PC204 Lecture 5

Conrad Huang

conrad@cgl.ucsf.edu

Genentech Hall, N453A

476-0415

Topics

- Homework review
- Case study code
- Implementation of Markov analysis
- Writing files
- Writing modules

Homework Review

- 4.1 – Check for duplicates in a list
- 4.2 – Puzzler: longest reducible word

Case Study Code

- The “data structure selection” case study is about trade-offs
  - How fast will this run?
  - How much memory will it use?
  - How hard is it to implement?
- The case study code uses
  - design patterns
    - Reading the contents of a file
    - Sorting items by associated data
  - optional arguments
- histogram.py

Markov Analysis

- Case study also mentions using Markov analysis for generating random (nonsense) sentences using statistics from some text
  - When generating a sentence, if we already generated some words, Markov analysis tells us what words are (somewhat) likely to follow
- markov.py
  - This implementation does not take into account ends of sentences or multi-word phrases or …

Writing Files

- Python provides many ways of writing data to files
- How will your data be used?
  - For reading by humans?
  - For reading by computers?
  - Both?

Human-readable Data Files

- Text files, usually formatted with words
  - Usually, output to files are more strictly structured than output to the screen, so the print statement may not be appropriate since it adds white space and line terminators at its discretion
  - The string formatting operator provides greater control over how strings are constructed

```
col1 = 10
col2  = 20
output = “%d\t%d” % (col1, col2)
```

Format Operator

- “%” is the format operator when operand on the left is a string, aka “format string”
  - format string may have zero or more “format sequences” which are introduced by “%”
    - “%d\t%d” is the format string in example
    - “%d” is the format sequence that specifies what part of the format string should be replaced with an integer
    - Other format sequences include “%s” (string), “%f” (floating point), “%g” (compact floating point), “%x” (base-16 integer), …

Format Operator (cont.)

- The operand to the right of the format operator is either a tuple of values or a single value
  - If there is only one format sequence in the format string, then a single value may be used in place a tuple
  - Each value in the right operand replaces a corresponding format sequence in the format string
  - If the number of values does not match the number of format sequences, or a value type does not match the corresponding format sequence type, an exception will be raised

Format Operator (cont.)

- Here are some examples of format exceptions

```
>>> data = "this is not an integer"
>>> "%d" % data
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: int argument required
>>> "%d %d" % (1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: not all arguments converted during string formatting
```

- Exceptions are not always errors (more on this later)

Opening Files for Writing

- Files may be opened for reading, writing or appending
  - Optional second argument to "open" function specified the "mode", which may be "r"ead, "w"rite, or "a"ppend. "r" is the default.
- Once open for writing or appending, there are two ways to send data to the file:
  - via the "write" method of the file
  - using the "print" statement

Writing to Files

```
f = open( 'output.txt' , 'w' )
f.write("Hello world\n")
print >> f, "Hello world"
f.close()
```

- The "write" method takes a single string argument and sends the data to the file.
  - Note that we need to explicitly include "\n" in the data string because "write" is very simpleminded
- The "print" statement may be used to send data to a file instead of the screen using the ">>" operator
  - Note that we do not need to include "\n" in the print statement because "print" automatically terminates the output line

Closing Files

```
f = open( ‘output.txt’ , ‘w’ )
f.write(“Hello world\n”)
print >> f, “Hello world”
f.close()
```

- The last operation on a file should be “close”, which tells Python that there will be no more output to the file
  - This allows Python to release any data associated with the open file
  - Most operating systems have a limit on the number of open files so it is good practice to clean up when files are no longer in use

with statement

```
with open( ‘output.txt’ , ‘w’ ) as f:
        f.write(“Hello world\n”)
        print >> f, “Hello world”
```

- The **with** statement may be used with file objects (as well as many other types) for automatic disposal of resources
- You still need to handle errors that occur in the “open” call, eg file does not exist
- You do not need to worry about closing the file once it has been opened

Computer-readable Data Files

- Are your data files meant for "internal" use, ie only by your code?
  - There are simple solutions for reading and writing data from and to files if you assume that no one else will try to read your data files (*pickle*, *anydbm*, *shelve*, …)
  - There are more complex solutions when accommodating others (tab-separated values, comma-separated values, extensible markup language [XML], …)

shelve Module

- Sometimes, you want to split a computation process into multiple steps
  - Preprocess once and reuse results multiple times
  - Checkpoint data in long calculations
- The "shelve" module lets you treat a file like a dictionary, except fetching and setting values actually read and write from the file
  - Saving and restoring data looks just like assignment operations
  - Example after next little detour

File Names and Paths

- Python provides functions for manipulating file names and paths
  - Python abstraction for file system is that files are stored in folders (aka directories)
  - Folders may also be stored in folders
  - The name of a file can be uniquely specified by joining a series of folder names followed by the file name itself, eg **/Users/conrad/.cshrc**
    - The joined sequence of names is called the "path"
    - Note that Python accepts "/" as the joining character, even on Windows which actually uses "\"

Files Names and Paths (cont.)

- The “os” and “os.path” modules provide functions for querying and manipulating file names and paths
  - “os.getcwd” – get current working directory
  - “os.listdir” – list names of items in a folder
  - “os.path.exists” – check if file or folder exists
  - “os.path.isdir” – check if name is a folder
  - “os.path.join” – combine names into path

Files Names and Paths (cont.)

- Examples of using file and path functions

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
/var/tmp/conrad
>>> os.listdir(cwd)
['x', 'chimera-build']
>>> cb = os.path.join(cwd, "chimera-build")
>>> print cb
/var/tmp/conrad/chimera-build
>>> os.path.isdir(cb)
True
>>> os.listdir(cb)
['build', 'foreign', 'install']
```

Example using *shelve*

- In the Markov analysis example above, we built the Markov data structures and then used them to generate (nonsensical) sentences
- Suppose we want to generate a sentence once a day from an arbitrary text, but do not want to rebuild the data structures each time (perhaps it takes too long)
- How do we do this?
  - Split Markov analysis and sentence generation into two separate programs
  - Use shelve to store data in analysis, and restore data in generation
  - markov1_prep.py
  - markov1_use.py

Interchangeable Data Files

- Saving and restoring data using *shelve* is very convenient, but not very useful for collaborators unless they also use your code
- Common interchange formats include tabular format with comma- or tab-separated values (CSV/TSV), and Extensible Markup Language (XML)
  - CSV/TSV files are easy to parse and (marginally) human readable
  - XML files are more flexible but (generally) not human friendly

Example using TSV Format

- The Markov example may be rewritten to use TSV instead of shelve
  - markov2_prep.py
  - markov2_use.py
- Note that if we were to change the representations for the Markov data, we would need to modify both source code files to read and write the new representations
- For the *shelve* version, we would not need to modify the saving/restoring part of the code

Debugging

- Reading
  - Read your code critically, just as you do when you edit a paper
- Running
  - Gather information about bugs by adding print statements
  - Do not debug by "random walk"
  - Do not make a dozen changes and hope that one works
- Ruminating
  - We do not do enough of this!
- Retreating
  - The last refuge, but do not be too quick to take it
  - After all, if this version of the code is your best effort, what makes you think the next version will work better?

Homework

- 5.1 – use **os.walk** to count files in a directory tree
- 5.2 – retrieve data from RCSB web service