使用sympy实现命题逻辑运算

hawksoft

在sympy包中，有一个子包logic，用于命题逻辑表达式的各种运算和化简。

其使用的主要步骤如下：

- 初始化
- 定义逻辑表达式
- 进行逻辑表达式运算
- 调用函数进行各种操作

1 初始化

将下面的代码放在程序的头部。

第二句的作用，是为了优美地显示逻辑表达式。配合一些包，如malplotlib,就可以以图形化的方式显示逻辑公式。

```
import sympy as sym
sym.init_printing()
```

2 定义逻辑表达式

输入逻辑表达式，关键是要知道常用的逻辑运算符如何输入。看下面的表-逻辑操作符号对应关系表

逻辑运算符	按键	说明
$\neg$	~	这个键在键盘右上角esc键的下面
$\wedge$	&	
$\vee$	|	
$\rightarrow$	>>	这是连在一起的两个>

下面的代码分别定义了几个表达式,具体说明见下表:

变量名	行号	表达式
`expr1_1`	2	$(p \wedge q) \vee (\neg p \wedge \neg q)$
`expr1_2`	3	$(q \wedge \neg p) \vee (\neg p \wedge \neg q)$
`expr1`	4	$((p \wedge q) \vee (\neg p \wedge \neg q)) \wedge ((q \wedge \neg p) \vee (\neg p \wedge \neg q))$
`expr2_1`	9	$(p \rightarrow q) \wedge (\neg p \rightarrow \neg q)$
`expr2_2`	10	$(q \rightarrow \neg p) \wedge (\neg q \rightarrow \neg p)$
`expr2`	11	$(p \rightarrow q) \wedge (q \rightarrow \neg p) \wedge (\neg p \rightarrow \neg q) \wedge (\neg q \rightarrow \neg p)$

```
p,q,r,s,t = sym.symbols('p,q,r,s,t')
expr1_1 = (p & q ) |(~p & ~q)
expr1_2 = (q & ~p) |(~q & ~p)
expr1 = expr1_1 & expr1_2
print(expr1_1)
print(expr1_2)
print(expr1)

expr2_1 = (p >> q ) &(~p >> ~q)
expr2_2 = (q >> ~p) &(~q >> ~p)
expr2 = expr2_1 & expr2_2
print(expr2_1)
print(expr2_2)
print(expr2)

```

```
(p & q) | (~p & ~q)
(q & ~p) | (~p & ~q)
((p & q) | (~p & ~q)) & ((q & ~p) | (~p & ~q))
(Implies(p, q)) & (Implies(~p, ~q))
(Implies(q, ~p)) & (Implies(~q, ~p))
(Implies(p, q)) & (Implies(q, ~p)) & (Implies(~p, ~q)) & (Implies(~q, ~p))
```

3 各种运算

3.1 计算真值

计算逻辑表达式的值，使用函数`subs()`。这是单词substitute的前四个字母，本意是替换，即用给定的值替换逻辑变量，计算真值。该函数的参数是一个dict类型的对象。下面的代码打印出逻辑表达式expr1的真值表。

```
expr1_1 = (p & q ) |(~p & ~q)
expr1_2 = (q & ~p) |(~q & ~p)
expr1 = expr1_1 & expr1_2

# 打印真值表
print('{:^6s} {:^6s} {:^10s}'.format("p","q","result"))  #print table head
print('{:-^6s} {:-^6s} {:-^10s}'.format("-","-","-"))    # print horizon line
for p1 in [False,True]:
    for q1 in [False,True]:
        r = expr1.subs({p:p1,q:q1})
        print('{:^6s} {:^6s} {:^10s}'.format(str(p1),str(q1),str(r)))
```

```
  p      q      result
------ ------ ----------
False  False     True
False   True    False
 True  False    False
 True   True    False
```

3.2 由逻辑表生成逻辑表达式

有时候，我们可以画出操作的逻辑表，需要获得该逻辑表对应的逻辑表达式。这种情况在逻辑电路设计中最为常见，比如下面的例子：

> 例 1： 二进制一位全加器：

> 二进制一位加法的逻辑如下表：

被加数	加数	下一位的进位	结果	向上进位
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

```
miniterm = [[0,0,1],[0,1,0],[1,0,0]]
p,q,r,s,t = sym.symbols('p,q,r,s,t')
result = sym.SOPform([p,q,r], miniterm)
print('result=',result)
miniterm = [[0,1,1],[1,0,1],[1,1,0],[1,1,1]]
carrier= sym.SOPform([p,q,r], miniterm)
print('carrier=',carrier)
```

```
result= (p & ~q & ~r) | (q & ~p & ~r) | (r & ~p & ~q)
carrier= (p & q) | (p & r) | (q & r)
```

这里的SOP是sum of production，可以理解为析取范式（Disjunctive Normal Form）。

根据上面的表达式，就可以画出结果和进位的逻辑电路图。

注意：在逻辑电路中，只有三个电子元件，分别是与门元件，或门元件，以及非门元件，分别实现逻辑与，逻辑或，以及逻辑非运算。

> 例 2： 三位评委的表决器设计：

> 根据三位评委对选手的表决，以少数服从多数原则，决定选手能否晋级。其逻辑如下表:

评委1	评委2	评委3	Pass
No	No	No	No
No	No	Yes	No
No	Yes	No	No
No	Yes	Yes	Yes
Yes	No	No	No
Yes	No	Yes	Yes
Yes	Yes	No	Yes
Yes	Yes	Yes	Yes

```
miniterm = [[0,1,1],[1,0,1],[1,1,0],[1,1,1]]
p,q,r,s,t = sym.symbols('p,q,r,s,t')
passExp= sym.SOPform([p,q,r], miniterm)
print('pass=',passExp)
```

```
pass= (p & q) | (p & r) | (q & r)
```

根据上面的表达式，就可以画出Pass的逻辑电路图。

3.3 求范式（Normal Form）

逻辑范式分为析取范式（Disjunctive Normal Form DNF）和合取范式（Conjunctive Normal Form CNF），采用不同的函数。

```
p,q,r,s,t = sym.symbols('p,q,r,s,t')
exp1 = (p>>q) &(p & q)
expr101 = sym.to_cnf(exp1,simplify=False)
expr102 = sym.to_cnf(exp1,simplify=True)
expr201 = sym.to_dnf(exp1,simplify=False)
expr202 = sym.to_dnf(exp1,simplify = True)
print(expr101)
print(expr102)
print(expr201)
print(expr202)
```

```
p & q & (q | ~p)
p & q
(p & q) | (p & q & ~p)
p & q
```

> 注意： 函数的第二个参数，决定是否对该范式进行简化。

另外，还有一个函数`simplify_logic()`,也可以获得逻辑表达式的范式，但默认会返回cnf和dnf中最简单的那个形式，这一点由这个函数的名字（ simplify_logic）可以看出，即求出最简单的等价逻辑表达式。

```
p,q,r,s,t = sym.symbols('p,q,r,s,t')
exp1 = (p>>q) &(p & q)
expr301 = sym.simplify_logic(exp1)
print(expr301)
```

```
p & q
```

3.4 可满足性

可满足性，是指给定一个逻辑表达式，求出该表达式的所有成真赋值。实际中的很多问题，都可以建模为逻辑表达式的可满足性问题。

> 例3: 一个岛屿上有两类人:骑士和无赖。骑士永远说真话，而无赖永远说假话。你去岛上碰见A和B,
> A说" B是骑士。”
> B说:“我们两个不是同类人。”
> 问A和B的类型是什么?

引入变量，A : A是骑士； B：B是骑士。根据题意，针对两个人的说法分别有：

$$(A \wedge B) \vee (\neg A \wedge \neg B) = T \tag{1}$$
$$(B \wedge \neg A) \vee (\neg B \wedge \neg A) = T \tag{2}$$

问题变成了：找到A和B的一个赋值，使上面的公式（1）和（2）都为真。也就是说，找到A和B的一个赋值，使公式(1)和公式（2）的合取为真，这就是命题公式的可满足性问题，可以用函数`satisfiable(()`获得结果，如下面的代码：

```
p,q,r,s,t = sym.symbols('p,q,r,s,t')
expr1_1 = (p & q ) |(~p & ~q)
expr1_2 = (q & ~p) |(~q & ~p)
expr1 = expr1_1 & expr1_2
models = sym.satisfiable(expr1,all_models=True)
for i in models:
    print(i)
```

```
{p: False, q: False}
```

函数的返回值是一个列表，列表的每一个元素是一个字典，对应一个该逻辑表达式的一个成真赋值。这个题目的的结果只有一个成真赋值，这就是该问题的解,即A和B都是无赖。

> 例4: 某科研所有三名青年高级工程师A，B，C。所里要选派他们中的1到2人出国进修，由于所里工作的需要，选派时必须满足以下条件：
> ①若A去，则C也去
> ②若B去，则C不能去
> ③若C不去，则A或B去
> 问所里应如何选派他们?
> 根据题意，首先引入变量：A代表派A去，$\neg A$代表不派A去。同理定义B和C，则可以列出如下的限制条件：

$$A \to C \tag{1}$$
$$B \to \neg C \tag{2}$$
$$\neg C \to (A \vee B) \tag{3}$$

这是一个可满足性问题，采用下面的代码：

```
A,B,C= sym.symbols('A,B,C')
expr1_1 = (A >> B)
expr1_2 = (B >> ~C)
expr1_3 = ~C >>(A | B)
expr1 = expr1_1 & expr1_2 & expr1_3
models = sym.satisfiable(expr1,all_models=True)
for i in models:
    print(i)
```

```
{B: True, A: False, C: False}
{B: True, A: True, C: False}
{C: True, A: False, B: False}
```

可以看出，该问题有三个可满足的解，根据题意，只要是1到2人，都满足要求，因此这三个解都是这个问题的解，即：

1. 方案1: 只派B去
2. 方案2: 派A和B去
3. 方案3: 只派C去

3.5 等价(Logical Equavalent)性判定

对两个逻辑表达式，判定能否找到一个符号映射，使两者等价。
如下面的例子：

```
p,q,r,a,b,c= sym.symbols('P,Q,R,A,B,C')
exp1 = p>>q
exp2 = ~a | b
result = sym.bool_map(exp1,exp2)
print(result)
```

```
(Q | ~P, {P: A, Q: B})
```

这个函数的返回值有些奇怪，分为两部分：第一部分是说，第一个表达式可以等价地化简为这个形式；第二部分是说，当将两个表达式中的符号以这种方式对应起来时，这两个逻辑表达式等价。

在实际使用中，如果两个表达式采用了相同的逻辑符号，则第二部分没有意义，如下面的代码：

```
p,q,r,a,b,c= sym.symbols('P,Q,R,A,B,C')
exp1 = p>>q
exp2 = ~p | q
result = sym.bool_map(exp1,exp2)
print(result)
```

```
(Q | ~P, {P: P, Q: Q})
```

这里以上面的例3为例子。

例3中的第一个条件即可表示为上面的（1）式，即：

$$(A \wedge B) \vee (\neg A \wedge \neg B) \tag{4}$$

也可以表示为下面的逻辑表达式：

$$(A \rightarrow B) \wedge (\neg A \rightarrow \neg B) \tag{5}$$

那么这两个表达式应该等价，是不是呢，我们可以通过下面的代码检验：

```
a,b,c= sym.symbols('A,B,C')
exp1 = (a & b) | (~a & ~b)
exp2 = (b >> a) & (~b >> ~a)
result = sym.bool_map(exp1,exp2)
print(result)
```

```
((A & B) | (~A & ~B), {A: A, B: B})
```

上面的代码说明，这两个表达式是等价的，而且都可以等价为：$(A \wedge B) \vee (\neg A \wedge \neg B)$

4 结尾

逻辑命题表达式的操作比较简单，sympy为此提供了全面的支持。

在采用命题逻辑解决实际问题时，建模完成后，就可以根据问题的要求，调用不同的函数，完成计算。