Filtered List Comprehension 2

```
[ expression for name in list if filter ]
```

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produced.

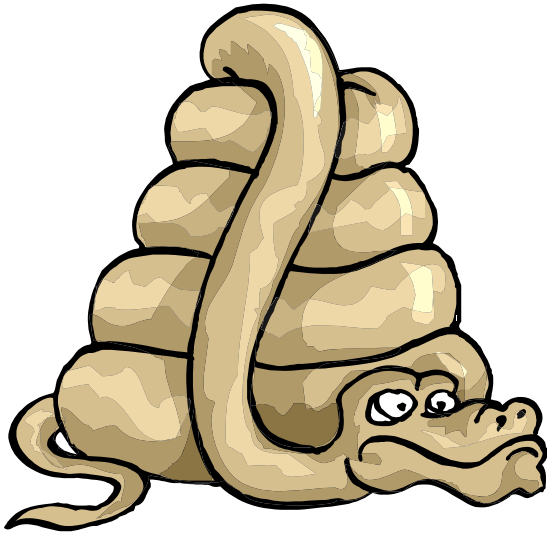
Nested List Comprehensions

- **Since list comprehensions take a list as input and produce a list as output, they are easily nested:**

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
          [item+1 for item in li] ]
[8, 6, 10, 4]
```

- **The inner comprehension produces: [4, 3, 5, 2].**
- **So, the outer one produces: [8, 6, 10, 4].**

String Conversions



String to List to String

- **join turns a list of strings into one string.**

  <separator_string>.join(<some_list>)

```
>>> ";".join( ["abc", "def", "ghi"] )
 "abc;def;ghi"
```

- **split turns one string into a list of strings.**

  <some_string>.split(<separator_string>)

```
>>> "abc;def;ghi".split( ";" )
 ["abc", "def", "ghi"]
```

- **Note the inversion in the syntax**

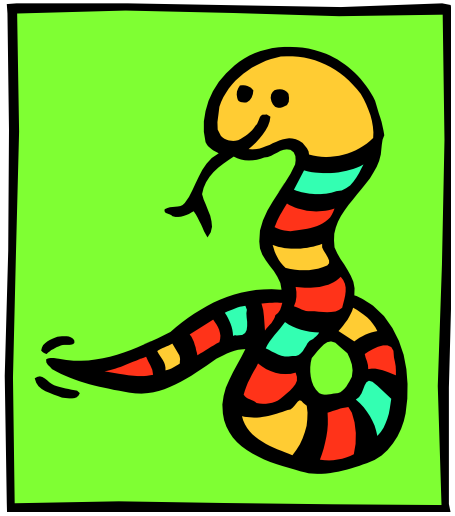
Convert Anything to a String

- **The built-in str() function can convert an instance of any data type into a string.**

  You can define how this function behaves for user-created data types. You can also redefine the behavior of this function for many types.

```
>>> "Hello " + str(2)
"Hello 2"
```

String Operations



String Operations

- A number of methods for the string class perform useful formatting operations:

```
>>> "hello".upper()
'HELLO'
```

- Check the Python documentation for many other handy string operations.

- Helpful hint: use `<string>.strip()` to strip off final newlines from lines read from files

String Formatting Operator: %

- **The operator % allows strings to be built out of many data items in a "fill in the blanks" fashion.**
  - Allows control of how the final string output will appear.
  - For example, we could force a number to display with a specific number of digits after the decimal point.

- **Very similar to the sprintf command of C.**

```
>>> x = "abc"
>>> y = 34
>>> "%s xyz %d" % (x, y)
'abc xyz 34'
```

- **The tuple following the % operator is used to fill in the blanks in the original string marked with %s or %d.**
  - **Check Python documentation for whether to use %s, %d, or some other formatting code inside the string.**

Printing with Python

- You can print a string to the screen using "print".
- Using the % string operator in combination with the print command, we can format our output text.

```
>>> print("%s xyz %d"  %  ("abc", 34) )
abc xyz 34
```

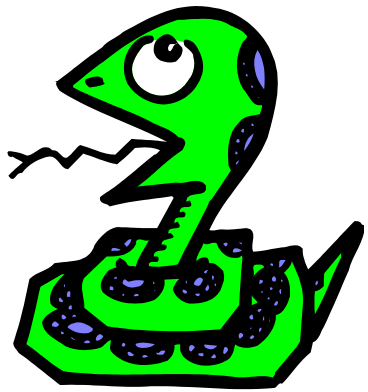
"print" automatically adds a newline to the end of the string. If you include a list of strings, it will concatenate them with a space between them.

```
>>> print("abc")
abc
```

```
>>> print("abc", "def")
abc def
```

- Useful trick: `>>> print("abc"),` doesn't add newline just a single space (i.e., note the trailing comma).

For Loops



For Loops / List Comprehensions

- **Python's list comprehensions and split/join operations provide natural idioms that usually require a for-loop in other programming languages.**
  - As a result, Python code uses many fewer for-loops
  - Nevertheless, it's important to learn about for-loops.

- *Caveat*! **The keywords *for* and *in* are also used in the syntax of list comprehensions, but this is a totally different construction.**

For Loops 1

- A for-loop steps through each of the items in a list, tuple, string, or any other type of object which is "iterable"

```
for <item> in <collection>:
  <statements>
```

- If <collection> is a list or a tuple, then the loop steps through each element of the sequence.

- If <collection> is a string, then the loop steps through each character of the string.

```
for someChar in "Hello World":
   print someChar
```

For Loops 2

```
for <item> in <collection>:
  <statements>
```

- **<item> can be more complex than a single variable name.**
  - When the elements of <collection> are themselves sequences, then <item> can match the structure of the elements.
  - This multiple assignment can make it easier to access the individual parts of each element.

```
for (x, y) in [(a,1), (b,2), (c,3), (d,4)]:
  print x
```

For loops and the *range()* function

- Since a variable often ranges over some sequence of numbers, the *range()* function returns a list of numbers from 0 up to but not including the number we pass to it.

- range(5) returns [0,1,2,3,4]
- So we could say:

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

- (There are more complex forms of *range()* that provide richer functionality…)

"mywc.py": one approach

```
#!/usr/bin/python

import sys

def main():
    num_chars = 0
    num_words = 0
    num_lines = 0

    for line in sys.stdin:
        num_lines = num_lines + 1
        num_chars += len(line)
        line =  line.strip()
        words = line.split()
        num_words += len(words)

    print num_lines, num_words, num_chars


if __name__ == "__main__":
    main()
```

"mywc.py": stdin or filename?

```
#!/usr/bin/python

import fileinput
import sys

def main():
    num_chars = 0
    num_words = 0
    num_lines = 0

    for line in fileinput.input():
        num_lines = num_lines + 1
        num_chars += len(line)
        line = line.strip()
        words = line.split()
        num_words += len(words)

    print num_lines, num_words, num_chars


if __name__ == "__main__":
    main()
```

If filenames are provided to the script, this loop will iterate through all lines in all of the files.

If no filename is provided, the loop will iterate through all lines in stdin.

"mywc.py": a contrived "while" loop

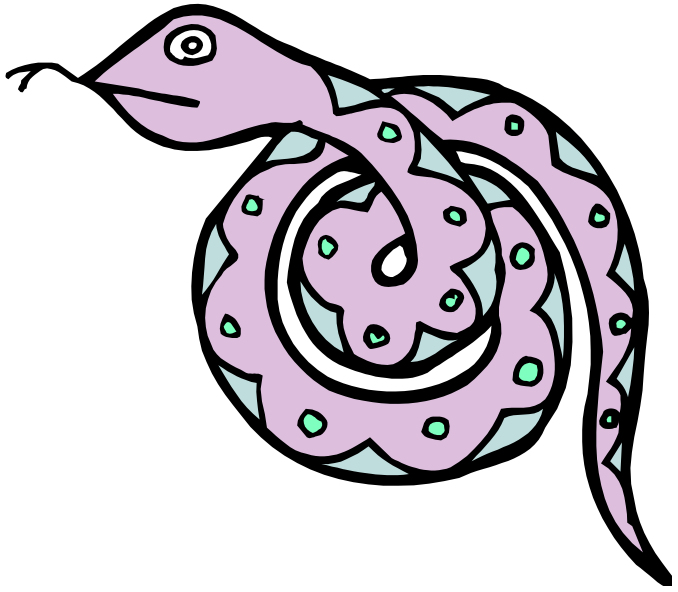
```
#!/usr/bin/python

import sys

def main():
    num_chars = 0
    num_words = 0
    num_lines = 0

    lines = sys.stdin.readlines()

    while (lines):
        a_line = lines[0]
        num_lines = num_lines + 1
        num_chars += len(a_line)
        a_line = a_line.strip()
        words = a_line.split()
        num_words += len(words)
        lines = lines[1:]

    print num_lines, num_words, num_chars


if __name__ == "__main__":
    main()
```

This line using "readlines() could lead to indigestion if the input is very large...

Note the difference between accessing the head of a list...

and accessing the tail of a list...

Some Fancy Function Syntax



Lambda Notation

- Functions can be defined without giving them names.
- This is most useful when passing a short function as an argument to another function.

```
>>> applier(lambda z: z * 4, 7)
  28
```

- The first argument to applier() is an unnamed function that takes one input and returns the input multiplied by four.
- Note: only single-expression functions can be defined using this lambda notation.
- Lambda notation has a rich history in program language research, AI, and the design of the LISP language.

Default Values for Arguments

- You can provide default values for a function's arguments
- These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello"):
            return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

All of the above function calls return 8.

The Order of Arguments

- You can call a function with some or all of its arguments out of order as long as you specify them (these are called keyword arguments). You can also just use keywords for a final subset of the arguments.

```
>>> def myfun(a, b, c):
        return a-b
>>> myfun(2, 1, 43)
  1
>>> myfun(c=43, b=1, a=2)
  1
>>> myfun(2, c=43, b=1)
  1
```

Functions as first-class values

```
#!/usr/bin/python

def function_a():
    print "Inside function_a"
    return function_b

def function_b():
    print "Inside function_b"

def function_c( p ):
    print "Inside function_c"
    p()

def main():
    m = function_a()
    m()
    function_c(m)

if __name__ == "__main__":
    main()
```