COM6115: Text Processing

Python Introductory Materials

---

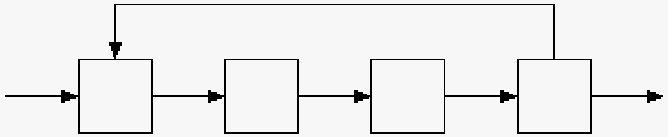
while-Loops and loop-control,
Importing modules
for-Loops, lists,
File input/output

Mark Hepple

Department of Computer Science
University of Sheffield

Loops

- Recall the major control structures: sequence, selection, repetition
  - ◇ Repetition: execute a statement or block of statements more than once



- Programming languages allow for *repetition* control structures by the provision of *looping constructs*
- Two main sorts of loops:
  - ◇ conditional loops ("*while*")
    - loop repeats until a certain condition is met

      e.g. successive approximations to solution to equation
  - ◇ counting loops ("*for*")
    - loop repeats a *preset number of times*

      e.g. changing the brightness of each pixel in an image

The *while* loop

- Programming languages allow for *repetition* control structures by the provision of *looping constructs*
- A key construct in Python (and many languages) is while
  - while loop has associated *condition*
  - repetition continues as long as the condition is satisfied
- Saw informal *pseudo-code* example previously: supermarket shopping

```
1. Get a trolley
2. While there are items on shopping list

   2.1 Read first item on shopping list
   2.2 Get that item from shelf
   2.3 Put item in trolley
   2.4 Cross item off shopping list

3. Pay at checkout
```

  - here, "*there are items on shopping list*" equates to a test, that evaluates as True or False

The *while* loop (ctd)

- In Python, while construct has the form:

```
while CONDITION:
    CODE-BLOCK
```

- *Example*: to print series of values produced when doing repeated *division* (by 2) of some initial value, might do:

```
x = 33
while x >= 1:
    print(x, ': ', end='')
    x = x / 2
print()
```

Output:

```
>>>
33 : 16.5 : 8.25 : 4.125 : 2.0625 : 1.03125 :
>>>
```

The *while* loop: examples

- Function for computing *triangular numbers*
  - *triangular number* of a +ve integer n is sum of values from n down to 1

    i.e. $$n + (n-1) + \cdots + 2 + 1$$

- Definition:

```
def triangular(n):
    trinum = 0
    while n > 0:
        trinum = trinum + n
        n = n - 1
    return trinum
```

e.g.

```
>>> triangular(1)
1
>>> triangular(3)
6
>>> triangular(5)
15
>>>
```

The *while* loop: examples

- Loop to determine if marks are pass/fail/first:

```
mark = 1
while mark > 0:
    mark = input('Enter mark: ')
    mark = int(mark)
    print("Mark is", mark, end='')
    if mark >= 70:
        print(" - first class!")
    elif mark >= 40:
        print(" - that's a pass")
    else:
        print(" - oh dear, that's a fail")
```

```
Enter mark: 77
Mark is 77 - first class!
Enter mark: 44
Mark is 44 - that's a pass
Enter mark: 0
Mark is 0  - oh dear, that's a fail
>>>
```

Loop Control: Early exit and continuation

- Python provides special commands: break and continue
  - *modify* normal *flow* of a loop
- A break statement in a loop:
  - immediately *terminates* the *current iteration*
  - and *ends the loop* overall
- *Example*: prints greeting for name entered, until enter 'done'
  - use of `while True` — loop will run indefinitely

```
while True:
    name = input('Enter name: ')
    if name == 'done':
        break
    print('Hello', name)
```

```
Enter name: Bill
Hello Bill
Enter name: done
>>>
```

Loop Control: Early exit and continuation (ctd)

- A continue statement in a loop:
  - immediately *terminates* the *current iteration*
  - and *starts* the *next* iteration

- Should be used with care:
  - example — might be seen in following code pattern:

```
while CONDITION-1:
    if CONDITION-2:
        STATEMENTS-1
        continue
    STATEMENTS-2
```

  - but same result can often be achieved *without* `continue`

Loop Control: Early exit and continuation (ctd)

- *. . . example continued . . .*
  - ◇ preceding code can be restated *more clearly without* using `continue`:

```
while CONDITION-1:
    if CONDITION-2:
        STATEMENTS-1
    else:
        STATEMENTS-2
```

- `break` and `continue` should be *used with care*:
  - ◇ its use / overuse often symptomatic of bad programming style
  - ◇ same result often better achieved by use of *conditionals*
    - can better express intended logic of task

Importing modules

- Lab sheets have introduced importing as needed for lab
  - but there's more to say . . .
- In Lab, have seen can *import* libraries of existing code
  - known as modules, e.g. saw `pylab`, also `math`, `random`, etc
- `import` statements appear at *top of code file*
  - indicates that imported material is assumed for code that follows
  - *NEVER* put import statements elsewhere in code

    e.g. *NEVER* put import statement inside a function definition
- The `import` command can be used in several different ways

Importing modules — *simple importing*

- The `import` command can be used in several different ways
- Simple import statement

```
import pylab
```

  - imports contents of module `pylab`, *BUT*
  - must *prefix* module name to access its functions/values, e.g.

```
>>> import pylab
>>> pylab.sin(2.2)
0.8084964038195901
>>> pylab.pi
3.141592653589793
```

Importing modules — *importing specific items*

- Alternative form:

```
from pylab import *
```

  - imports *everything* (*) from pylab module
  - *but* now don't prefix module name to use

    e.g. refer to sin, plot, pi functions/value *directly*

- Variant form:

```
from pylab import sin, cos, pi
```

  - imports *only* *named* items from module
  - *but* still no need to prefix module name
  - note that this version is *preferred* over * version

    i.e. is considered better programming style

    - module may contain thousands of definitions
    - this form means you are being *explicit* about what your code requires
  - Spyder even gives error-like messages for * version

Importing modules — *clashing definitions*

- Imports with from, such as:

```
from pylab import *
```

```
from pylab import sin, cos, pi
```

  ◇ allow functions/values to be referred to *directly*

  i.e. don't need to prefix with module name

- PROBLEM: modules may use *same name* to define *different* functions

  e.g. one cos function might just return cosine of an *angle in radians vs.* another that might compute cosine of angle between *two vectors*

  ◇ importing as above (from . . .), cannot tell definitions apart

    - in practice, definition loaded later will *overwrite* one loaded earlier

  ◇ approach where prefix module name to use imported function/value avoids this issue

    - always know which module's function is being used

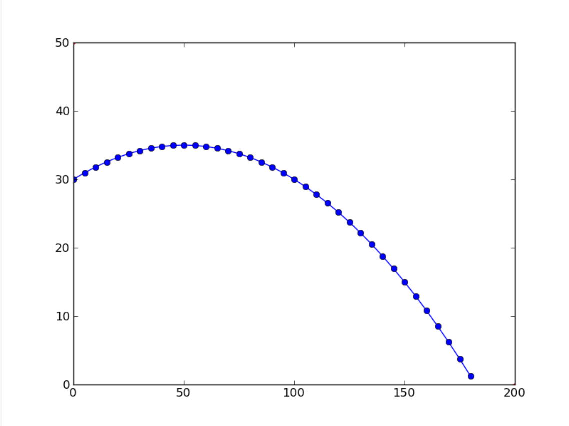
Importing modules — *shorthand module re-naming*

- Import form:
```
import pylab as pl
```
  - allows you provide *shorthand* name for module (some have *long* names)
  - then, use shorthand name as *prefix* to access item
  - thus avoids *name clash* problem, e.g.

```
>>> import pylab as pl
>>> import math as m
>>> pl.pi
3.141592653589793
>>> m.pi
3.141592653589793
>>> pl.pi == m.pi
True
>>>
```

Lists

- Often necessary to deal with data consisting of *collections* of values
  - e.g. might compute *trajectory* of missile as its position (x, y coords) at a series of time points:
    - this gives a *sequence* of values



- In Python, can store such data using a list
  - a *list* is a *single variable* that can hold many values
  - comparable to array data structure of many other languages
    - *BUT* arrays typically restricted to hold values of a *single type*
    - whereas, a single Python list may mix together values of *different types*, e.g. strings and integers

Lists (ctd)

- Lists are inherently *ordered*
  - items can be accessed in terms of their *position* in list
  - positions are identified by their index
  - the *first* item is at *index* 0, the next at 1, . . . , etc

```
>>> x = ['this', 55, 'that']
>>> x[0]
'this'
>>> x[1]
55
>>> x[2]
'that'
>>> x[3]
Traceback (most recent call last):
  File "<pyshell#106>", line 1, in <module>
    x[3]
IndexError: list index out of range
>>>
```

  - accessing a *non-existent position* gives an *error*

Lists (ctd)

- A list can be *changed*
  - individual values can be *overwritten*
  - the list can be *extended*, by *appending*

```
>>> x = ['this', 55, 'that']
>>> x[1] = 'and'
>>> x
['this', 'and', 'that']
>>> x.append('again')
>>> x
['this', 'and', 'that', 'again']
```

- Can use + to compute the *concatenation* of two lists:

```
>>> x = ['the', 'cat', 'sat']
>>> y = ['on', 'the', 'mat']
>>> z = x + y
>>> z
['the', 'cat', 'sat', 'on', 'the', 'mat']
```

Lists (ctd)

- Can also take a *slice* of a list, using *two indices*: `[i:j]`
  - slice starts with item at index `i`, plus items up to (but not including) `j`

```
>>> x = ['this', 'and', 'that', 'once', 'again']
>>>  x[1:4]
['and', 'that', 'once']
```

- Slicing is more 'permissive' than access-by-index
  - as we saw, accessing a *non-existent position* by index gives an *error*
  - *BUT* a *slice* ranging beyond actual positions, just gives what's available

```
>>> z
['the', 'cat', 'sat', 'on', 'the', 'mat']
>>> z[3:10]
['on', 'the', 'mat']
>>> z[8:10]
[]
```

Tuples

- Python has alternative sequence type: tuple
  - written with *round brackets*, e.g. `('this', 55)`
  - like Python list, is *ordered* and allows *access by index*
  - but *cannot* be changed, i.e.:
    - cannot assign new value to a position in an existing tuple
    - cannot append to an existing tuple
  - so why bother?
    - it's more *memory efficient* — but that's not a big concern here

The *for* loop

- The for loop construct widely used to implement *counting loops*
  - in most languages, for loop has an explicit *loop variable*, whose value counts in fixed steps from an initial value to a final value
  - when final value reached, loop stops

    e.g. var `i`, counting 0, 1, . . . , 9
  - loop var may be used within the loop

    e.g. as an *index*, to access successive elements of an *array*
- In Python, use of *for* is slightly different — no explicit *counting*
- Common case: looping is done over a *list*, has form:

```
for VAR in LIST:
    CODE-BLOCK
```

  - with each cycle, *successive items* in `LIST` assigned to `VAR`
  - loop ends when there are *no more items* in `LIST`

The *for* loop *vs.* the *while* loop (intuitively)

- Informal *pseudo-code* example for while-loop: supermarket shopping

```
1. Get a trolley
2. While there are items on shopping list
   2.1 Read first item on shopping list
   2.2 Get that item from shelf
   2.3 Put item in trolley
   2.4 Cross item off shopping list
3. Pay at checkout
```

- Corresponding for-loop *pseudo-code* simpler:

```
1. Get a trolley
2. For (each) item on shopping list
   2.1 Get item from shelf
   2.2 Put item in trolley
3. Pay at checkout
```

The *for* loop — *use for simple iteration*

- For example:

```
>>> values = ['this', 55, 'that']
>>> for item in values:
        print('***', item)
*** this
*** 55
*** that
>>>
```

- Can similarly *iterate* over other types, such as *tuples* and *strings*

```
>>> for char in "Yes":
        print(char)
Y
e
s
>>>
```

- More generally, various types demonstrate *iterable* behaviour, and can appear in a *for* loop

The *for* loop — *use for simple iteration* (ctd)

- This means of accessing items in a list is sufficient for many tasks:

  e.g. scanning list to search for a particular value

  e.g. computing sum of values in list of numbers

  ```
  values = [3, 12, 9]
  total = 0
  for val in values:
      total += val
  print('TOTAL:', total)
  ```

  - ◇ here `total += val` means same as `total = total + val`

- The *loop control* commands break and continue
  - ◇ also work with for-loops, just as with while-loops
  - ◇ break command: immediately *terminates* the *current iteration*, and *ends the loop* overall
  - ◇ continue command: immediately *terminates* the *current iteration*, and *starts* the *next* iteration
    - for a `for` loop, causes loop to move on to next *item* of iteration

The *for* loop — *accessing list positions by index*

- for some purposes, need to address list items *by index*
  - e.g. if want to change value at a particular *position* in list, need to be able to refer to that position in *by index*

```
>>> nums = [10, 44, 17]
>>>  n = nums[1]
>>>  n = n * 2
>>>  nums[1] = n
>>>  nums
[10, 88, 17]
>>>
```

The *for* loop — *accessing list positions by index* (ctd)

- To access positions *by index*, need to use `range` function
  - returns a special 'object', for generating series of integers

```
>>> for i in range(3):
...     print(i)
...
0
1
2
>>>
```

  - `range(3)` – gives values *0, 1, 2*
  - `range(n)` – gives values *0, . . . , (*`n`*-1)*

The *for* loop — *accessing list positions by index* (ctd)

- To get *index* nums for a list, use range together with len function

  e.g. list vals has length 3

```
>>> nums = [8, 12, 10]
>>> len(vals)
3
>>>
```

  - ◇ hence, `range(len(nums))` = `range(3)`
  - ◇ `range(3)` gives values 0, 1, 2
  - ◇ these *values* are precisely the *index positions* of list `nums`
  - ◇ hence, can use them to *access* the list values *by position*, e.g.

```
>>> for i in range(len(nums)):
        print(i, ':', nums[i])
0 : 8
1 : 12
2 : 10
```

The *for* loop — *accessing list positions by index* (ctd)

- By using *index* nums for positions in a list, can both *look-up* value there, and *modify* it

  e.g. in following, increase each of the values in list by adding 100 to it

```
>>> nums = [8, 12, 10]
>>> len(vals)
3
>>> for i in range(len(nums)):
        nums[i] = nums[i] + 100

>>> nums
[108, 112, 110]
>>>
```

Modifying `range` — for special cases of counting loops

- Can modify behaviour of range by adding extra arguments
  - can be useful for 'special cases' of counting loops

- Examples:
  - range with 1 argument: arg gives end value

    e.g. `range(5)` gives: 0, 1, 2, 3, 4
  - range with 2 arguments: args give start and end values

    e.g. `range(2,5)` gives: 2, 3, 4
  - range with 3 arguments: args give start, end and step values

    e.g. `range(3,10,2)` gives: 3, 5, 7, 9

- Can use `list` function to see values generated by range object:

```
>>> list(range(3,9))
[3, 4, 5, 6, 7, 8]
```

File Input/Output

- Command `open(<filename>,<mode>)`:
  - *opens a connections* to named file, for reading or writing
  - creates and returns a *file object*, storing connection info:

```
f = open('foo.txt','r')   # read only
f = open('foo.txt','w')   # write only
f = open('foo.txt','a')   # append only
```

  - default *mode* is `'r'`, i.e. if call just `open('foo.txt')`
  - when mode is `'w'`:
    - if file does not exist, then it is newly created as an empty file
    - if already exists, then is *overwritten*

      — so now have empty file: *be careful!*
  - assign *file object* to a var, so can use it later (any var name okay)

```
infile = open('foo.txt')
```

File Input/Output (ctd)

- Simplest way to *read from* a (text) file = *use a* `for` *loop*:

  e.g.
  ```
  infile = open('foo.txt')
  for line in infile:
      print(line, end='')
  ```

  - with each cycle of loop, next line of text from file assigned to loop var
  - this is a *clean* and *readable* way to read from a file — *recommended!*

- Simplest way to *write to* a (text) file = *use the print command*
  - by default, prints to standard output stream — usually goes *to screen*
  - can use its *optional argument* file to direct output to a file (or stream)

    e.g.
    ```
    f = open('foo.txt','w')
    print('Hello World!', file=f)
    ```

  - print has other *optional args:*
    - end — specifies string added *at end* (defaults to newline)
    - sep — specifies string added *between* expressions (defaults to 1 space)

File Input/Output (ctd)

- Depending on "mode", file objects have various *methods* available:

```
f.readline()   # read line from file
f.read()       # careful: may swallow big file in one!
f.write(s)     # write string s to file
f.close()      # close file
```

  - *BUT* suggest use only if task really requires their use

- Simple example: copy a text file, but adding line numbers:

```
inf = open('shakespeare.txt','r')   # open input file
out = open('numbered.txt','w')      # open output file

for line in inf:              # read input file, line by line
    num += 1
    print(num, ':', line, file=out, end='')

inf.close()                         # close input file stream
out.close()                         # close output file stream
```

- For more complex *formatted* printing: use format method of strings
  - see Sec 8.16 of recommended text

File Input/Output: "with ...as ..." construct

- Filestreams often handled using with ...as ... construct:
  - executes open command and assigns to var
  - filestream automatically closes when code block exits

```
import sys

with open('foo.txt') as infile:
    num = 0
    for line in infile:
        num += 1
        print(num, line, end='')
```

Standard Input/Output Streams

- The standard input, output and error streams are available from the `sys` module as `sys.stdin`, `sys.stdout` and `sys.stderr`
  - must first:
    ```
    import sys
    ```
  - streams have similar *methods* to file objects

    e.g. write string `s` to error stream with:
    ```
    sys.stderr.write(s)
    ```
- Can direct output of `print` statement:
  - to error stream:

    ```
    print('Hello World!', file=sys.stderr)
    ```