Programming Languages

Week 5

Logic programming in Python

College of Information Science and Engineering
Ritsumeikan University
plan

this week:
• first-order logic
• modelling first-order logic programmatically in Python

next week:
• using first-order logical inference as a programming language
• writing declarative programs in Python

third week:
• writing ‘real’ declarative programs in Prolog
Boolean algebra

statements about truth values, using three basic logical operators

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>⇒</td>
<td>1</td>
</tr>
<tr>
<td>¬B</td>
<td>⇒</td>
<td>0</td>
</tr>
<tr>
<td>A ∧ ¬B</td>
<td>⇒</td>
<td>1</td>
</tr>
<tr>
<td>A ∨ ¬B</td>
<td>⇒</td>
<td>1</td>
</tr>
<tr>
<td>any</td>
<td>any</td>
<td>0</td>
</tr>
</tbody>
</table>

where $A$ and $B$ might be

- simple truth values (0/1)
- Boolean-valued expressions ($x < y$)
- real-world conditions (it is raining)
- etc.

new operators and functions defined in terms of existing ones

$$\text{exclusive-or}(A, B) = (A \land \neg B) \lor (\neg A \land \neg B)$$
propositional calculus

adds:

- **conditional**: if $P$ then $Q$ \( P \rightarrow Q \)
- **biconditional**: $P$ is equivalent to $Q$ \( P \leftrightarrow Q \)

**propositions**

<table>
<thead>
<tr>
<th>premise</th>
<th>conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1. \quad P \rightarrow Q )</td>
<td>( 2. \quad P )</td>
</tr>
</tbody>
</table>

\( (P \text{ and } Q \text{ are Boolean values or functions}) \)

‘zeroth-order’ logic
predicate calculus

adds: unbound variables and quantification

makes statements indirectly using _quantified variables_

given the set of all foods \( F \), “some food is delicious”:

- there exists a \( p \) such that
  - \( p \) is a kind of food
  - \( p \) is delicious

\[
\exists p : p \in F, \text{is-delicious}(p)
\]

“all food is edible”:

- for all \( p \) such that
  - \( p \) is a kind of food
  - \( p \) is edible

\[
\forall p : p \in F, \text{is-edible}(p)
\]

_first-order logic_
modeling logic in Python

we have seen how generators can

• generate sequences (including infinite ones) for a loop body
• guard a loop body, controlling whether or not it is run
• be combined to make ‘and’ and ‘or’ behaviour

and how value holders can

• remember whether or not they are bound to a value
• allow or prevent a new value being bound
• guard a loop body, running it only when a binding is valid
  – i.e: the binding can be made, or it already exists

this is enough to program a complete model of first-order logic

let’s review some of the components and clues that we need
review: generators

using a loop + generator as a conditional statement

```python
def foo():
    yield
for _ in foo():
    print "success"
#=> success
```

```python
def foo():
    if False:
        yield
for _ in foo():
    print "success"
#=>
```

note that the value of a `yield` defaults to `None`

- we don’t care what the value is, because
- we just want to control the execution of the loop body
review: generators

using a generator’s argument to control a conditional statement

def isEven(x):
    if x % 2 == 0:
        yield

for i in range(10):
    for _ in foo(i): print i,
#=> 0 2 4 6 8

def isTriple(x):
    if x % 3 == 0:
        yield

for i in range(10):
    for _ in isTriple(i): print i,
#=> 0 3 6 9
review: generators

placing two generators in series makes a logical ‘or’

```python
for i in range(10):
    for _ in isEven(i):
        for _ in isTriple(i):
            print i,
#=> 0 0 2 3 4 6 6 8 9 (even OR odd)
```

placing one generator inside another makes a logical ‘and’

```python
for i in range(10):
    for _ in isEven(i):
        for _ in isTriple(i):
            print i,
#=> 0 6 (even AND odd)
```
review: value holders

last week we used a list (or tuple) to make a promise

let’s be more sophisticated and make a trivial class of objects

class valueHolder:
    def __init__(self):  # this function initialises new objects
        self.value = None  # ’self’ is the name of the new object

(then, if we write \(X = \text{valueHolder}()\), we can check whether or not \(X\) is a valueHolder

    if isinstance(X, valueHolder): ...

and use \(X.value\) to store any values we want)

e.g., we can write a function to set the value, provided \(v\) is unbound

    def setValue(v, value):
        if not v.value is None:
            v.value = value

and later get the value back from \(v\)

    def getValue(v):
        return v.value
def genIfNone(v, value):
    if getValue(v) is None:
        # v is unbound
        setValue(v, value)
        # bind it to value
        yield
        # run the loop body once
        setValue(v, None)
        # unbind v again

setValue(x, None)
    # make x unbound

print getValue(x)

for _ in genIfNone(x, "outer"):
    print getValue(x)
    # unbound x -> x bound to "outer"
    for _ in genIfNone(x, "inner"):
        print getValue(x)
        # bound x cannot be bound again
        # this body does not run

print getValue(x)
    # x is unbound again

output:
None
outer
None
bound and unbound variables

but what if...

- we want to be able to really set the value to None?
- the variable is already bound to the same value?

in both cases the generator should succeed

class Var:
    def __init__(self):
        self.value = None
        self.bound = False  # bound is independent of value

    def assign(v, value):
        # genIfNone rewritten without setValue()
        if not v.bound:
            v.value = value
            v.bound = True
            yield
            v.bound = False
        elif v.value == value:
            yield
what should happen if we set one variable equal to another?

- they should both represent the same value
- assigning one should assign to the other too

```python
x = var()
y = var()
x = y
for _ in assign(x, 4): print getValue(y) #=> 4
for _ in assign(y, 2): print getValue(x) #=> 2
```

note that this should work even if `y` is unbound

- assigning variables to variables forms a chain
- any variable in the chain is equivalent to the last one in the chain
  - including whether or not it is currently bound

we can obtain this behaviour by modifying the `getValue()` function
review: value holders

def get(varOrVal):  # getValue rewritten to handle chains of variables
    if isinstance(varOrVal, var) and varOrVal.bound:
        # varOrVal is a bound variable: follow the chain to the end
        return get(varOrVal.value)
    # varOrVal must be a real value or an unbound variable
    return varOrVal

recursive use of get() follows variable chains to their end

get() can be called with a value, as well as with a variable
equivalence should be symmetrical

we can now write

\[ x, y = \text{var}(), \text{var}() \]

\[ \text{for } _\_ \text{ in assign}(x, y): \ldots \text{x means y in the loop body} \]

\[ \text{for } _\_ \text{ in assign}(x, 4): \ldots \text{x means 4 in the loop body} \]

but we cannot write

\[ \text{for } _\_ \text{ in assign}(2, x): \ldots \text{x means 2 in the loop body} \]

mathematics is not quite so rigid

- if we say \( 2 = x \) in mathematics or logic, we are stating a fact
- not performing an assignment

stating two things are the same, either way round, is called \textit{unifying} them
unification

unification states two things are the same

it is a guard, controlling a loop, which runs if

- the two things already mean the same thing
- one of the things can be made to mean the other thing
  - either way around

def unify(L, R):
    # make L and R mean the same thing
    Lval, Rval = get(L), get(R)  # follow variable chains to the end
    if isinstance(Lval, var):
        # Lval can be assigned to
        for _ in _assign(Lval, Rval): yield  # make Lval mean Rval, succeed
    elif isinstance(Rval, var):
        # Rval can be assigned to
        for _ in _assign(Rval, Lval): yield  # make Rval mean Lval, succeed
    else:
        if Lval == Rval:
            yield  # values are the same, succeed
exercises

this week’s exercises:

• develop useful `assign` and `get` functions
  – following the same ideas as presented here
• combine them into a unification operation

which is all we need, to do useful logic programming in Python

next week’s exercises:

• review how logic programs are written
• develop a simple example of a knowledge base and how to query it
• present a more complex example for you to attempt with less help
  – which may even seem like magic, when you see it work for the first time
bind — to associate a value with a variable.

binding — an association between a variable and a value.

bound variable — a variable that currently has a valid value assigned to it.

declarative — programs that describe what they want to know (and the information needed to derive it) rather than describing the actual steps needed to calculate it.

facts — a statement of truth about a property of a thing, or a relationship between two things.

first-order logic — an extension of Boolean (zero-order) logic, which adds quantifiers over logical variables (‘there exists $X$ such that ...’) and relations between statements using those quantifiers.

generator — a Python object used to provide a sequence of values to a for loop. Each time the generator yields the loop body runs with the loop variable set to the value that was yielded.

inference — making logical deductions based on a set of facts and rules. Given a fully-qualified statement, inference can tell us if it is true or false. Given a partially-qualified statement, inference can tell us all possible values that can be assigned to the free variables to make the statement true.

predicate calculus — another name for first-order logic.

quantified variable — a variable whose value satisfies a certain condition. The existential quantifier provides a condition that must be satisfied by at least one possible value of the variable. The universal quantifier provides a condition that must be satisfied by all possible values of a variable.

query — a question posed to a knowledge base, such as ‘is this statement true?’, or ‘what are all possible values of the variables in this statement that cause it to be true?’
**rule** — a function which takes a premise and a set of values, and uses facts and/or other rules to determine whether the premise is true or false.

**satisfy** — to make an assignment of values to variables that causes a premise to be true.

**unbound variable** — a variable that has no valid value assigned to it, e.g., either because it has never been initialised or because all previous bindings have expired.