COM3110/4115/6115: Text Processing

Programming for Text Processing:

Defining Functions, Dictionaries and Regular Expressions

Mark Stevenson

Department of Computer Science
University of Sheffield

Outline

- Sorting lists
- Dictionaries
  - Sorting Dictionaries
- Defining Functions
- Regular expressions

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:
  - 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’])
>>> ’alf’ in tel        # check for key
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, key=lambda c:counts[c])
[’c’, ’a’, ’b’]
>>> sorted(counts, key=lambda c:counts[c], reverse=True)
[’b’, ’a’, ’c’]
>>>
```

Sorting Dictionaries by Value (ctd)

- EXAMPLE: print metals in ascending 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]))
```

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

- 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 up to 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
```

Regular Expressions

- A regular expression (or **regex**) is a **pattern** describing a set of strings
  - ◇ a *matching* process tests if a given string matches the pattern
  - ◇ may also then *modify* the string

    e.g. by *substituting* a substring, or *splitting* it into substrings

- Regular expressions are a powerful programming tool widely used in text processing.
  - ◇ Found in a wide range of tools (e.g. Perl, Tcl/Tk, Python, Java, `grep`, `sed`, `awk`, `lex`)
  - ◇ *but* note that regex syntax varies

- **Many** applications, such as
  - ◇ Strip the html out of a set of web pages (e.g. to build documentation)
  - ◇ Extract comment blocks from a program (e.g. to build documentation)
  - ◇ Check a document for doubled words ("the the", "here here")

Simple Patterns

- The simplest example of a regex is a **literal pattern**
  - ◇ most chars just match against themself
  - ◇ likewise, most char sequences form a regex to match against identical char sequence in a string
  - ◇ *but* some chars have special behaviour: *metachars*
- For example, string `"pen"`:
  - ◇ will serve as a regex that matches any string that contains the substring `pen`

    e.g. `string "the pen broke"`

    e.g. `"what is epenthesis?"`

Simple Patterns — Example: Python

- Python provides extensive regex facilities
  - not in basic language — must import module ”`re`”
  - can do regex matching using module functions ‘directly’
- Example:

```
import sys, re
with open(sys.argv[1],’r’) as infs:
    for line in infs:
        if re.search(’pen’,line):
            print(line,end=’’)
```

  - `search` scans for *first* substring matching regex *anywhere* in string
  - if finds match, returns a *match object*, else returns `None` (=False)

Simple Patterns — Example: Python (contd)

- When a regex is to be used many times, is better (i.e. faster) to *compile* a regex *object*
- Example:

```
import sys, re
penRE = re.compile(’pen’)
with open(sys.argv[1],’r’) as infs:
    for line in infs:
        if penRE.search(line):
            print(line,end=’’)
```

- Assigning object to a *well-named variable* can also give clearer code

  e.g. having regexes for ‘`word`’, ‘`URL`’, etc

Regex Syntax (1): Alternatives and Groupings

- To specify that one of several **options** are permitted in a match, separate them by a *vertical bar* (or 'pipe') (i.e. |)
  - ◇ *Example*: regex `"car|bike|train"` matches any of:

```
carnation
motorbike
detraining
```

- Can **group** parts of a pattern, using *parentheses*
  - ◇ *Example*: regex `"(e|i)nquir(e|y|ing)"` matches any of:

```
enquiry
inquiring
enquire
```

Regex Syntax (2): Quantifiers

- `*`, `+` and `?` are **quantifiers**
  - ◇ `*` indicates zero or more occurrences
  - ◇ `+` indicates one or more occurrence
  - ◇ `?` indicates zero or one occurrences (i.e. optionality)

- *Example*: regex `"ab*d?e"` matches `abde` and `aeeee` but **not** `bde` or `abd`

Regex Syntax (3): Character Classes

- **[]** indicate a **character class**
  - *Example*: regex `"c[ad]r"` matches `car` and `cdr` but **not** `cadr`

- Can specify char ranges using a hyphen, e.g.

◇ `[A-Z]`	upper case roman alphabet
◇ `[a-z]`	lower case roman alphabet
◇ `[A-Za-z]`	upper and lower case letters
◇ `[0-9]`	digits `0..9`

- Some common char classes have *predefined* names:
  - `.` matches *any* char
  - `\d` abbreviates `[0-9]`
  - `\w` abbreviates `[A-Za-z0-9_]`
  - `\s` abbreviates `[ \f\t\n\r]` (i.e. whitespace)

- To *negate* a char class, put the "carat" sign `^` at the start
  - matches anything *except* chars indicated, e.g. `[^0-9]`

Summary

- Sorting lists
- Dictionaries
  - ◇ Sorting Dictionaries
- Defining Functions
- Regular expressions