Ling 5801: Lecture Notes 9
From Recursive Types to Recursive Functions

Contents

9.1 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:

```
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:

1. $\langle$stmt$\rangle$ $\rightarrow$ `def` $\langle\alpha$-to-β-var$\rangle$ `(` $\langle\alpha$-var$\rangle$ `)` `:` NEWLINE $\langle$suite$\rangle$

   define a function from expressions of type α to expressions of type β

2. $\langle$stmt$\rangle$ $\rightarrow$ `return` $\langle\beta$-expr$\rangle$

   return an expressions of appropriate type at the end of the function suite

3. $\langle\beta\text{-expr}\rangle \rightarrow \langle\alpha\text{-to-}\beta\text{-expr}\rangle$ ($\langle\alpha\text{-expr}\rangle$)

   apply a function to an argument of appropriate type

For example:

```
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')
```

9.2 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.

9.3 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)

Practice

Write a recursive function `conc(n,s)` that concatenates together an `n`-length sequence of `s`'s, observing that this is simply an `s` concatenated with an `n-1`-length sequence of `s`'s.

9.4 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:

1. $\langle$stmt$\rangle$ $\rightarrow$ `class` $\langle\tau$-type-id$\rangle$ **:** NEWLINE $\langle$suite$\rangle$

(where $\langle\tau$-type-id$\rangle$ is a class name, like '`Greeting`')

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

2. $\langle$stmt$\rangle$ $\rightarrow$ `def` $\langle\tau$-to-β-var$\rangle$ `(` $\langle\tau$-var$\rangle$ `)` **:** NEWLINE $\langle$suite$\rangle$
3. $\langle$stmt$\rangle$ $\rightarrow$ `def` $\langle\tau\times\alpha$-to-β-var$\rangle$ `(` $\langle\tau$-var$\rangle$ **,** $\langle\alpha$-var$\rangle$ `)` **:** NEWLINE $\langle$suite$\rangle$

Class instances can then be ***constructed*** using the class name:

4. $\langle\tau$-expr$\rangle$ $\rightarrow$ $\langle\tau$-type-id$\rangle$ `( )`

   If you have a method `__init__` with parameter $\langle\tau$-var$\rangle$, it will execute here.
5. $\langle\tau$-expr$\rangle$ $\rightarrow$ $\langle\tau$-type-id$\rangle$ `(` $\langle\alpha$-var$\rangle$ `)`

   If you have a method `__init__` with parameters $\langle\tau$-var$\rangle$ and $\langle\alpha$-var$\rangle$, it will execute here.

For example, if you define a class:

```
class Greeting:
  def __init__(this):
    print( 'How do you do?' )
```

and create an instance of it with this constructor:

```
m = Greeting()
```

then it will greet you:

```
How do you do?
```

We can add ***member variables*** to objects using '.' (e.g. `this.numLetters`):

6. $\langle \alpha\text{-var} \rangle \;\rightarrow\; \langle \tau\text{-expr} \rangle$ `.` $\langle \alpha\text{-var} \rangle$

For example:

```
class Word:
  def __init__(this,s):
    this.numLetters = len(s)

w = Word('dog')
print( w.numLetters )
```

will print:

```
3
```

We can also refer to ***methods*** using '.' (e.g. `this.write`):

7. $\langle \beta\text{-expr} \rangle \;\rightarrow\; \langle \tau\text{-expr} \rangle$ `.` $\langle \tau\text{-to-}\beta\text{-var} \rangle$ `( )`

8. $\langle \beta\text{-expr} \rangle \;\rightarrow\; \langle \tau\text{-expr} \rangle$ `.` $\langle \tau \times \alpha\text{-to-}\beta\text{-var} \rangle$ `(` $\langle \alpha\text{-expr} \rangle$ `)`

For example:

```
class Greeting:
  def write(this):
    print( 'How do you do?' )

w = Greeting()
w.write()
```

will print:

```
How do you do?
```

Practice

Write a class `Word` that takes a string `s` in its constructor and has a method `getString` that returns the string and a method `countLetters` that reports the number of letters in the string.

9.5 A Useful Tree Class

Sample class for reading/writing syntax trees:

```
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()
                s = 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:

```
c :'S'
ch: [ c :'NP'                                    ,  c :'VP'                ]
      ch: [ c :'the'   ,  c :'cat'  ]               ch: [ c :'slept' ]
            ch:[]         ch:[]                           ch:[]
```

9.6 Inheritance

It's often useful to base a new class τ on one or more existing classes (superclasses) σ:

1. ⟨stmt⟩ → `class` ⟨τ-type-id⟩ `(` ⟨σ-type-id⟩ `)` **:** NEWLINE ⟨suite⟩

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

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

```
# define distribution to map value tuples to probabilities, frequencies or scores
class Model(dict):
    # init with model id
    def __init__(self,i):
        self.id = i
    # read model
    def read(self,s):
        m = re.search('^ *'+self.id+' +: +(.*) += +(.*) *$',s)
        if m is not None:
            v = tuple(re.split(' +',m.group(1)))
            if len(v)==1: v = v[0]
            self[v] = float(m.group(2))
    # write model
    def write(self):
        for v in sorted(self):
            s = self.id
            s = s + ' :'
            if type(v) is tuple:
                for f in v:
                    s = s + ' ' + f
            else: s = s + ' ' + v
            print( s + ' = ' + str(self[v]) )
```

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

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

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:

```
# define distribution to map value tuples to probabilities, frequencies or scores
class Model(dict):
    # populate with default values when queried on missing keys
    def __missing__(self,k):
        self[k]=0.0
        return self[k]
    # define get without promiscuity, using ordinary dictionary method
    def get(self,k):
        return dict.get(self,k,0.0)
    # init with model id
    def __init__(self,i):
        self.id = i
    # read model
    def read(self,s):
        m = re.search('^ *'+self.id+' +: +(.*) += +(.*) *$',s)
        if m is not None:
            v = tuple(re.split(' +',m.group(1)))
            if len(v)==1: v = v[0]
            self[v] = float(m.group(2))
    # write model
    def write(self):
        for v in sorted(self):
            s = self.id
            s = s + ' :'
            if type(v) is tuple:
                for f in v:
                    s = s + ' ' + f
            else: s = s + ' ' + v
            print( s + ' = ' + str(self[v]) )
```

Sample run:

```
>>> 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}
```