Ling 5801: Lecture Notes 5
From Programs to Projects

Contents

5.1 Modules 1
5.2 Input streams 1
5.3 Regular expressions 2
5.4 Dictionaries 3
5.5 Tuples 4
5.6 Implementation of FSA 5

5.1 Modules

You can build projects with shared code files

Other code files (written by you or others) can be 'imported' into a program

1. ⟨stmt⟩ → `import` ⟨modulename⟩

   allow access to constants or commands defined in ⟨modulename⟩.py

   the constants or commands must then be preceded by '⟨modulename⟩.'

For example, if you have some constant 'PI' in a file called '`myconst.py`':

```
PI = 3.1415
```

you can access it within another .py file by importing it as a *module*:

```
import myconst
print ( myconst.PI )
```

to print:

```
3.1415
```

5.2 Input streams

Projects can also share data in files, read by piping to standard input

Standard input (e.g. from a unix pipe or user typing) is a 'stream': potentially infinite list

1. ⟨string-list-expr⟩ → `sys.stdin`

   input stream is unbounded list of input lines (program pauses if line yet to be created)

(when typing, you can signal an end of input by typing CONTROL-D)

It requires you to import the module ‘sys’ (not in your directory; Python knows where it is)

For example:

```
import sys
for line in sys.stdin:
  print ( 'I got:  '  + line )
```

will print each line you type (or each line of a piped-in file) preceded by ‘I got:’

In a unix terminal, if the above is in a file called ‘`myprog.py`’, you can type:

```
echo 'This is a test.'  | python myprog.py
```

to see each line preceded by ’I got:’

5.3 Regular expressions

Formatted input lines can be parsed into variable values using regular expressions

1. ⟨match-expr⟩ → `re.search(` ⟨string-expr⟩ `,` ⟨string-expr⟩ `)`

   create a match data structure with parts of second ⟨string-expr⟩ matching ‘(...)’ in first
2. ⟨match-expr⟩ → `None`

   searches can also fail, returning ‘None’ constant
3. ⟨string-expr⟩ → ⟨match-expr⟩ `.group(` ⟨num-expr⟩ `)`

   obtain the substring matching the ⟨num-expr⟩-th group in ⟨match⟩

   (group 0 is the entire string)

It requires you to import the module ‘re’ (not in your directory; Python knows where it is)

For example (a ‘lemmatizer’):

```
import sys
import re
for line in sys.stdin:
  m = re.search('(.*)ing',line)
  if not m == None:
    print ( 'The base form is: '  + m.group(1) )
```

will read in words, e.g.:

```
eating
```

and print the lemma or base form:

```
The base form is: eat
```

Some more fancy regular expression commands:

4. $\langle$string-list-expr$\rangle$ $\rightarrow$ `re.findall(` $\langle$string-expr$\rangle$ `,` $\langle$string-expr$\rangle$ `)`

   obtain list of substrings of the second $\langle$string-expr$\rangle$ matching the first $\langle$string-expr$\rangle$
5. $\langle$string-list-expr$\rangle$ $\rightarrow$ `re.split(` $\langle$string-expr$\rangle$ `,` $\langle$string-expr$\rangle$ `)`

   obtain list of substrings of the second $\langle$string-expr$\rangle$ separated by the first $\langle$string-expr$\rangle$
6. $\langle$string-expr$\rangle$ $\rightarrow$ `re.sub(` $\langle$string-expr$\rangle$ `,` $\langle$string-expr$\rangle$ `,` $\langle$string-expr$\rangle$ `)`

   obtain copy of third $\langle$string-expr$\rangle$ with every first $\langle$string-expr$\rangle$ replaced with second

   (replacing '`\\1`', '`\\2`' in second $\langle$string-expr$\rangle$ with matches to parenthesized groups in first)

Practice

1. Write a 'mad libs' program to read two nouns from standard input, e.g.:

   ```
   banana left-turn-lane
   ```

   and insert them into a hilarious sentence, e.g.:

   ```
   Magellan sailed around the banana and discovered a left-turn-lane
   ```
2. What would your program do on the following input?

   ```
   banana left turn lane
   ```

5.4 Dictionaries

We can look up data for non-numerical modeled values using a 'dictionary.'

(It's like a list, but can be indexed on non-integers.)

1. $\langle\alpha$-to-β-dict-expr$\rangle$ $\rightarrow$ `{ }`
2. $\langle\alpha$-to-β-dict-expr$\rangle$ $\rightarrow$ `{` $\langle\alpha$-to-β-dict-entry-seq$\rangle$ `}`
3. $\langle\alpha$-to-β-dict-entry-seq$\rangle$ $\rightarrow$ $\langle\alpha$-expr$\rangle$ `:` $\langle\beta$-expr$\rangle$
4. $\langle\alpha$-to-β-dict-entry-seq$\rangle$ $\rightarrow$ $\langle\alpha$-expr$\rangle$ `:` $\langle\beta$-expr$\rangle$ `,` $\langle\alpha$-to-β-dict-entry-seq$\rangle$

   dicts may contain sequences of expressions mapped to other expressions

And here are some things we can do with them:

5. $\langle\beta$-var$\rangle$ $\rightarrow$ $\langle\alpha$-to-β-dict-var$\rangle$ `[` $\langle\alpha$-expr$\rangle$ `]`

   dict elements can be indexed by *any* type of expression (unlike list)

   For example:

```
wordtypes = { 'sneeze' : 'verb' }
wordtypes['bird'] = 'noun'
print ( wordtypes['bird'] )
```

6. $\langle\beta$-expr$\rangle$ $\rightarrow$ $\langle\alpha$-to-β-dict-var$\rangle$ `.get (` $\langle\alpha$-expr$\rangle$ `,` $\langle\beta$-expr$\rangle$ `)`

   obtain element indexed by $\langle\alpha$-expr$\rangle$, or $\langle\beta$-expr$\rangle$ if not found

   For example:

   ```
   wordtypes = { 'sneeze' : 'verb' }
   wordtypes['bird'] = 'noun'
   print ( wordtypes.get('whiffle','No such word.') )
   ```

   (If you just use brackets, it will be grammatical, but you will get a run-time error!)

7. $\langle$bool-expr$\rangle$ $\rightarrow$ $\langle\alpha$-expr$\rangle$ `in` $\langle\alpha$-to-β-dict-expr$\rangle$

   a boolean can indicate true if $\langle\alpha$-expr$\rangle$ is in $\langle\alpha$-to-β-dict-expr$\rangle$, false otherwise

8. $\langle$bool-expr$\rangle$ $\rightarrow$ $\langle\alpha$-expr$\rangle$ `not in` $\langle\alpha$-to-β-dict-expr$\rangle$

   a boolean can indicate false if $\langle\alpha$-expr$\rangle$ is in $\langle\alpha$-to-β-dict-expr$\rangle$, true otherwise

9. $\langle\alpha$-list-expr$\rangle$ $\rightarrow$ $\langle\alpha$-to-β-dict-expr$\rangle$

   a list can be defined by a dictionary (it will be the list of keys in the dictionary)

   For example:

   ```
   wordtypes = { 'sneeze' : 'verb' }
   wordtypes['bird'] = 'noun'
   for x in wordtypes :
     print ( x + ' is a ' + wordtypes[x] )
   ```

Practice:

Write a 'pet minder' program that reads sentences and questions from standard input in the following form:

```
Squeaky is the mouse
Woofy is the dog
Meowy is the cat
who is Squeaky?
```

then answers the question(s) based on the sentences read up to that point, e.g.:

```
the mouse
```

5.5 Tuples

Python supports tuples — sequences of expressions separated by commas:

1. $\langle\alpha$-cross-β-expr$\rangle$ $\rightarrow$ $\langle\alpha$-expr$\rangle$ `,` $\langle\beta$-expr$\rangle$

2. $\langle\alpha$-cross-β-var$\rangle$ $\rightarrow$ $\langle\alpha$-var$\rangle$ `,` $\langle\beta$-var$\rangle$

The elements in these tuples can be indexed like lists (starting with zero):

3. $\langle\beta$-expr$\rangle$ $\rightarrow$ $\langle\alpha$-cross-$\ldots$-cross-γ-expr$\rangle$ `[` $\langle$num-expr$\rangle$ `]`

   where $\langle\beta$-expr$\rangle$ is the type of the $\langle$num-expr$\rangle$th element in the tuple

For example:

```
t = 7,'is lucky'
print ( t[0] )
```

will print:

```
7
```

5.6 Implementation of FSA

We can now update our FSA recognizer to read models and inputs from files

(which we will cat to standard input, as in our 'pet minder' example)

Sample FSA model file 'petfsa.model':

```
S qHappy
F qHungry
M qHappy burble qHappy
M qHappy whiffle qHungry
M qHungry burble qHungry
```

Sample FSA input file 'burble.input':

```
I burble whiffle burble
```

Sample Python program 'fsarec.py' implementing FSA:

```
# import standard input and regular expression modules
import sys
import re

# initialize FSA parameters as dictionaries
Q = {}
S = {}
F = {}
M = {}
Input = []

# for each line of input in model file
for line in sys.stdin:
   m = re.search('([^ ]+) (.*)',line) # identify FSA param and value
   if not m == None:
      if m.group(1)=='S':                  # if start state, add to S
         S[m.group(2)] = True
      if m.group(1)=='F':                  # if final state, add to F
```

```
         F[m.group(2)] = True
      if m.group(1)=='M':                # if trans model tuple, add to M
         T = re.split(' +',m.group(2))     # isolate tuple elements
         M[T[0],T[1],T[2]] = True          # update M
         Q[T[0]] = True                    # update Q
         Q[T[2]] = True
      if m.group(1)=='I':                # if input, set as Input (starting at t=1)
         Input = ['-'] + re.split(' +',m.group(2))

# initialize table of possible states at time step 0 using start states
V = {}
for q in Q:
   V[0,q] = S.get(q,False)

# for each possible state qP in V at time t-1, for each qP,x,q in M, add q
for t in range(1,len(Input)):
   for qP in Q:
      for q in Q:
         V[t,q] = V.get((t,q),False) or (V[t-1,qP] and M.get((qP,Input[t],q),False))

# if any final states possible at end, accept
for q in F:
   if V[len(Input)-1,q] and F[q]:
      print ( 'yes' )
```

You can run this by catting the data files into the python interpreter:

```
cat petfsa.model burble.input | python fsarec.py
```

Or, here's a sample make item:

```
.SECONDEXPANSION:
%.output: $$(basename %).input $$(subst .,,$$(suffix %)).model fsarec.py
        cat $(word 2,$^) $< | python $(word 3,$^) > $@
```

It works on the following command:

```
make burble.petfsa.output
```