COM6115: Text Processing

Programming Tips

Mark Hepple

Department of Computer Science
University of Sheffield

Python Tips — *the Good, the Bad, and the Ugly*

- *Elegance* is important:
  - ◇ clear, readable coding helps rapid/effective code development
- Learn to use the clean constructs Python provides

  e.g. use `k in dict` rather than `dict.has_key(k)`
- Know the *default iteration* behaviour of your data structure
  - ◇ so can usually address content via a simple *for*-loop
- Understand the importance of *hash-based* data structures
  - ◇ allow *constant time* look-up / update
  - ◇ usually much more efficient than sequence-based data structures
  - ◇ *beware* of doing sequence-based look-up in hash-based structures

Python Tips — *know the default iteration behaviour*

- Simple *for*-loop provides clean, readable way to address content of an interable data structure:

```
for item in Iterable:
    do_something(item)
```

  - so, useful to know *default iteration behaviour* for *common cases*

- Iterating over X gives items Y . . .
  - a string gives chars in their given (left-to-right) order
  - a list gives its elements, in their given order
  - a tuple gives its elements, in their given order
  - a set gives its elements, in no particular order
  - a dictionary gives its keys, in no particular order
  - a file-stream gives its lines of text, in file order

Python Tips — *hash-based data structures*

- In text processing, often want to handle info about *very many items*
  e.g. counts for 100K words, or *millions* of ngrams
- Hash-based data structures are very suitable for this
  i.e. Python *dictionary* and *set* data structures
- Why? — allow (roughly) *constant time* access to info for a key/item
  i.e. in a *fixed* (small) amount of time *irrespective of how many items stored*
- Using sequential data structs (e.g. list) for similar tasks is a *bad idea*
  - ◇ gives (typically) *linear time* access (i.e. $\propto$ num items stored)
- Test "`item in D`" uses look-up method appropriate to `D`
  - e.g. if it's a *list*, look-up is by *left-to-right sequential comparison*
  - e.g. if it's a *set*, look-up uses *hash-based* method
  - e.g. if it's a *dictionary*, look-up uses *hash-based* method

Python Tips — *hash-based data structures* (ctd)

- Avoid changing hash look-up to sequential one — *common error*
- If `D` is a dictionary, `D.keys()` gives a 'smart iterator' over D's keys
  - ◇ so `x in D.keys()` as efficient as `x in D` (but *less elegant*!)
- BUT all of `list(D)`, `list(D.keys())`, `sorted(D)` return a *list*
  - ◇ so (e.g.) `x in sorted(D)` is *sequential* and *v.inefficient*
- Also v.inefficient is following attempt to check for `x` in `D`:

```
for k in D.keys():
    if k == x:
        ...
```

  - ◇ recreates sequential character of look-up
  - ◇ surprisingly commonly seen!

Python Tips — *avoid piecemeal coding solutions*

- Desire to break task into manageable 'chunks' sometimes leads to *inelegant 'piecemeal' solutions*
  - ◇ avoid this, *unless the task really requires it*
- *Example*: task = count the non-stoplist words in a file
  - ◇ might be tempted to handle as follows (assume stoplist loaded):
    - read the lines of text into a list
    - iterate over list to split each line into a list of tokens
    - iterate again, to delete stop list words
    - iterate again, counting tokens (into a dictionary)

      — this is a poor solution !!
  - ◇ better solution — more efficient, and simpler to code:
    - read the text line by line (i.e. using a for-loop)
    - for each line read, access tokens

      e.g. using .split() string method, or using a regex+findall
    - for each token: if it's a stopword, skip it, otherwise count it