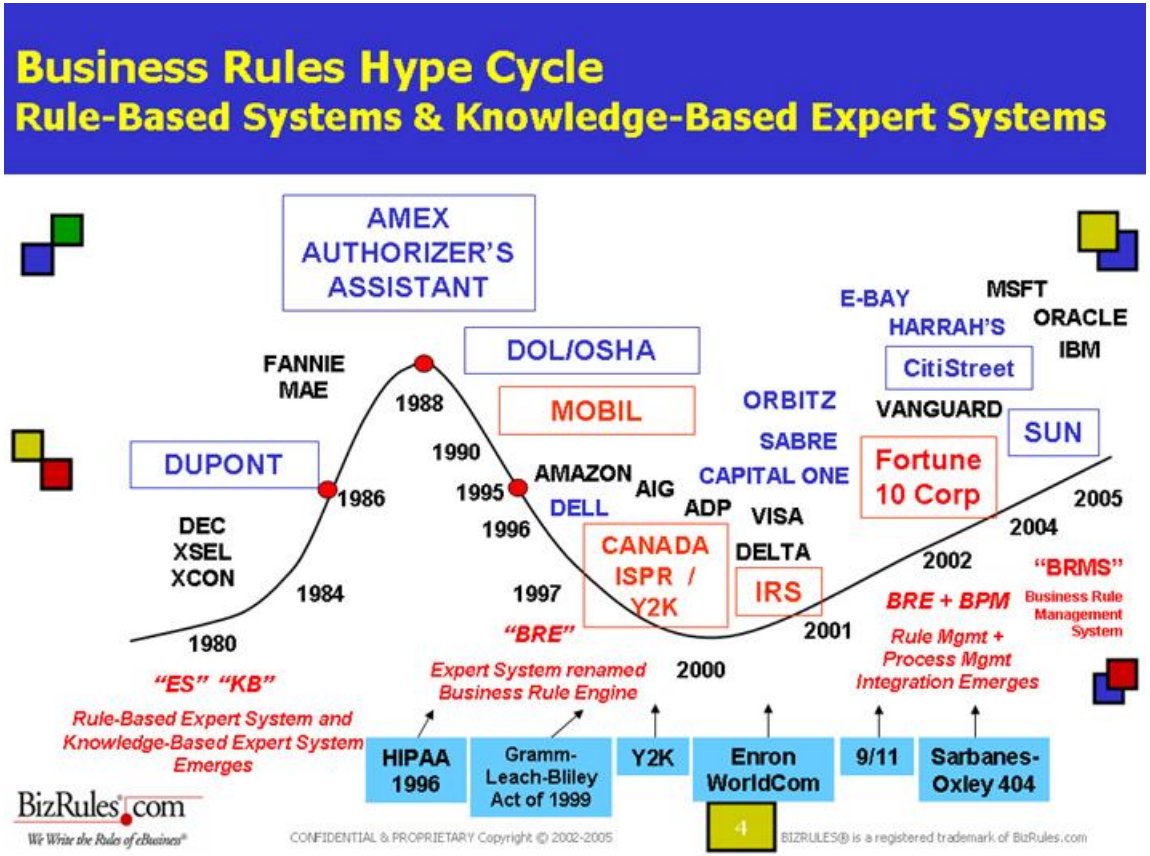
Primeros años

- A inicios de los 70 se crearon los primeros sistemas expertos y empezaron a hacerse muy populares en la década de los 80
- Algunos proyectos fueron muy exitosos (DENDRAL, MYCIN, Mistral)
- Todo el mundo se quería subir al carro de IA



Desilusión

- A principios de los 90 muchos SE se mostraron muy difíciles de mantener
- Los SE seguían siendo útiles pero sólo en determinados escenarios
- La tecnología era menos potente de lo que se pensó inicialmente



Introducción:
El panorama en Python

Desarrollo de SE en Python

PyCLIPS

- Interfaz para utilizar **CLIPS** desde Python
- Último commit en 2008
- Paquetes para Python 2.4/2.5

PYKE

- Programación lógica. Inspirado por **PROLOG**
- Último commit en 2010

¡Nace PyKnow!

Debido, entre otras cosas, al desolador panorama...

CLIPS y PyKnow

CLIPS

- Es un **lenguaje de programación** para sistemas expertos
- Su desarrollo comenzó en 1985 y continúa a día de hoy (**32 tacos**)
- Proyecto muy maduro. Implementación de referencia para otros motores de inferencia
- Soporta múltiples modelos de inferencia, orientación a objetos, programación procedimental, etc

PyKnow

- **Librería/Biblioteca** para la creación de sistemas expertos en **Python**
- Inspirada en **CLIPS** (mismos conceptos, componentes similares, etc)
- Toda la potencia de **Python**... ¡TODO ES UN OBJETO, YAY!
- Se inició su desarrollo en 2015
- Capacidades de inferencia mucho más limitadas que **CLIPS**

Curso acelerado de PyKnow

Arquitectura de PyKnow

A. **Memoria de Trabajo**
   a. Conjunto de información que maneja el sistema en cada sesión
   b. Definida en base a **hechos**
   c. Volátil
B. **Base de reglas**
   a. Conocimiento que posee el sistema
   b. Definida, programáticamente, en base a **reglas**
   c. Permanente
C. **Intérprete**
   a. Provisto por la librería
   b. Personalizable

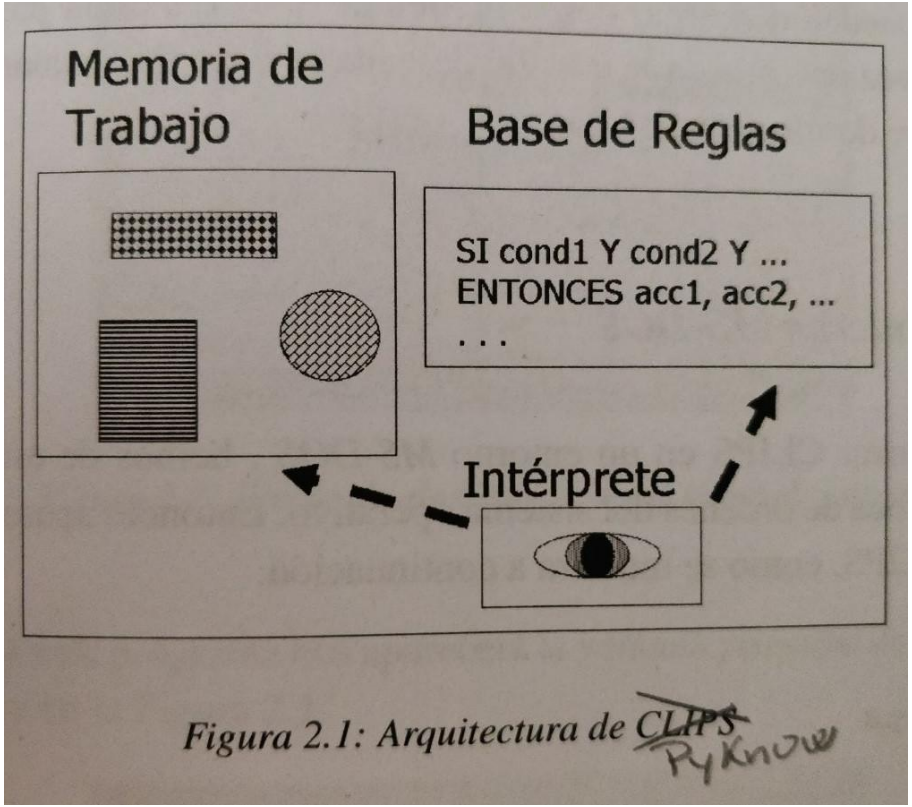


Figura 2.1: Arquitectura de ~~CLIPS~~ PyKnow

El motor de conocimientos

- Provisto por la clase **KnowledgeEngine**
- Proporciona el interfaz para interactuar con el sistema experto
- Contiene el resto de elementos del sistema
  - Reglas
  - Memoria de trabajo
  - Agenda
  - ...

```
from pyknow import *

class MiSistemaExperto(KnowledgeEngine):
    pass

mse = MiSistemaExperto()
mse.run()  # ← Ejecuta el sistema experto
```

Los hechos

- Unidad de **información** que maneja el sistema
- Se representan mediante instancias de la clase **Fact**
- La clase **Fact** es una subclase de **dict**, por lo tanto se comporta como un diccionario
- A diferencia de un diccionario, es válido omitir las claves al instanciar un **Fact**, el sistema les proporcionará una clave autonumérica
- Es recomendable crear subclases de **Fact** para representar distintos tipos de información o para enriquecer la funcionalidad añadiendo métodos

```
>>> f = Fact('x', 'y', 'z', a=1, b=2)
>>> f[1]
'y'
>>> f['b']
2
```

```
class Alert(Fact):
    """The alert level."""
    def __str__(self):
        return "ALERT(%s)" % self[0].upper()

class Status(Fact):
    """The system status."""
    pass

f1 = Alert('red')
f2 = Status('critical')
```

Las reglas (estructura)

- Unidad de **conocimiento** que maneja el sistema
- Representa unas acciones que se realizarán si se dan las condiciones necesarias
- Está compuesta por dos partes:
  - Antecedente (LHS): Condición lógica que se tiene que cumplir
  - Consecuente (RHS): Acciones a realizar si se cumple el *antecedente*
- En PyKnow una regla es cualquier método de **KnowledgeEngine** decorado con **@Rule**

```
class Alert(Fact):
    """The alert level."""

class AlertDetector(KnowledgeEngine):
    @Rule(Alert())  # Antecedente
    def match_on_every_alert():
        """
        Esta regla será lanzada cuando ocurra
        cualquier tipo de Alerta
        """
        # Consecuente
        print("Aauuua, Aauuaa")
```

Las reglas (comparación de patrones)

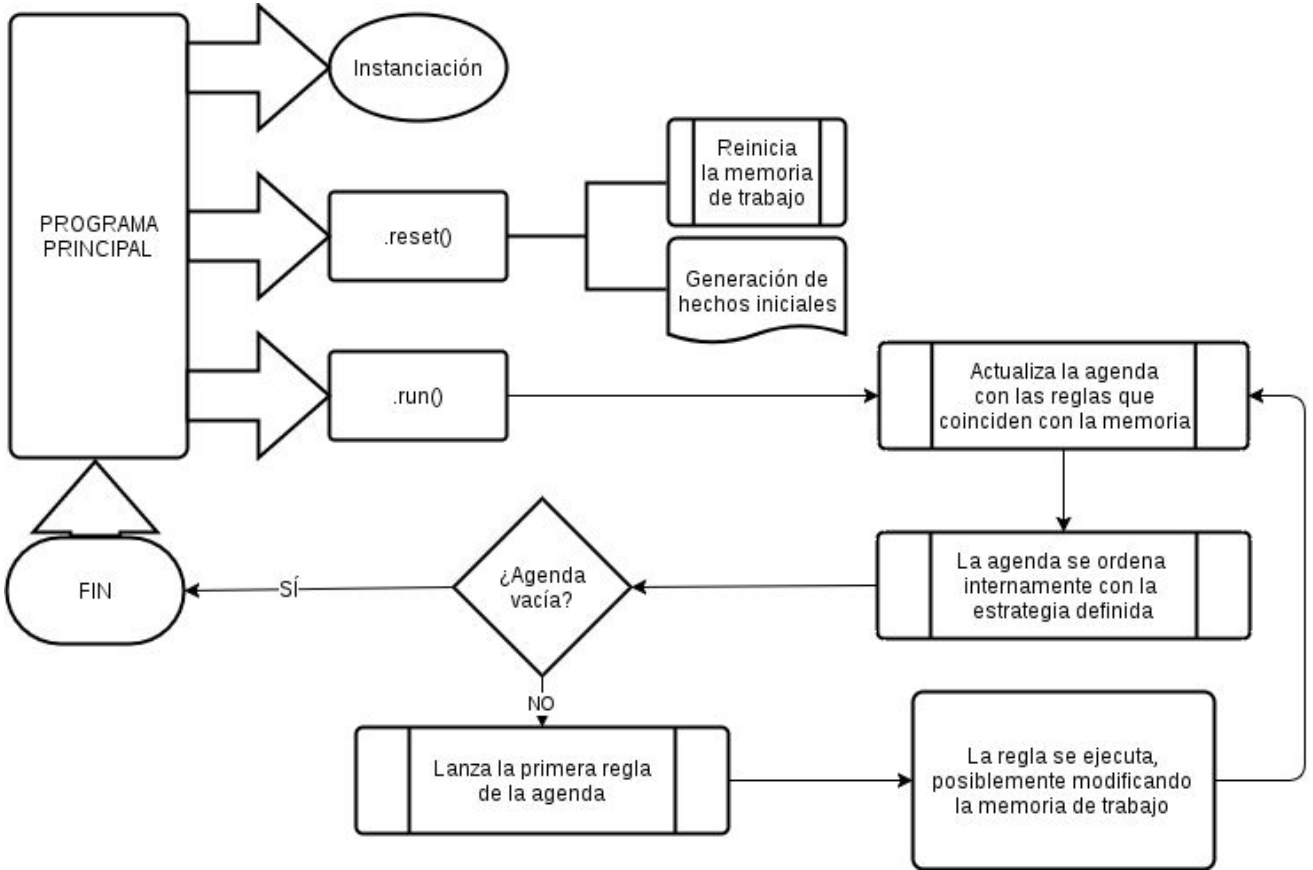
- En el antecedente expresamos las condiciones necesarias para nuestra regla utilizando patrones
- **PyKnow** provee varios operadores para expresar restricciones dentro del antecendente
  - Conditional Elements: Permite definir combinaciones de hechos (AND, OR, NOT, etc)
  - Field Constraints: Restringe los valores posibles **de un campo**, se pueden combinar con operadores (**&**, **|**, **~**)
    - **W**: Acepta cualquier valor
    - **P**: El valor debe pasar una función de filtrado
    - **L**: El valor debe ser exacto al dado

```
@Rule(Camisa(color=MATCH.c & (L('azul') |
L('gris'))),
      Pantalon(color=MATCH.c))
def _(c):
    print("Tenemos camisa y pantalon de color", c)

@Rule(Camiseta(precio=P(lambda p: 10 < p < 25)))
def _():
    print("La camiseta tiene un precio adecuado")

@Rule(Camiseta(precio=MATCH.p & P(lambda p: p < 30)),
      Presupuesto(MATCH.t),
      TEST(lambda p, t: p <= t))
def _(p, t):
    print("Entra en nuestro presupuesto y no es muy
cara")
```

El ciclo de ejecución



Ejemplo práctico:
Un generador de cupones descuento

Objetivos del sistema

1. Incentivar nuevas compras del cliente en el establecimiento
2. Fomentar el consumo de otros productos
3. Fomentar el consumo de productos con más margen de beneficio

Interfaz

Entrada

- Lista de artículos que ha comprado el consumidor

Salida

- Lista de cupones descuento que imprimir junto al recibo de compra

Implementación

DEMO

TIME!

Casos de uso

Usos apropiados y buenos hábitos

Usos apropiados

- Problemas **concretos y de corto alcance**
- Algoritmos desconocidos o sujetos a cambios constantes
- Soluciones donde importe poco o nada el orden de ejecución

Buenos hábitos

- Dividir las reglas por categorías en varias clases y componer el motor en base a **mixins**
- Dejar al motor hacer su trabajo, no forzar el orden de ejecución

Usos inapropiados y malos hábitos

Usos inapropiados

- Programas donde los algoritmos estén **claramente definidos**
- Intentar modelar el conocimiento de varios expertos dentro del mismo motor
- Permitir a NO programadores escribir las reglas **directamente**

Malos hábitos

- Programar **todo** el sistema en base a reglas
- Declarar hechos con valores mutables

¡Gracias!

github.com/nilp0inter
github.com/buguroo/pyknow