COM6115: Text Processing

Python Introductory Materials

Nested Loops
Pylab Arrays and Images
Dictionaries and Sorting

Mark Hepple

Department of Computer Science
University of Sheffield

Nested Loops

- One loop can *contain* another loop:
  - referred to as *nested loops*

```
outer_vals = [1, 2, 3]
inner_vals = ['A', 'B', 'C']

for oval in outer_vals:
    for ival in inner_vals:
        print(oval, ival)
```

← *inner loop*

  - inner loop *runs to completion* for *each iteration* of outer loop

Nested Loops (ctd)

- Inner loop *runs to completion* for *each iteration* of outer loop

```
outer_vals = [1, 2, 3]
inner_vals = ['A', 'B', 'C']

for oval in outer_vals:
    for ival in inner_vals:
        print(oval, ival)
```

  - ◇ above code produces output:

```
1 A
1 B
1 C
2 A
2 B
2 C
3 A
3 B
3 C
```

Nested Loops — example: multiplication table

- Code to print a small *multiplication table*

```
for i in range(1,7):
    for j in range(1,11):
        print(i * j, end=' ')
    print()
```

← *inner loop*

◇ prints:

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
```

◇ inner loop generates a single *row* of the table

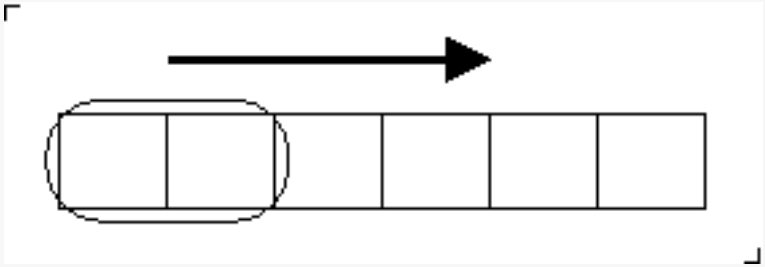
◇ outer loop causes *multiple rows* to be printed

Nested loops — example: sorting

- Nested loops have many uses, e.g: *sorting* values into order

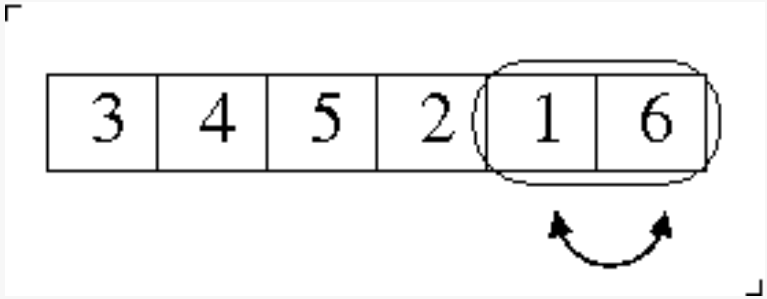
  e.g. list of values: `[4, 3, 6, 5, 2, 1]` — how get into *ascending* order

- Various sorting algorithms
- We'll look at method called Bubble Sort
  - ◇ method moves along list, comparing *adjacent* values
  - ◇ *swaps* adjacent values *if they are out of order*
  - ◇ likened to moving a small window (or 'bubble') along list
    - compare values in window, and swap if needed



Nested loops — example: sorting (ctd)

- *Example*: sorting list: `[4, 3, 6, 5, 2, 1]`
- Bubble *passes* over list::



  - ◇ as bubble moves, highest value seen so far is carried along
  - ◇ at end of pass, list not yet in order
  - ◇ *but* highest value has been moved to final position — *its correct place*
- Pass bubble over for *second time*:
  - ◇ second highest value will be carried along to its correct position
- After N passes (where N = length of list):
  - ◇ all values carried to correct position — list is now sorted

Nested loops — example: sorting (ctd)

- Doing this in Python ...
  - first, making a single pass of the bubble:

```
values = [4, 3, 6, 5, 2, 1]
N = len(values)

for i in range(N-1):
    if values[i] > values[i+1]:
        tmp = values[i]
        values[i] = values[i+1]
        values[i+1] = tmp
```

  - why does `i` here range up to `N-1`, rather than `N`?
    - because, otherwise, accessing value at position `i+1` will cause an `index-out-of-bounds` error

Nested loops — example: sorting (ctd)

- Single pass of the bubble must be *repeated* over, until list is sorted
  - ◇ nest previous 'bubble pass' loop within another, to repeat it `N` times:

```
values = [4, 3, 6, 5, 2, 1]
N = len(values)

for j in range(N):
    for i in range(N-1):
        if values[i] > values[i+1]:
            tmp = values[i]
 Swap       values[i] = values[i+1]
            values[i+1] = tmp
```

Nested loops — example: sorting (ctd)

- Preceding version works, but ...
  - ◇ we can improve it, by avoiding some unnecessary work
  - ◇ need only run outer loop `N-1` times
    - once `N-1` items correctly in place, so also must be the final one
  - ◇ After `j` runs of inner loop, `j` final items correct
    - so bubble can stop its pass earlier — no need to look at these items

```
values = [4, 3, 6, 5, 2, 1]
N = len(values)

for j in range(N-1):
    for i in range(N-1-j):
        if values[i] > values[i+1]:
            tmp = values[i]
            values[i] = values[i+1]
            values[i+1] = tmp
```

Pylab Numeric Arrays

- `pylab` provides a special data type of *numeric arrays*
  - ◇ for efficient storage of numeric data
  - ◇ esp. large matrices
  - ◇ memory efficient, fast matrix operations
- Use `zeros` function to create array of specified size
  - ◇ with values initialized to zero

e.g.

```
>>> from pylab import *
>>> zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
>>>
```

- `arange` function creates array initialised with sequence of values
  - ◇ allows *non-integer* step value (standard `range` function does not)

e.g.

```
>>> arange(0,2,0.3)
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
>>>
```

Pylab Numeric Arrays (ctd)

- Can also have *multi-dimensional* arrays

  e.g. 2D array with dimensions `(3,5)`

```
>>> zeros((3,5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>>
```

- These arrays have a `shape` attribute
  - ◇ reports the dimensions of the array

```
>>> data = zeros((3,5))
>>> data.shape
(3,5)
>>>
```

Pylab Numeric Arrays (ctd)

- Can use nested loops to address the elements of a 2D array

  e.g. 3×5 array:

```
>>> values = zeros((3,5))
>>> values
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

  - run code:

```
val = 0
for row in range(3):
    for col in range(5):
        val = val + 0.01
        values[row][col] = val
```

  - result:

```
>>> values
array([[ 0.01,  0.02,  0.03,  0.04,  0.05],
       [ 0.06,  0.07,  0.08,  0.09,  0.1 ],
       [ 0.11,  0.12,  0.13,  0.14,  0.15]])
```

Pylab Numeric Arrays (ctd)

- More generally:
  - use shape attribute to access dimensions of arrray
  - use these values to specify the nested loops
  - example:

```
val = 0
(d1,d2) = values.shape
for row in range(d1):
    for col in range(d2):
        val = val + 0.01
        values[row][col] = val
```

2D arrays and images

- An image!:

- Images are often represented as 2D arrays
  - ◇ where each array element represents the brightness or colour of a pixel

    e.g. image above is an array with 65 rows and 134 columns:
  - ◇ in this case, each element is a number between 0.0 and 1.0, giving the *intensity* (i.e. brightness) of the pixel on a greyscale
  - ◇ for a colour image, element might itself be a triple, recording separate brightness values for RGB (red, green, blue) components
    - then, *2D array of pixels* stored as *3D array of numeric values*

Python Dictionaries

- We have seen several *compound* types:
  - i.e. lists, tuples and strings
    - ◇ group together *multiple elements*
    - ◇ these are all *sequence types*, i.e. are inherently *ordered*
- Another form of compound type is the Python *dictionary*
  - ◇ is inherently *NOT an ordered type*
  - ◇ instead is a mapping type
  - ◇ serve to *map* KEYS *to* VALUES
  - ◇ alternatively, can say they store key:value *pairs*
  - ◇ *BUT* any KEY in a dictionary is *unique*
    - i.e. a dictionary can store *at most one* value with any key

Python Dictionaries — *example*

- Example — telephone directory:
  - can start with an *empty* dictionary "{ }", and *populate* by assigning values to new keys:

```
>>> tel = {}
>>> tel['alf'] = 111
>>> tel
{'alf': 111}
>>> tel['bobby'] = 222
>>> tel
{'alf': 111, 'bobby': 222}
>>>
```

    - note the *'print format'* for dictionaries
  - here *prepopulate* with some `name:number` pairs:

```
>>> tel = { 'alf':111, 'bobby':222, 'calvin':333 }
>>> tel
{'alf': 111, 'calvin': 333, 'bobby': 222}
>>>
```

Python Dictionaries — *example* (ctd)

- can now look up / update values

```
>>> tel['bobby']          # access a value
222
>>> tel['bobby'] = 555    # update a value
>>> tel
{'alf': 111, 'bobby': 555, 'calvin': 333}
```

- other operations:

```
>>> del tel['bobby']     # delete entry with given key
>>> tel
{'alf': 111, 'calvin': 333}
>>> tel.keys()           # get list of keys (non-standard)
dict_keys(['alf', 'calvin'])
>>> list(tel.keys())     # get (standard) list of keys
['alf', 'calvin']
>>> 'alf' in tel         # also check keys exists
True
>>> 'dave' in tel        # again, check keys exists
False
```

Python Dictionaries — *avoiding key errors*

- If ask value for a *key that is not there*, gives an *error*
  - ◇ main issue for correct use of dictionaries
  - ◇ will crash your code!

e.g.

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

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

  - ◇ if not sure value there, must check before asking for its value

e.g.

```
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

e.g.

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

alf : 111
calvin : 333
bobby : 222
>>>
```

Sorting

- 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 — *modifying sort behaviour*

- 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]
>>>
```

Sorting — *modifying sort behaviour* (ctd)

- Keyword key allows you to supply a *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*
- Lambda notation:
  - ◇ from maths: notation for writing *functions*

    e.g. what is $x^2 + 1$ — a single value, or a function?
    - expression $\lambda x.x^2 + 1$ *unambiguously* denotes function
  - ◇ in Python:

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

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

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

    which only makes sense if *either*:

    — `s` is a *sequence*, so `s[1]` is its 2nd element, or

    — `s` is a *dictionary*, so `s[1]` looks up value for key `1`

Sorting — *modifying sort behaviour* (ctd)

- Example with lambda: sorting *list of pairs* (tuples) by *second* value
  - ◇ by default, sorts by first value

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

  - ◇ here, use key keyword arg, and a lambda expression
    - lambda function looks up second item of each pair
    - sorting done based on these *alternative values*

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

- 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

Sorted Handling of Dictionaries

- Sometimes want to address contents of a dictionary in *sorted order*
  - This is easy for sort based on order of keys
  - Example: print telephone directory in name order

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

alf : 111
calvin : 333
bobby : 222
>>> for k in sorted(tel):
        print(k, ':', tel[k])

alf : 111
bobby : 222
calvin : 333
>>>
```

  - More tricky for sort based on ordering of the values . . .

Sorting Dictionaries by Value

- May use dictionaries to store *numeric values* associated with keys
  - e.g. density of different metals
  - e.g. share price of companies
  - e.g. how often each possible outcome occurred in a series of experiments
- May want to handle dictionary in a manner ordered w.r.t. the values
  - e.g. print metals in descending order of density
  - 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}
>>> labels = list(counts.keys())
>>> labels
['a', 'c', 'b']
>>> labels.sort(key=lambda v:counts[v])
>>> labels    # puts labels into ascending order of count
['c', 'a', 'b']
>>>
```

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}
>>> metals = list(densities.keys())
>>> metals
['zinc', 'gold', 'iron', 'lead']
>>> metals.sort(reverse=True,key=lambda m:densities[m])
>>> metals
['gold', 'lead', 'iron', 'zinc']
>>> for metal in metals:
        print('{0:>8} = {1:5.1f}'.format(metal,densities[metal]))

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