COM6115: Text Processing

Programming for Text Processing: Dictionaries and Sorting

Mark Hepple

Department of Computer Science
University of Sheffield

Python Dictionaries

- *Compound* types store *multiple elements*
- Have seen lists, tuples and strings, as examples of compound types
  - these are all inherently *ordered* – also known as *sequence types*
- Another form of compound type is the Python *dictionary*
  - is inherently *NOT* ordered
  - rather, is a mapping type — maps from KEYS to VALUES
  - can alternatively say they store key:value *pairs*
    - *BUT* any KEY in a dictionary is *unique*, i.e. dictionary can store *at most one* value with any key

Python Dictionaries — *example*

- Example: telephone directory
  - start with an *empty* dictionary "{}"

```
>>> tel = {}
>>> tel['alf'] = 111      # new key, assign value
>>> tel['bob'] = 222      # new key, assign value
>>> tel['cal'] = 333      # new key, assign value
>>> tel
{'alf': 111, 'bob': 222, 'cal': 333}
>>> tel['bob']            # access a value
222
>>> tel['bob'] = 555      # update a value
>>> tel
{'alf': 111, 'bob': 555, 'cal: 333}
>>> 'alf' in tel          # check item is a key
True
>>> 'dan' in tel          # again, check item is a key
False
```

Python Dictionaries — *avoiding key errors*

- Asking value for a *key that is not there*, gives an *error*
  - ◇ big issue for correct use of dictionaries (will crash your code)

```
>>> tel['eric']

Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    tel['eric']
KeyError: 'eric'
>>>
```

- If not sure item is there as a key, must check before accessing value

```
k = 'eric'
if k in tel:
    print(k, ':', tel[k])
else:
    print(k, 'not found!')
```

Python Dictionaries — *iteration*

- Can use a `for loop` to *iterate* over a dictionary
  - ◇ with each cycle, loop var assigned *the next key* of dictionary
  - ◇ but no guarentee as to *order* in which keys returned
    - so your code must not depend order of this in any way

```
>>> tel = {'alf': 111, 'bob': 222, 'cal': 333}
>>>
>>> for k in tel:
        print(k, ':', tel[k])

alf : 111
cal : 333
bob : 222
>>>
```

Sorting Lists

- Often want to *sort* values into some order:
  - e.g. numbers into *ascending / descending order*
  - e.g. strings (such as *words*) into *alphabetic order*
- Python provides for sorting of lists with:
  - ◇ `sorted` general function — *returns* a sorted copy of list
    - (more generally returns sorted list of items *from an iterable*)
  - ◇ `.sort()` called from list — sorts the list "*in place*", e.g.:

```
>>> x = [7,11,3,9,2]
>>> sorted(x)
[2, 3, 7, 9, 11]   # "sorted" returns sorted variant of x
>>> x              # but x itself unchanged
[7, 11, 3, 9, 2]
>>> x.sort()       # ".sort()" modifies list 'in-place'
>>> x              # so x itself now different
[2, 3, 7, 9, 11]
>>>
```

Sorting Lists (ctd)

- By default, sorting puts
  - ◇ numbers into *ascending* order
  - ◇ strings into standard *alphabetic* order (upper *before* lower case)
- Can change default behaviour, using *keyword args*:

  e.g. can *reverse* standard sort order as follows:

```
>>> x = [7,11,3,9,2]
>>> sorted(x)
[2, 3, 7, 9, 11]
>>> sorted(x, reverse=True)
[11, 9, 7, 3, 2]
>>>
```

- Same keyword args used for function and method sorting approaches

  e.g. could use `x.sort(reverse=True)` as *in place* variant above

Sorting Lists — *using* key (keyword argument)

- Keyword key allows you to supply a (single arg) *function*
  - ◇ function computes some *alternate value* from item (of list being sorted)
  - ◇ items of list then sorted on basis of these alternate values
- For 'one-off' functions, can use *lambda notation*

  e.g. `lambda x:(x * x) + 1` :

  means give me one input (`x`) and I'll give you back result $x^2 + 1$

  e.g. `lambda i:i[1]` : given item `i`, computes/returns `i[1]`

  which makes sense if `i` is a *sequence*, so `i[1]` is its 2nd element
- Example: sorting *list of pairs* (tuples) by *second* value
  - ◇ would otherwise sort by first value

```
>>> x = [('a', 3), ('c', 1), ('b', 5)]
>>> sorted(x)
[('a', 3), ('b', 5), ('c', 1)]
>>> sorted(x, key=lambda i:i[1])
[('c', 1), ('a', 3), ('b', 5)]
```

Sorting Lists — *using* cmp (keyword argument)

- A further *keyword arg* `cmp`:
- Lets you supply a *custom* two arg function for comparing list items

  e.g. `sorted(data, cmp=my_order)`
- Function should return negative/0/positive value (usually -1/0/+1)
  - ◇ value depends on whether first arg is considered smaller than/same as/bigger than second
- Sort then puts items into *ascending* order (by default) according to this order
- Allows you to set arbitrarily complex criteria for sorting

  e.g. might sort a list of personnel records, based on
  - i firstly, surname (alphabetically), then (if that's not sufficient info)
  - ii secondly, person's first name, and then
  - iii thirdly, their seniority within company, and then ...

Sorted Handling of Dictionaries

- May use dictionaries to store *numeric data* associated with keys
  - e.g. the counts of different words in a text corpus
  - e.g. density of different metals
  - e.g. share price of companies
- In Text Processing / NLP, we do a lot of counting ...
  - ◇ count frequency of each word appearing in a large corpus of text
  - ◇ for each word *w* in a text, count *which words* appear immediately following *w*, *and how often*
- Often want to address data in a dictionary in an *ordered* fashion
  - ◇ fairly straightforward if order is based on a sort of the keys

    e.g. print telephone directory in *alphabetic order* of names

```
for name in sorted(tel):
    print('NAME:', name, '=', tel[name])
```

Sorted Handling of Dictionaries (ctd)

- May instead want to handle data in a manner ordered w.r.t. the *values*
  - e.g. identify the most common words in text corpus
  - e.g. sort companies by share price, so can identify "top ten" companies
- Can use lambda function returning key's value in dictionary, e.g.

```
>>> counts = {'a': 3, 'c': 1, 'b': 5}
>>> sorted(counts)
['a', 'b', 'c']
>>> sorted(counts, key=lambda v:counts[v])
['c', 'a', 'b']
>>> sorted(counts, key=lambda v:counts[v], reverse=True)
['b', 'a', 'c']
```

Sorting Handling of Dictionaries by Value — *example*

- EXAMPLE: print metals in descending order of density

```
densities = {'iron':7.8, 'gold':19.3, 'zinc':7.13, 'lead':11.4}

for metal in sorted(densities, reverse=True,
                    key=lambda m:densities[m]):
    print('%8s = %5.1f' % (metal, densities[metal]))
```

Prints:

```
    gold =  19.3
    lead =  11.4
    iron =   7.8
    zinc =   7.1
```