

COM6115: Text Processing

Programming for Text Processing:

Programming in Python

Mark Hepple

Department of Computer Science
University of Sheffield

What is Python?

- A free, portable, object-oriented scripting language, combining:
 - ◇ soft. eng. features of traditional systems languages
 - ◇ power and flexibility of scripting languages
- In short:
 - ◇ clean, attractive and compact syntax
 - ◇ supports all major programming styles
 - ◇ runs on all major platforms
 - ◇ free, open source
 - ◇ comprehensive standard library
 - ◇ allows small programs, e.g. 10 lines, where 100+ needed for C++/Java
 - ◇ but, used for large software systems

Basic python code features

- Comment convention: " #" to end of line
- Nesting indicated by **indentation**
- Statements terminated by end of line
- No variable declarations — has **dynamic typing**
 - ◇ type checking done at run-time, rather than at compile-time
 - ◇ less code
 - ◇ eliminates 'redeclaration' errors
 - ◇ useful to allowing rapid prototyping approach to coding
 - ◇ BUT typo on LHS of "=" creates a new variable

Basic python code features (ctd) – indentation as syntax

- Code structure expressed by indentation
 - ◇ produces very readable code
 - ◇ less code clutter (; and { })
 - ◇ eliminates many common syntax errors
 - ◇ promotes and teaches proper code layout
 - ◇ produces compact, clean code
 - ◇ BUT get occasional subtle error from inconsistent spacing
 - makes it important to use an indentation-aware editor

The Python Interpreter

- Python is an *interpreted* language
 - ◇ no separate *compile* step required before run code
- Can run the interpreter in *interactive mode*:
 - ◇ useful for trying out ideas when coding/learning language

```
> python
Python 3.5.2 |Anaconda 4.1.1 (x86_64)| (default, Jul ....
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)]....
Type "help", "copyright", "credits" or "license" for m....
>>> 5 + 3
8
>>> a = 5 + 6
>>> a
11
>>> s = "To be, or not to be!"
>>> s
'To be, or not to be!'
```

Lists

- Lists are a key Python data structure
 - ◇ are *mutable*, i.e. can change both elements of list, and list size

```
>>> x = [ 'what', 'can', 'I', 'put', 'in', 'my', 'list' ]
>>> x[3]           # accessing value at index 3
'put'
>>> x[-2]          # negative position counts in from end
'my'
>>> x[1:3]         # taking a slice
['can', 'I']
>>> x[:3]          # missing value defaults to list start
['what', 'can', 'I']
>>> x[3:]          # missing value defaults to list end
['put', 'in', 'my', 'list']
>>> x[1:3] = [ 'would', 'you', 'have' ] # assign to slice
>>> x
['what', 'would', 'you', 'have', 'put', 'in', 'my', 'list']
```

Lists (ctd)

```
>>> x
['what', 'would', 'you', 'have', 'put', 'in', 'my', 'list']
>>> x[1:6]
['would', 'you', 'have', 'put', 'in']
>>> x[1:6:2]      # slice with step=2
['would', 'have', 'in']
>>> x[6:1:-2]     # slice with negative step
['my', 'put', 'you']
>>> x[::-1]      # does what? - reverses list!
['list', 'my', 'in', 'put', 'have', 'you', 'would', 'what']
>>> x = ['this']
>>> y = ['that']
>>> x.append('and') # add single item to end of list
>>> x
['this', 'and']
>>> z = x + y      # '+' builds concatenated list
>>> z
['this', 'and', 'that']
```

Control structures: if/then/else

```
mark = int(input("Please enter an integer mark: "))  
if mark >= 40:  
    print("Result: pass")
```

```
if mark >= 40:  
    print("Result: pass")  
else:  
    print("Result: fail")
```

```
if mark >= 70:  
    print("Result: first")  
elif mark >= 40:  
    print("Result: pass")  
else:  
    print("Result: fail")
```


Control structures: loops — while

- For indefinite loops use while:

```
while <condition>:  
    <body>
```

e.g.

```
score = 1  
while score > 0:  
    score = int(input("Please enter score: "))  
    print("Score was", score)
```

- continue, break — standard meanings
 - ◇ continue: continue with next iteration of loop
 - ◇ break: exit loop

Control structures: loops — for

- When feasible, prefer use of **for** loop
 - ◇ generally gives more elegant solution
 - ◇ also supports **break**, **continue**
- The **for** loop iterates over a **sequence** (or **any iterable**):

```
for <variable> in <sequence>:  
    <body>
```

- ◇ sequences can be *lists*, but also strings, tuples, etc
- ◇ or other iterables: dictionaries, sets, files, also user-defined classes

e.g.

```
mystring = 'this and that'  
for c in mystring:  
    print(c,end='')
```

prints: t h i s a n d t h a t

Control structures: loops — for (ctd)

- In other languages (e.g. C), common use of for illustrated by:

```
for(i=0; i<10; i++)  
    myarray[i] = myarray[i]+2;
```

- In Python, instead use `range` function to create numeric sequences:

```
for i in range(5):  
    print(i)
```

- ◇ `range(5)` creates and returns an *iterator*
 - in an appropriate context, returns series of values
 - first 0, then 1, ..., then finally 4
- ◇ Can vary behaviour of range by specifying a *start* and *step* values:
 - `range(5)` → 0, 1, 2, 3, 4
 - `range(3,7)` → 3, 4, 5, 6
 - `range(0,10,2)` → 0, 2, 4, 6, 8
 - `range(10,0,-2)` → 10, 8, 6, 4, 2

Control structures: loops — for (ctd)

- Prefer use of *simple for* loop if just need to access elements in turn:

```
scores = [5, 12, 7, 15]
for value in scores:
    if value > 10:
        print(value)
```

- But to *change* list elements, must address them by index:

- ◊ use *range-len* construction

```
scores = [5, 12, 7, 15]
for i in range(len(scores)):
    scores[i] = scores[i] + 2
```

— modifies list, so each value incremented by 2

Basic printing / string formatting

- Basic printing:

```
print(<exp1>, ..., <expn>)
```

- ◇ by default, prints expressions on one line, with a space between (sep), and outputs a final newline (end)
- ◇ can override defaults, with keyword args, e.g.

```
print('this', 'that', sep='\n', end='\n\n')
```

- All Python built-in types have printable representations
- Can create formatted strings by *interpolation*

Basic printing / string formatting (ctd)

- Can create formatted strings by *interpolation*
 - ◇ the *formatting*, or *interpolation*, operator: `'%'`
 - ◇ left-hand arg: a string containing *conversion specs*
 - ◇ right-hand arg: a *tuple* of values for insertion into format string (or single non-tuple value if only one required)
 - ◇ returns result after conversion specs are replaced with values

```
>>> myPi = 3.141592
>>> form = 'The value of %s (to 3 decimal places) is: %.3f'
>>> form % ('PI',myPi)
'The value of PI (to 3 decimal places) is: 3.142'
>>> print('%s = %.3f (3 decimal places)' % ('PI',myPi))
PI = 3.142 (3 decimal places)
>>>
```

File Input/Output

- Call `open(<filename>,<mode>)` creates/returns a file object:

```
f = open('/home/hepple/foo','r') # read only
f = open('/home/hepple/foo','w') # write only
f = open('/home/hepple/foo','a') # append only
```

- Depending on their “mode”, file objects 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
```

- Can read lines from file using `for` loop:

```
f = open('/home/hepple/foo','r')
for line in f:
    print(line,end='')
```

◇ this is an elegant/efficient approach for many text applications

File Input/Output: example

- Copy a text file, but adding line numbers:

- ◇ file names given as *command line args*

e.g. script invoked as:

```
python add_line_nums.py foo.txt foo_copy.txt
```

```
import sys
infile  = open(sys.argv[1], 'r')           # open input file
outfile = open(sys.argv[2], 'w')           # open output file
num=0
for line in infile:                        # read input file stream, line by line
    num = num+1
    print(num, line, end='', file=outfile) # write to out-stream
infile.close()                             # close input stream
outfile.close()                            # close output stream
```


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(sys.argv[1], 'r') 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 (e.g.) error stream:

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

- ◇ to a file (object):

```
f = open('/home/hepple/foo', 'w')  
print('Hello World!', file=f)
```