Euromasters summer school 2005

Introduction to Python

Trevor Cohn

July 11, 2005

Course Overview

Introduction to Python (Monday)

- language fundamentals
- ~1 hour lecture
- ~2 hour laboratory session

Introduction to NLTK (Tuesday)

- morning lecture & lab
  - tokenisation, tagging, language modelling
- afternoon lecture & lab
  - shallow parsing, CFG parsing, classification

Why Python?

Popular languages for NLP

- Prolog (clean, learning curve, slow)
- Perl (quick, obfuscated, syntax)

Why Python is better suited

- easy to learn, clean syntax
- interpreted, supporting rapid prototyping
- object-oriented (encourages good programming style)
- powerful

Why NLTK?

NLTK: a software package for manipulating linguistic data and performing NLP tasks

- advanced tasks are possible from an early stage
- permits projects at various levels:
  - individual components vs complete systems
- consistent interfaces:
  - sets useful boundary conditions
  - models structured programming
  - facilitates reusability

Python

- Python is an open source scripting language.
  - developed by Guido van Rossum in the early 1990s
  - named after Monty Python
- Version 2.3.4 available on local machines at
  - /usr/bin/python2.3
- Download from www.python.org
- Tutorial:
  - http://www.python.org/doc/tut/tut.htm
- *These slides by Edward Loper, Steven Bird and Trevor Cohn*

Python: Outline

- Data
  - strings, variables, lists, dictionaries
- Control Flow
- Working with files
- Modules
- Functions
- Classes
- etc

The Python Interpreter

- Interactive interface to Python

```
% python2.3
Python 2.3.4 (#1, Sep  6 2004, 13:19:08)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Enter an expression, and Python evaluates it:

```
>>> 3*(7+2)
27
```


Non-interactive mode

- Python can also run on a file

```
% python2.3 myfile.py
27
```

  - Where the file 'myfile.py' contains the single line:

```
print 3*(7+2)
```

- Also: python2.3 -i file
  - enters interactive mode after processing file
- idle2.3
  - interactive GUI for interpreter, editing files and debugging

Strings

- A string is a single piece of text.
  - Strings are written `'...'` or `"..."`

```
>>> "the king of spain"
the king of spain
>>> 'the king said "hello."'
the king said "hello."
```

  - Spaces are significant

```
>>> '  the  knights of  ni '
   the  knights of  ni
```

  - Backslashes mark special characters

```
>>> 'hello\nworld'    # '\n' is a newline
hello
world
```


Operations on Strings

```
>>> 'the' + 'king'
'theking'
>>> len('the df')
6
>>> 'the df'.count('the')
1
>>> 'the king'.replace('the', 'a')
'a king'
>>> 'the king'.upper()
'THE KING'
>>> '  hello   there   '.strip()
'hello   there'
```


Variables

- A variable is a name for a value.
  - Use "=" to assign values to variables.

```
>>> first_name = 'John'
>>> last_name = 'Smith'
>>> first_name + ' ' + last_name
'John Smith'
```

  - Variable names are case sensitive
  - Variable names include only letters, numbers, and "_"
  - Variable names start with a letter or "_"
  - Any variable can hold any value (no typing)

Lists

- A list is an ordered set of values
  - Lists are written $[elt_0, elt_1, \ldots, elt_{n-1}]$

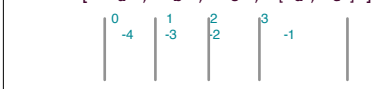
```
>>> [1, 3, 8]
>>> ['the', 'king', 'of', ['spain', 'france']]
>>> []
>>> [1, 2, 'one', 'two']
```

  - `lst[i]` is the i^{th} element of `lst`.
  - Elements are indexed from zero

```
>>> words = ['the', 'king', 'of', 'spain']
>>> words[0]
'the'
>>> words[2]
'of'
```

Indexing Lists

```
>>> determiners = ['a', 'b', 'c', ['d', 'e']]
>>> determiners[0]                  # 0th element
'a'
>>> determiners[-2]                 # N-2th element
'c'
>>> determiners[-1][0]              # sublist access
'd'
>>> determiners[0:2]                # elements in [0, 2)
['a', 'b']
>>> determiners[2:]                 # elements in [2, N)
['c', ['d', 'e']]
```



Operations on Lists

```
>>> determiners = ['the', 'an', 'a']
>>> len(determiners)
3
>>> determiners + ['some', 'one']
['the', 'an', 'a', 'some', 'one']
>>> determiners
['the', 'an', 'a']
>>> determiners.index('a')
2
>>> [1, 1, 2, 1, 3, 4, 3, 6].count(1)
3
```


Operations on Lists 2: List Modification

```
>>> determiners
['the', 'an', 'a']
>>> del determiners[2]              # remove the element at 2
>>> determiners.append('every')     # insert at the end of the list
>>> determiners.insert(1, 'one')    # insert at the given index
>>> determiners
['the', 'one', 'an', 'every']
>>> determiners.sort()              # sort alphabetically
>>> determiners
['an', 'every', 'one', 'the']
>>> determiners.reverse()           # reverse the order
['the', 'one', 'every', 'an']
```


Lists and Strings

- Strings act like lists of characters in many ways.

```
>>> 'I saw a man with a telescope'[-9:]
'telescope'
```

- Converting strings to lists:

```
>>> list('a man')                   # get a list of characters
['a', ' ', 'm', 'a', 'n']
>>> 'a man'.split()                 # get a list of words
['a', 'man']
```

- Converting lists to strings:

```
>>> str(['a', 'man'])               # a representation of the list
"['a', 'man']"
>>> '-'.join(['a', 'man'])          # combine the list into one string
'a-man'
```


Tuples

- Immutable sequence: tuple (cannot be modified)
  - Creation using [optional] parenthesis

```
>>> t = 12345, 54321, 'hello!'
>>> t2 = (12345, 54321, 'hello!')    # equivalent to above
>>> t
(12345, 54321, 'hello!')
```

  - Indexing as with list, strings

```
>>> t[0]
12345
```

  - Modification

```
>>> t[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```


Dictionaries

- A dictionary maps keys to values
  - Like a list, but indexes (keys) can be anything, not just integers
  - Dictionaries are written `{key:val, …}`

```
>>> numbers = {'one':1, 'two':2, 'three':3}
```

  - Dictionaries are indexed with `dict[key]`

```
>>> numbers['three']
3
>>> numbers['four'] = 4
```

  - Dictionaries are *unordered*.

Operations on Dictionaries

```
>>> determiners = {'the':'def', 'an':'indef',
...                'a':'indef'}
>>> determiners.keys()
['an', 'a', 'the']
>>> determiners.has_key('an')
1                                       # 1 is true
>>> del determiners['an']
>>> determiners.has_key('an')
0                                       # 0 is false
>>> determiners.items()
['the':'def', 'a':'indef']
```

Truth Values

- Every expression has a truth value
  - 0 is false, all other numbers are true.
  - "" is false, all other strings are true
  - [] is false, all other lists are true

```
>>> 5 == 3+2                          # == tests for equality
1
>>> 5 != 3*2                          # != tests for inequality
1
>>> 5 > 3*2                           # >, <, >=, <= test for ordering
0
>>> 5 > 3*2 or 5<3*2      # or, and combine truth values
0
>>> 'hello' or 'world'
'hello'
>>> 'hello' and 'world'
'world'
```

Control Flow

- **`if`** statement tests if a condition is true
  - If so, it executes a body
  - Otherwise, it does nothing

```
>>> if len(determiners) > 3:
...         del determiners[3]
...         print 'deleted the 3rd determiner'
```

body

  - Indentation is used to mark the body.
  - Note the ":" at the end of the *if* line.

Control Flow 2

- **`if-else`** statement

```
>>> if len(sentence) > 3:
...             print sentence
>>> else:
...             print 'too short'
```

- **`if-elif`** statement

```
>>> if x<3:
...             print x*3
>>> elif x<6:
...             print x*2
>>> else:
...             print x
```

- **`while`** statement

```
>>> while x<1000:
...         x = x*x+3
```

- **`for`** statement

```
>>> for n in [1, 8, 12]:
...         print n*n+n
```

- **`range()`**

```
>>> for n in range(0, 10):
...         print n*n
```

Control Flow 3

- **`break`** statement
  - quits the current loop (for, while)

```
>>> for x in range(0, 10):
...         print x
...         if x > 5:
...             break
```

- **`continue`** statement
  - skips to the next iteration of the loop
- **`else`** statement following **`for`** or **`while`**
  - else block runs when the loop exits normally (i.e. not via a break)

Working with Files

- To read a file:

```
>>> for line in open('corpus.txt', 'r').readlines()
...     print line
```

- To write to a file:

```
>>> outfile = open('output.txt', 'w')
>>> outfile.write(my_string)
>>> outfile.close()
```

- Example:

```
>>> outfile = open('output.txt', 'w')
>>> for line in open('corpus.txt', 'r').readlines():
...     outfile.write(line.replace('a', 'some'))
>>> outfile.close()
```

Functions

- A function is a reusable part of a program.
- Functions are defined with **`def`**

```
>>> def square(x):
...     return x*x
>>> print square(8)
64
```

- Optional arguments:

```
>>> def power(x, exp=2):              # exp defaults to 2
...     if x <= 0: return 1
...     else: return x*power(x, exp-1)
```

Lambda Functions

- Lambda functions are anonymous functions
- Defined with **`lambda`**

```
>>> square = lambda x: x*x
>>> print square(8)
64
```

- Often used with higher order methods

```
>>> map(lambda x: x*x, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> filter(lambda x: x > 5, [1, 4, 7, 11])
[7, 11]
```

Classes

- A class is a kind of object (like lists or strings) that contains variables and operations (or methods)
- The simplest class:

```
>>> class Simple: pass
```

- Class objects are created with a constructor having the same name as the class:

```
>>> obj = Simple()
```

- Variables are accessed as **`obj.var`**

```
>>> obj.x = 3
```

An Example Class

```
>>> class Account:
...     def __init__(self, initial):
...         self.balance = initial
...     def deposit(self, amt):
...         self.balance = self.balance + amt
...     def withdraw(self,amt):
...         self.balance = self.balance - amt
...     def getbalance(self):
...         return self.balance
```

- **`__init__`** defines the constructor
- **`self`** is the object that is being manipulated
  - It is the first argument to every method.

Using the example class

```
>>> a = Account(1000.00)
>>> a.deposit(550.23)
>>> print a.getbalance()
1550.23
>>> a.deposit(100)
>>> a.withdraw(50)
>>> print a.getbalance()
1600.23
```

Class inheritance

- Can extend existing classes

```
class LimitedAccount(Account):
    def __init__(self, initial, floor = 0):
        Account.__init__(self, initial)
        self.floor = floor

    def withdraw(self, amt):
        if self.balance - amt < self.floor:
            raise Exception(
                'Insufficient funds for withdrawal')
        return Account.withdraw(self, amt)
```

  - overrides Account's withdraw method
  - raises *Exception* when balance is too low

Exceptions

- Exceptions raised using **`assert`** or **`raise`**

```
>>> assert 1 > 2, 'message'
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in -toplevel-
    assert 1 > 2, 'message'
AssertionError: message
```

- Caught using **`except`**

```
>>> try:
...     a = 5/0
... except ZeroDivisionError:
...     print 'Division failed'
...
Division failed
```

Modules and Packages

- Python *modules* "package program code and data for reuse" (Lutz)
  - Cf *library* in C, *package* in Java
- Python *packages* are hierarchical modules (i.e. modules containing other modules)
- Example:

```
>>> import re
>>> re.search('\w+', str)
>>> dir(re)
['DOTALL', ..., 'match', 'search', 'split', ...]
>>> reload(re)

>>> from re import search
>>> search('\w+', str)
```

import vs from..import

Import:

- Keeps module functions separate from user functions
- Requires use of dotted names
- Works with reload

From..import:

- Puts module functions and user functions together
- More convenient names
- Doesn't work with reload

Getting Help

- The pydoc module can be used to get information about objects, functions, etc.

```
>>> from pydoc import help
>>> help(re)
```

- pydoc can also be used from the command line, to provide manpage-like documentaiton for anything in Python:

```
% pydoc re
```

- dir() lists all operations and variables contained in an object (list, string, etc):

```
>>> dir(re)
['DOTALL', 'I', ..., 'split', 'sub', 'subn', 'template']
```

Regular Expressions

- Powerful string manipulation tool (module: re)
  - Search, replace, split
- Regular expression syntax (review):
  - Most characters match themselves
  - [abc] – match one of a, b, or c
  - [^abc] – match any single character except a, b, c
  - "." - match a single character
  - *, +, ?, | - Kleene star, plus, optionality, disjunction
  - (abc)+ - accepts {abc, abcabc, abcabcabc, ...}
  - x{m,n} – match *i* x's, where *i* in [m,n]

Regular Expressions (cont)

- Useful escape sequences:
  - \d digit, \D non-digit
  - \s space, \S non-space
  - \w wordchar (alphanumeric), \W non-wordchar
  - ^ beginning, $ end
  - r'string' (raw string – important for regexps)
- Zero-width assertions
  - \b word-boundary here, \B non word-boundary
  - (?=...) match ... to right of this position
    - e.g. R'Hello (?=World)' matches 'Hello ' only if followed by 'World'
    - Also: (?!...), (?<...), (?<!...)

Other useful modules

- sys
  - command line arguments list `sys.argv`
  - input and output streams `sys.stdin`, `sys.stdout`
- os
  - file operations, process control, pipes and threading
- pickle
  - object persistence
- math
  - basic mathematical operations (trig, log, rounding, etc)
- any many more (see `http://www.python.org/doc/`)

Python references

- http://www.python.org
  - tutorials, library reference
- Books:
  - Introductory: **Learning Python** Mark Lutz, David Ascher
  - Intermediate: **Programming Python (Second Edition)** Mark Lutz
  - Reference: **Python in a Nutshell** Alex Martelli
  - And many more:
    - http://wiki.python.org/moin/PythonBooks