1. Programming Functions in Python

PDAs use a store to remember current states, process sub-strings, then come back. Programs can do recursive processing in the same way (just more complex state).

This requires functions, useful for re-using similar operations in your programs. For example, if you have the following program for recognizing ‘... is ...’ assertions:

```python
import sys
import re

for s in sys.stdin:
    m = re.match('(.*) is (.*)', s)
    if (m is not None and
        (re.match('a .*', m.group(1)) is not None or
         re.match('an .*', m.group(1)) is not None ) and
        (re.match('a .*', m.group(2)) is not None or
         re.match('an .*', m.group(2)) is not None ))
    print('assertion')
```

which reads sentences like the following and reports whether they are assertions:

```
a cat is a mammal
```

then you may want to consolidate the ‘a’/‘an’ behavior for the parts before and after ‘is’.

This can be done by defining functions:

(a) \( \langle \text{delim-stmt} \rangle \rightarrow \text{def} \ (\alpha\to\beta\text{-var}) \ (\langle\alpha\text{-var}\rangle) : \text{NEWLINE} \ (\langle\text{suite}\rangle) \)

\( \) define a function from expressions of type \( \alpha \) to expressions of type \( \beta \)

(b) \( \langle \text{delim-stmt} \rangle \rightarrow \text{return} \ (\beta\text{-expr}) \)

\( \) return an expressions of appropriate type at the end of the function suite

(c) \( \langle \beta\text{-expr} \rangle \rightarrow \langle \alpha\to\beta\text{-expr} \rangle \ (\langle\alpha\text{-expr}\rangle) \)

\( \) apply a function to an argument of appropriate type

For example:

```python
import sys
import re

def isNounPhrase(s):
    return (re.match('a .*', s) is not None or
        re.match('an .*', s) is not None)
```
for s in sys.stdin:
    m = re.match('(.*) is (.*)',s)
    if ( m is not None and
        isNounPhrase(m.group(1)) and
        isNounPhrase(m.group(2)) ):
        print('assertion')

2. Local Variables: What Happens in Vegas

   Note: for the most part, what happens in functions, stays in functions:

```
def printFromDifferentVariable():
    s = 'in here, s is this new sentence'
    print(s)

    s = 'out here, s is this old sentence'
    printFromDifferentVariable()
    print(s)
```

will print:

```
in here, s is this new sentence
out here, s is this old sentence
```

What happened to `s` in the function didn’t change the `s` outside the function.

3. Recursive Functions

   Locality means a function can be used inside its own definition — called a recursive function:

```
import sys
import re

def isNounPhrase(s):
    m = re.match('the .* of (.*)',s)
    return ( (m is not None and isNounPhrase(m.group(1))) or
             re.match('a .*', s) is not None or
             re.match('an .*',s) is not None )

for s in sys.stdin:
    m = re.match('(.*) is (.*)',s)
    if ( m is not None and
         isNounPhrase(m.group(1)) and
         isNounPhrase(m.group(2)) ):
        print('assertion')
```

accepts:

```
a lion is the cousin of a cat
```

As it executes, this function does the same thing that a PDA does:
• remember the current state (and local variables, pushed onto a ‘program stack’)
• execute some sub-process (in this case, calling itself a on sub-list)
• return to the remembered program state (and local variables, popped off the stack)

4. **Practice**

Write a recursive function \( \text{conc}(n, s) \) that concatenates together an \( n \)-length sequence of \( s \)-s’s, observing that this is simply an \( s \) concatenated with an \( n-1 \)-length sequence of \( s \)-s’s.

5. **Objects**

Objects are types, that have their own member variables and functions (‘methods’)

Objects can be defined using a ‘class’ statement, with functions defined in the suite:

(a) \( ⟨\text{delim-stmt}⟩ \rightarrow \text{class} \langle \tau\text{-type-id} \rangle : \text{NEWLINE} \langle \text{suite} \rangle \)

(where \( \langle \tau\text{-type-id} \rangle \) is a class name, like ‘ListLengthDetector’)

Member functions defined in the suite of a class take the class itself as an initial parameter:

(b) \( ⟨\text{delim-stmt}⟩ \rightarrow \text{def} \langle \tau\text{-to-β-var} \rangle ( \langle \tau\text{-var} \rangle ) : \text{NEWLINE} \langle \text{suite} \rangle \)

c) \( ⟨\text{delim-stmt}⟩ \rightarrow \text{def} \langle \tau \times α\text{-to-β-var} \rangle ( \langle \tau\text{-var} \rangle , \langle α\text{-var} \rangle ) : \text{NEWLINE} \langle \text{suite} \rangle \)

Member variables and functions can be invoked using ‘.’ (e.g. \( \text{this.desiredLength} \)):

(d) \( ⟨α\text{-var}⟩ \rightarrow ⟨\tau\text{-expr}⟩ . \langle α\text{-var}⟩ \)

e) \( ⟨β\text{-expr}⟩ \rightarrow ⟨\tau\text{-expr}⟩ . \langle τ\text{-to-β-var}⟩ ( \) \)

(f) \( ⟨β\text{-expr}⟩ \rightarrow ⟨\tau\text{-expr}⟩ . \langle τ \times α\text{-to-β-var}⟩ ( \langle α\text{-expr} \rangle ) \)

For example:

```python
class ListLengthDetector:
    def setup(this, n):
        this.desiredLength = n
    def detect(this, l):
        return ( len(l) == this.desiredLength )
```

Class instances can then be constructed using the class name:

(g) \( ⟨τ\text{-expr}⟩ \rightarrow ⟨τ\text{-type-id}⟩ ( \)

For example (after the class definition above):

```python
mydetector = ListLengthDetector()
mydetector.setup(4)
print ( mydetector.detect(['first','second','third','fourth']) )
```

will print:

```
True
```

3
6. **Practice**
   Write a class `ListCopyDetector` to recognize a list of $n$ copies of a string $s$.

7. **A Useful Tree Class**
   Sample class for reading/writing syntax trees:

   ```python
   import re
   import sys

   # a Tree consists of a category label 'c' and a list of child Trees 'ch'
   class Tree:
       # obtain tree from string
       def read(this, s):
           this.ch = []
           # a tree can be just a terminal symbol (a leaf)
           m = re.search('^ *([^()]*) *(.*)', s)
           if m != None:
               this.c = m.group(1)
               return m.group(2)
           # a tree can be an open paren, nonterminal symbol, subtrees, close paren
           m = re.search('^ *)\( *([^)]*) *(.*)', s)
           if m != None:
               this.c = m.group(1)
               s = m.group(2)
               while re.search('^ *)\)', s) == None:
                   t = Tree()
                   line = t.read(s)
                   this.ch = this.ch + [t]
               return re.search('^ *)(.*)', s).group(1)
           return ''

       # obtain string from tree
       def str(this):
           if this.ch == []:
               return this.c
           s = '(' + this.c
           for t in this.ch:
               s = s + ' ' + t.str()
           return s + ')

   Sample code to read/write syntax trees:

   # for each line in input
   for line in sys.stdin:
       # for each tree in line
       while line != '':
           t = Tree()
           line = t.read(line)
           print t.str()
   ```

Run this on a file containing a bracketed tree:
( S ( NP the cat ) ( VP slept ) )

And it will print back the same tree, neatened up:

(S (NP the cat) (VP slept))

Here’s the tree:

8. Inheritance

It’s often useful to base a new class \( \tau \) on one or more existing classes (superclasses) \( \sigma \):

(a) \( \langle \text{delim-stmt} \rangle \rightarrow \text{class} \langle \tau\text{-type-id} \rangle \ (\langle \sigma\text{-type-id} \rangle) : \text{NEWLINE} \langle \text{suite} \rangle \)

This allows your new class to inherit all the methods of the superclass.

A particular member function ‘\_init\_’ is responsible for creating new instances of a class:

(b) \( \langle \text{delim-stmt} \rangle \rightarrow \text{def} \ _\text{init}\_ \ (\langle \tau\text{-var} \rangle) : \text{NEWLINE} \langle \text{suite} \rangle \)

(c) \( \langle \text{delim-stmt} \rangle \rightarrow \text{def} \ _\text{init}\_ \ (\langle \tau\text{-var} \rangle, \langle \alpha\text{-var} \rangle) : \text{NEWLINE} \langle \text{suite} \rangle \)

For example, define class ‘Model’ (in file ‘model.py’) to refine i/o behavior of a dictionary:

```python
import re
# define distribution to map value tuples to probabilities (or frequencies or scores,
class Model(dict):
    # init with model id
    def __init__(this,i):
        this.id = i
    # read model
    def read(this,s):
        m = re.search('^ *' + this.id + ': +(. *) += +(. *) *', s)
        if m is not None:
            v = tuple(re.split(' +', m.group(1)))
            if len(v) == 1: v = v[0]
            this[v] = float(m.group(2))
    # write model
    def write(this):
        for v in sorted(this):
            if this[v] > 0.0:
                print this.id, v, this[v]
            else:
                print v, this[v]
```
New instances of the class are invoked using the class name (via ‘\_init\_’ function, if any):

(d) \(\langle \tau\text{-expr} \rangle \rightarrow \langle \tau\text{-type-id} \rangle \ (\langle \alpha\text{-expr} \rangle \ )\)

Now we can read a model with only a single command:

```python
import sys
import model
m = model.Model('M')
for line in sys.stdin:
    m.read(line)
```

Run this on a file containing an FSA model:

```
M : q0 a q0 = 1.0
M : q0 b q1 = 1.0
```

And it will print back the same model, neatened up:

```
M : q0 a q0 = 1.0
M : q0 b q1 = 1.0
```

‘Derived’ classes (derived from superclasses) allow superclass methods to be overridden. E.g. modify the default behavior of the dict so it initializes entries for queried keys:

```python
import re
class Model(dict):
    # populate with default values when queried on missing keys
    def __missing__(this,k):
        this[k]=0.0
        return this[k]
    # define get without promiscuity, using ordinary dictionary method
    def get(this,k):
        return dict.get(this,k,0.0)
    # init with model id
    def __init__(this,i):
        this.id = i
    # read model
    def read(this,s):
        m = re.search('ˆ * '+this.id+' +: +(. * ) += +(. * ) * ',s)
        if m is not None:
            v = tuple(re.split(' +',m.group(1))
            if len(v)==1: v = v[0]
            this[v] = float(m.group(2))
    # write model
    def write(this):
        for v in sorted(this):
            if this[v]>0.0:
                print this.id,
                print ' :',
            if type(v) is tuple:
                for f in v:
                    print f,
            else: print v,
            print ' = ',this[v]
```

6
Sample run:

```python
>>> import model
>>> m = model.Model('M')
>>> m['a'] = 1  # adds 'a' and sets value
>>> m  # see?
{'a': 1}
>>> m['b']  # adds 'b' with default value
0.0
>>> m  # see?
{'a': 1, 'b': 0.0}
>>> m.get('c')  # does not add 'c'
0.0
>>> m  # see?
{'a': 1, 'b': 0.0}
```