Introduction to Python

Lecture #3

Computational Linguistics

CMPSCI 591N, Spring 2006

University of Massachusetts Amherst

Andrew McCallum

Today’s Main Points

- Check in on HW#1.
- Intro to Python computer programming language.
- Some examples Linguistic applications.
- The NLTK toolkit.
- Pointers to more Python resources.

Python Outline

- Introduction
  - Python attributes and ‘Why Python?’
  - Running programs
  - Modules
- Basic object types
  - Numbers and variables
  - Strings
  - Lists, Tuples
  - Dictionaries
- Control Flow
  - Conditionals
  - Loops

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Python Features

- Free. Runs on many different machines.
- Easy to read.
  - Perl = “write only language”
- Quick to throw something together.
  - NaiveBayes Java vs Python
- Powerful. Object-oriented.

- THE modern choice for CompLing.
- NLTK

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Using Python Interactively

The easiest way to give Python a whirl is interactively.
(Human typing in red. Machine responses in black.)

```
$ python
>>> print “Hello everyone!”
Hello everyone!
>>> print 2+2
4
>>> myname = “Andrew”
>>> myname
‘Andrew’
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Modules

To save code you need to write it in files.
Module: a text file containing Python code.

Example: write the following to file **`foo.py`**

```
print 25*3                              # multiply by 3
print ‘CompLing ‘ + ‘lecture 3’         # concatenate with +
myname = ‘Andrew’
```

(No leading spaces!)

Then run it as follows:

```
$ python foo.py
75
CompLing lecture 3
$
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Importing Modules

Every file ending in **`.py`** is a Python module.
Modules can contain attributes such as functions.
We can import this module into Python.

```
$ python
>>> import foo
75
CompLing lecture 3
>>> foo.myname
‘Andrew’
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Module Reloading

Importing is expensive--after the first import of a module, repeated imports have no effect (even if you have edited it).

Use **`reload`** to force Python to rerun the file again.

```
>>> import foo
75
CompLing lecture 3
```

Edit foo.py to print 25*4 (instead of 25*3) and reload

```
>>> reload(foo)
75
CompLing lecture 3
<module ‘foo’ from ‘foo.py’>
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Module Attributes

Consider file **`bar.py`**

```
university = ‘UMass’
department = ‘Linguistics’
```

```
>>> import bar
>>> print bar.department
Linguistics

>>> from bar import department
>>> print department
Linguistics

>>> from bar import *
>>> print university
UMass
```

`from` copies named attributes from a module, so they are variables in the recipient.

Python Program Structure

- Programs are composed of modules
- Modules contain statements
- Statements contain expressions
- Expressions create and process objects

- Statements include
  - variable assignment, function calls
  - control flow, module access
  - building functions, building objects
  - printing

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Python’s built-in objects

- Numbers: integer, floating point
- Strings
- Lists
- Dictionaries
- Tuples
- Files

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Numbers and Variables

- Usual number operators, e.g: `+, *, /, **`
- Usual operator precedence:
  `A * B + C * D = (A * B) + (C * D)`
  (use parens for clarity and to reduce bugs)
- Useful modules: `math, random`

- Variables
  - created when first assigned a value
  - replaced with their values when used in expressions
  - must be assigned before use
  - no need to declare ahead of time

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Strings

- String handling in Python is easy and powerful (unlike C, C++, Java)
- Strings may be written using single quotes: `‘This is a Python string’`
- or double quotes `“and so is this”`
- They are the same, it just makes it easy to include single (or double) quotes: `‘He said “what?”’` or `“He’s here.”`

(Learning Python, chapter 5)

Backslash in strings

Backslash \ can be used to escape (protect) certain non-printing or special characters.

For example, **\n** is newline, **\t** is tab.

```
>>> s = ‘Name\tAge\nJohn\t21\nBob\t44’
>>> print s
Name        Age
John        21
Bob         44
>>> t = ‘”Mary\’s”’
>>> print t
“Mary’s”
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Triple quote

Use a triple quote (“”” or ‘’’) for a string over severa lines:

```
>>> s = “””this is
... a string
... over 3 lines”””
>>> t = ‘’’so
... is
... this’’’
>>> print s
this is
a string
over 3 lines
>>> print t
so
is
this
```

String operations

- Concatenation (+)
- Length (len)
- Repetition (*)
- Indexing and slicing ([ ])

```
s = ‘computational’
t = ‘linguistics’
cl = s + ‘ ‘ + t        # ‘computational linguistics’
l = len(cl)             # 25
u = ‘-’ * 6             # ------
c = s[3]                # p
x = cl[11:16]           # ‘al li’
y = cl[20:]             # ‘stics’
z = cl[:-1]             # ‘computational linguistic’
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

String methods

- ➢Methods are functions applied to and associated with objects
- ➢String methods allow strings to be processed in a more sophisticated way

```
s = ‘example’
s = s.capitalize()            # ‘Example’
t = s.lower()                 # ‘example’
flag = s.isalpha()            # True
s = s.replace(‘amp’,‘M’)      # ‘exMle’
i = t.find(‘xa’)              # 1
n = t.count(‘e’)              # 2
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Lists in Python

- Ordered collection of arbitrary objects
- Accessed by indexing based on offset from start
- Variable length (grows automatically)
- Heterogeneous (can contain any type, nestable)
- Mutable (can change the elements, unlike strings)

```
>>> s = [‘a’, ‘b’, ‘c’]
>>> t = [1, 2, 3]
>>> u = s + t                  # [‘a’, ‘b’, ‘c’, 1, 2, 3]
>>> n = len(u)                 # 6
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Indexing and slicing lists

- Indexing and slicing work like strings
- Indexing returns the object at the given offset
- Slicing returns a list
- Can use indexing and slicing to change contents

```
l = [‘a’, ‘b’, ‘c’, ‘d’]
x = l[2]                        # ‘c’
m = l[1:]                       # [‘b’, ‘c’, ‘d’]
l[2] = ‘z’                      # [‘a’, ‘b’, ‘z’, ‘d’]
l[0:2] = [‘x’, ‘y’]             # [‘x’, ‘y’, ‘z’, ‘d’]
```

(Learning Python, chapter 6)

List methods

- Lists also have some useful methods
- append adds an item to the list
- extend adds multiple items
- sort orders a list in place

```
l = [7, 8, 9, 3]
l.sort ()                 # [3, 7, 8, 9]
l.append(6)               # [3, 7, 8, 9, 6]
l.append([1, 2])          # [3, 7, 8, 9, [1, 2]]
l.extend([‘r’, ‘s’])      # [3, 7, 8, 9, [1, 2],‘r’, ‘s’]
```

(Learning Python, chapter 6)

Dictionaries

Dictionaries are

- Address by *key*, not by offset
- *Unordered collections* of arbitrary objects
- *Variable length*, heterogeneous
  (can contain contain any type of object), nestable
- *Mutable* (can change the elements, unlike strings)

- Think of dictionaries as a set of key:value pairs
- Use a key to access its value

(Learning Python, chapter 7)

Dictionary example

```
level = {‘low’:1, ‘medium’:5}
x = level[‘medium’]           # 5
n = len(level)                # 2

flag = level.has_key(‘low’)   # True
l = level.keys()              # [‘low’,‘medium’]

level[‘low’] = 2      # {‘low’:2, ‘medium’:5}
level[‘high’] = 10    # {‘low’:2, ‘high’:10, ‘medium’:5}

level.items()
[(‘low’,2), (‘high’,10), (‘medium’,5)]

level.values()
[2, 10, 5]
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Notes on dictionaries

- Sequence operations don’t work (e.g. slice) dictionaries are mappings, not sequences.
- Dictionaries have a set of keys:
  only one value per key.
- Assigning to a new key adds an entry
- Keys can be any immutable object, not just strings.

- Dictionaries can be used as records
- Dictionaries can be used for sparse matrices.

Other objects

Tuples: list lists, but immutable (cannot be changed)

```
emptyT = ()
t1 = (1, 2, 3)
x = t1[1]              # 2
n = len(t1)            # 3
y = t1[1:]             # (2, 3)
```

Files: objects with methods for reading and writing to files

```
file = open(‘myfile’, ‘w’)
file.write(‘hellow file\n’)
file.close()

f2 = open(‘myfile’, ‘r’)
s = f2.readline()               # ‘hello file\n’
t = f2.readline()               # ‘’
all = open(‘myfile’).read()     #entire file as a string
```

(Learning Python, chapter 7)

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Conditionals: if tests

```
course = ‘Syntax’
if course == ‘Syntax’:
  print ‘Bhatt’
  print ‘or Potts’
elif course == ‘Computational Linguistics’:
  print ‘McCallum’
else:
  print ‘Someone else’
```

- Indentation determines the block structure
  Indentation to the left is the only place where whitespace matters in Python
- Indentation enforces readability
- Tests after **`if`** and **`elif`** can be just about anything:
  `False, 0, (), [], ‘’`, all count as false
  Other values count as true.

(Learning Python, chapter 9)

while loops

A while loop keeps iterating while the test at the top remains True.

```
a = 0
b = 10
while a < b:
  print a
  a = a + 1
```

```
s = ‘abcdefg’
while len(s) > 0:
  print s
  s = s[1:]
```

(Learning Python, chapter 10)

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

for loops

`for` is used to step through any sequence object

```
l = [‘a’, ‘b’, ‘c’]
for i in l:
   print i

sum = 0
for x in [1, 2, 3, 4, 5, 6]:
   sum = sum + x
print sum
```

`range()` is a useful function:

```
range(5)          # [0, 1, 2, 3, 4]
range(2,5)        # [2, 3, 4]
range(0,6,2)      # [0, 2, 4]
```

(Learning Python, chapter 10)

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

for loops with style

Do something to each item in a list (e.g. print its square)

```
l = [1, 2, 3, 4, 5, 6]  # or l = range(1,7)

# one way to print the square
for x in l:
  print x*x

# another way to do it
n = len(l)
for i in range(n):
  print l[i]*l[i]
```

Which is better?

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Example: intersecting sequences (Keyword *in*)

The intersection of
`[‘a’, ‘d’, ‘f’, ‘g’]` and `[‘a’, ‘b’, ‘c’, ‘d’]`
is `[‘a’, ‘d’]`

```
l1 = [‘a’, ‘d’, ‘f’, ‘g’]
l2 = [‘a’, ‘b’, ‘c’, ‘d’]
# one way
result = []
for x in l1:
  for y in l2:
      if x == y:
            result.append(x)
# or, alternatively
result = []
for x in l1:
  if x in l2:
      result.append(x)        # result == [‘a’, ‘d’]
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Built-in, imported and user-defined functions

- Some functions are built-in, e.g.

```
l = len([‘a’, ‘b’, ‘c’])
```

- Some functions may be imported, e.g.

```
import math
from os import getcwd
print getcwd() # which directory am I in?
x = math.sqrt(9)    # 3
```

- Some functions are user-defined, e.g.

```
def multiply(a, b):
     return a * b
print multiply(4,5)
print multiply(‘-’,5)
```

Functions in Python

- Functions are a way to group a set of statements that can be run more than once in a program.
- They can take parameters as inputs, and can return a value as output.
- Example

```
def square(x):      # create and assign
     return x*x
y = square(5)       # y gets 25
```

- `def` creates a function object, and assigns it to a name
- return sends an object back to the caller
- Adding () after the function’s name calls the function.

(Learning Python, chapter 12)

Intersection function

```
def intersect(seq1, seq2)
     result = []
     for x in seq1:
          if x in seq2:
               result.append(x)
     return result
```

- Putting the code in a function means you can run it many times.
- General -- callers pass any 2 sequences
- Code is in one place. Makes changing it easier (if you have to)

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Local variables

Variables inside a function are *local* to that function.

```
>>> intersect(s1, s2):
...   result = []
...   for x in s1:
...         if x in s2:
...               result.append(x)
...   return result
...
>>> intersect([1,2,3,4], [1,5,6,4])
[1, 4]
>>> result
Traceback (most recent call last):
  File “<stdin>”, line 1, in ?
NameError: name ‘result’ is not defined
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Argument passing

Arguments are passed by assigning objects to local names.

```
>>> def plusone(x):
...     x = x + 1
...     return x
...
>>> plusone(3)
4
>>> x = 6
>>> plusone(x)
7
>>> x
6
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Passing mutable arguments

Recall that numbers, strings, tuples are **immutable**, and that lists and dictionaries are **mutable**:

```
>>> def appendone(s):
...   s.append(‘one’)
...   return s
...
>>> appendone([‘a’, ‘b’])
[‘a’, ‘b’, ‘one’]
>>> l = [‘x’, ‘y’]
>>> appendone(l)
[‘x, ‘y’, ‘one’]
>>> l
[‘x’, ‘y’, ‘one’]
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

map

```
>>> counters = range(1,6)
>>> updated = []
>>> for x in counters:
...   updated.append(x+3)
...
>>> updated
[4, 5, 6, 7, 8]

# Another way...
>>> def addthree(x):
...   return x+3
...
# map() applies a function to all elements of a list
>>> map(addthree, counters)
[4, 5, 6, 7, 8]
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Anonymous functions and list comprehensions

```
# lambda is a way to define a function with no name
>>> map((lambda x: x+3), counters)
[4, 5, 6, 7, 8]


# a list comprehension does something similar,
# but can offer more flexibility
>>> result = [addthree(x) for x in counters]
>>> result
[4, 5, 6, 7, 8]
>>> [addthree(x) for x in counters if x < 4]
[4, 5, 6]
```

Also check out **`apply`**, **`filter`**, and **`reduce`**.

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Variable number of arguments

Sometimes you don’t know how many arguments a function will receive.

`*a` receives them in a list.

```
def max (*a):
  maximum = 9999999
  for x in a:
      if a > maximum:
          maximum = a
  return maximum
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Optional (named) arguments

Sometimes you want to define a function with optional argument.

Give a name and a default value.

```
def exp (x, exponent=2.718):
  return exponent ** x
```

```
>>> exp(1)
2.718
>>> exp(1, 2.0)
2.0
>>> exp(3, 2.0)
8.0
>>> exp(3, exponent=2.0)
8.0
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Multiple optional arguments

If multiple optional arguments are given, you can pass some and not others.

```
def exp_plus (x, exponent=2.718, addend=0):
  return (exponent ** x) + addend
```

```
>>> exp(1)
2.718
>>> exp(1, 2.0)
2.0
>>> exp(1, exponent=2.0)
2.0
>>> exp(1, addend=2.0)
4.718
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Arbitrary number of named arguments

The **argument notation receives all extra arguments in a dictionary.

```
def showargs (separator, **d):
   for key in d.keys():
        print str(key)+”:“+str(d[key])+separator,
   print
```

```
>>> showargs(“;”, bi=2, tri=3, quad=4)
tri:3;bi:2;quad:4;
```

(Or another way with an assignment to two variables at once!)

```
def showargs (separator, **d):
   for (key,val) in d.items():
        print str(key)+”:“+str(val)+separator,
   print
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Guido van Rossum

Grew up in the Netherlands.

“December 1989, I was looking for a ‘hobby’ programming project that would keep me occupied during the week around Christmas....”
...Python 2.4... NASA, WWW infrastructure, Google...

In December 2005, hired by Google.

Useful module: re

Regular expressions

```
import re

r = re.compile(r’\bdis(\w+)\b’)
s = ‘Then he just disappeared.’
match = r.search(s)
if match:
  print “Found the regex in the string!”
  print “The prefix was”, match.group(1)
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Useful module: random

Random number generator and random choices

```
>>> import random

>>> random.uniform(0,1)
0.16236

>>> list = [‘first’, ‘second’, ‘third’, ‘fourth’]
>>> random.choice(list)
‘third’
>>> random.choice(list)
‘first’
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

NLTK: Python Natural Language Toolkit

- NLTK is a set of Python modules which you can import into your programs, e.g.:

```
from nltk_lite.utilities import re_show
```

- NLTK is distributed with several corpora.
- Example corpora with NLTK:
  - gutenberg (works of literature from Proj. Gutenberg)
  - treebank (parsed text from the Penn treebank
  - brown (1961 million words of POS-tagged text)
- Load a corpus (eg gutenberg) using:

```
>>> from nltk_lite.corpora import gutenberg
>>> print gutenberg.items
['autsen-emma', 'austen-persuasion',...]
```

Simple corpus operations

- Simple processing of a corpus includes tokenization (splitting the text into word tokens), text normalization (eg by case), and word stats, tagging and parsing.
- Count the number of words in “Macbeth”

```
from nltk_lite.corpora import gutenberg
nwords = 0
for word in gutenberg.raw(‘shakespeare-macbeth’):
    nwords += 1
print nwords
```

- `gutenberg.raw(`textname`)` is an iterator, which behaves like a sequence (eg a list) except it returns elements one at a time as required.

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Richer corpora

- The Gutenberg corpus is tokenized as a sequence of words with no further structure.
- The Brown corpus has sentences marked, and is stored as a list of sentences, where a sentence is a list of word tokens. We can use the extract function to obtain individual sentences

```
from nltk_lite.corpora import brown
from nltk_lite.corpora import extract
firstSentence = extract(0, brown.raw(‘a’))
# [‘The’, ‘Fulton’, ‘County’, ‘Grand’, ‘jury’...]
```

- Part-of-speech tagged text can also be extracted:
  taggedFirstSentence = extract(0, brown.tagged(‘a’))

```
# [(‘The’, ‘at’), (‘Fulton’, ‘np-tl’),
   (‘County’, ‘nn-tl’)...
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

Parsed text

Parsed text from the Penn treebank can also be accessed

```
>>> from nltk_lite.corpora import treebank
>>> parsedSent = extract(0, treebank.parsed())
>>> print parsedSent
>>> print parsedSent
(S:
  (NP-SBJ:
      (NP: (NNP: 'Pierre') (NNP: 'Vinken'))
      (,: ',')
      (ADJP: (NP: (CD: '61') (NNS: 'years')) (JJ: 'old'))
      (,: ','))
  (VP:
      (MD: 'will')
      (VP: (VB: 'join') (NP: (DT: 'the') (NN: 'board'))
  (PP-CLR: (IN: 'as') (NP: (DT: 'a') (JJ: 'nonexecutive')
  (NN: 'director'))) (NP-TMP: (NNP: 'Nov.') (CD: '29'))))
  (.: '.'))
```

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh

More Python Resources

- “Learning Python” book.
- NLTK Python intro for Linguists
  http://nltk.sourceforge.net/lite/doc/en/programming.html
- Others listed at
  “Resources” link on course home page
- Your TAs!

Thank you!

Andrew McCallum, UMass Amherst,
including material from Eqan Klein and Steve Renals, at Univ Edinburghh