COM3110/4115/6115: Text Processing

Programming for Text Processing:

Dictionaries, Sorting and Defining functions

Mark Hepple

Department of Computer Science
University of Sheffield

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
  - ◇ `.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 (ctd)

- 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)]
>>>
```

Dictionaries

- Python dictionary data type:
  - ◇ corresponds to PERL hashes, a.k.a. associative arrays
  - ◇ consist of *unordered sets* of `key:value` pairs
  - ◇ keys must be unique (within given dictionary)
- Example — telephone directory:
  - ◇ here prepopulate with some `name:number` pairs:

```
>>> tel = { 'alf':111, 'bob':222, 'cal':333 }
>>> 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}
>>>
```

Dictionaries (ctd)

```
>>> tel['deb'] = 444    # new key - create new entry
>>> tel
{'alf': 111, 'bob': 555, 'deb': 444, 'cal': 333}
>>> del tel['bob']      # delete entry with given key
>>> tel
{'alf': 111, 'deb': 444, 'cal': 333}
>>> tel.keys()          # get list of keys
dict_keys(['alf', 'deb', 'cal'])
>>> tel.has_key('cal')  # check keys exists
True
>>> 'alf' in tel        # also check for key - nicer
True
>>> for k in tel:       # iterate over keys
...     print(k, tel[k], end='; ')
...
alf 111 ;  deb 444 ;  cal 333 ;
>>>
```

Sorting Dictionaries by Value

- May use dictionaries to store *numeric values* 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
- May want to handle dictionary 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 Dictionaries by Value (ctd)

- EXAMPLE: print metals in descending order of density

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

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

```
    gold =  19.3
    lead =  11.4
    iron =   7.9
    zinc =   7.1
```

- A further *keyword arg* `cmp`:
  - ◇ lets you supply a *custom* two arg function for comparing list items
  - ◇ should return negative/0/positive value depending on whether first arg is considered smaller than/same as/bigger than second

Defining functions

- Use keyword def

  e.g. function to compute Fibonacci series upto n, returned as a list (stops when next value would be >= n)

```
def fib(n):        # compute Fibonacci
    a, b = 0, 1    # series upto n
    series = []
    while b < n:
        series.append(b)
        a, b = b, a + b
    return series
```

- return: can use explicit "return <val>" statement, as above
  - ◇ a return with no argument returns special value "None"
  - ◇ a function call that completes without a return also returns "None"

Defining functions (ctd)

- Function arguments can have *default values*, or be called *by keyword* (i.e. by name):
  - ◇ arguments that have a default value can be *omitted* in function call
  - ◇ keyword args can be given *out of order*
  - ◇ non-keyword args are identified *by position* – must come first in call
- Example: simplified "`range`" function:

```
def myrange(end,start=0,step=1):
    range = []
    if start <= end and step >= 1:
        while start < end:
            range.append(start)
            start = start + step
    return range
```

Defining functions (ctd)

- Example: simplified “range” function:

```
def myrange(end,start=0,step=1):
    ...
```

- Some *okay* function calls:

```
myrange(11)
myrange(11,3,2)
myrange(11,step=2)
myrange(start=3,end=11,step=2)
myrange(step=2,start=3,end=11)
```

- Some *bad* function calls:

```
myrange(start=3,11)        # non-keyword args come 1st
myrange(11,step=2,end=11) # multi values for 'end' arg
myrange(start=3,step=2)    #'end' arg needed:no default
```