KIT103/KMA155 Practical 04: Logic

This week we'll apply some of the logic techniques you've learnt to generate truth tables and to simplify Boolean expressions. Most of the variables will be p's, q's, a's and b's, but they could just as easily be is_Australian, is_studying_maths or system_is_down; the principles are the same.

Preparation

In the Labs section of MyLO download the file **truth_table.py**. It contains a function (`truth_table`) that accepts another function representing a predicate and an integer indicating the number of (Boolean) parameters that predicate has. Load truth_table.py in Spyder and run it. Use `help(truth_table)` to see its own documentation or `help('truth_table')` to see the documentation for the whole module. There is also an over-commented version at the bottom of the script if you want to understand how the different parts work later.

1 Build-your-own truth tables

Task: As a warm up exercise, define the following functions representing predicates. Once you've defined them, use `truth_table` to check that they work as expected, setting the second argument to the number of parameters that each predicate has.

```
def negate(p):
    return not p

def my_and(p, q):
    return p and q

def my_or(p, q):
    return p or q
```

You will not need to use these functions again; they are just a way of passing the *logic* of *not*, *and* and *or* to the truth table generator.

1.1 (optional) $P \Rightarrow Q$ does not imply `if p: return q`

The truth_table.py script also contains two versions of functions that try to implement the logical operator $\Rightarrow$ (implies). The function called `implies` is a valid implementation, while `naive_implies` is not.

Task: Can you identify what is wrong with `naive_implies` and fix it? (Remember to use `truth_table` to check what result they give for different inputs.)

2 Testing and simplifying predicates

In this part you'll define (and simplify) a number of predicates, and use the `truth_table` function to verify that your simplified versions are correct. First, we need to go over the two ways you can define a predicate as a function: by defining a named function using `def`, or by using a lambda expression to define a short, unnamed function. The two examples in the table below both represent the predicate $p \land \neg q$. (If you want to save time then go with the first option and skip ahead to the task.)

Named function	Lambda expression

`def p1(p, q):`
`    return p and not q`
	`lambda p, q: p and not q`

Note the differences in syntax: the lambda expression is simpler, with no parentheses around its parameters and no `return`; whatever expression appears after the `:` is the result of the function.

The named function, once defined, can be referred to by its name, `p1`. The lambda expression, on the other hand, is just a value (that happens to be a function), so the only way to use it is either to assign it to a variable for use later or to put it where you would otherwise put a function's name, such as the first argument to `truth_table`. For instance, this is perfectly valid in Python:

```
g = lambda p, q: p and not q
g(True, False) #returns True
```

Task: Using either approach, implement the following predicates as written and examine their corresponding truth tables (remember that you could, and most often would, write the truth table by hand). In the following the letters a, b and c represent the Boolean values on which the predicates operate, and so are the parameters to the predicate functions you will write.

1. $\neg a \land \neg b \land \neg c$
2. $a \lor (b \land a)$
3. $\neg a \lor a$
4. $(a \land b) \lor (\neg a \land b)$
5. (optional) $(a \land b) \lor (a \land c)$

2.1 Simplifying predicates

Task: Now simplify each of the predicates above. That is, write a new definition for each function that uses a shorter Boolean expression to calculate its result. (Consult the Math slides on Logical Equivalences for help.) After the change, use `truth_table` to check your modified version is correct (i.e., gives the same result as the original).

3 Implementing your own $\exists$ and $\forall$

In mathematics we can write $\exists$ and $\forall$ to express that there exists some value that satisfies a given predicate or that a given predicate is true for all values in a particular set. For many mathematical problems it is possible to prove a given statement that involves ∃ and ∀. In contrast, in most programs when these questions arise there is no way to answer them without (the program) actually looking at the data.

Task: Write two functions named `exists` and `for_all` that accept a set (or any collection really) and a predicate as arguments and which implement the logic of ∃ and ∀. The first line of their definition should be `def exists(s, p):` and `def for_all(s, p):`, respectively. Their expected behaviour should be the following (where >>> is the Python interactive prompt):

```
>>> exists({1, 2, 3, 4, 5}, lambda e: e == 4)
True
>>> exists({1, 2, 3, 5}, lambda e: e == 4)
False
>>> for_all({1, 2, 3, 4, 5}, lambda e: e < 5)
False
>>> for_all({1, 2, 3, 4}, lambda e: e < 5)
True
```

(For those who have some prior programming experience) does the structure of your code for `exists` and `for_all` remind you of other functions you have implemented in the past?

Task: (optional) If you'd like to try out your new `exists` and `for_all` functions then load the set of three-letter words from last week and try different predicates with both. For instance, are there any words in that collection that end with 'q', or begin with 'q'.

Hint: you can refer to the individual letters in a string by position, so `s[1]` is the second letter in the string variable `s`. You can also count backwards from the end a string, so `s[-1]` is the last letter in the string variable `s`.

Useful references

Lambda expressions: You will not be explicitly tested on lambda expressions, but they are a powerful and very useful mechanism in Python (and other languages).

https://docs.python.org/3/reference/expressions.html#lambda