Programming for Science
Possible answers for workshop 2

Here are possible answers to the exercises in Workshop 2. Usually these are not the only way of solving the problem, but they are relatively straightforward and clean ways.

1. Iterating over a list.

```
# Iterating over a list

birthday = [ 13, 10, 1958]  # Paddington bear's birthday

prod = 1
sum = 0

for i in birthday:
    print i
    prod = prod*i
    sum = sum + i

print 'Sum of ', birthday, 'is', sum
print 'Product of ', birthday, 'is', prod
```

Note that `i` takes on the value of each item of the list 13, 10, 1958 here each time the loop is executed.

2. Number of words and letters in 'Mary had a little lamb ...'

```
rhyme = ['Mary', 'had', 'a', 'little', 'lamb', 'whose', 'fleece', 'was',
'white', 'as', 'snow']

Nwords = 0
Nletters = 0
for word in rhyme:
    Nwords = Nwords + 1
    Nletters = Nletters + len(word)

print Nwords, 'words and', Nletters, 'letters'
```

As we iterate over the `rhyme` list we just add the number of letters in each word and add 1 to the number of words we've seen. A quicker and neater way of finding the number of words is just: `Nwords = len(rhyme)`.

3. Calculating the average length of the words in `words.txt` is a relatively straightforward extension of the rhyme program.

```
# Find the average length of words in the file words.txt

words = open('words.txt', 'r').readlines()
```

```
Nwords = 0
Nletters = 0
for word in words:
    Nwords = Nwords + 1
    Nletters = Nletters + len(word) - 1 # Subtract 1 to account for newlin

print Nwords, 'words and', Nletters, 'letters'
print 'Average word length', Nletters/float(Nwords) # Float makes an integ
```

4. Here's a program form a new list by appending 'and eggs' to each item in the foods list.

```
foods = ['ham', 'toast', 'spam', 'bacon', 'spinach']

meals = []
for food in foods:
    meals.append(food + ' and eggs')

for meal in meals:
    print meal
```

5. A possible demonstration of adding and scalar multiplication is:

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> b = [7, 8, 9, 10]
>>> a + b
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a*3
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

Note that they are very similar to strings in this respect.

6. Here's a program to reverse a list. Each element of `L` is inserted at the front of the reverse list.

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

reverse = []

for el in L:
    reverse.insert(0, el)

print reverse
```

7. Here's a program that iterates over a list `[32, 33, ..., 84]` generated by `range` and sums the integers.

```
sum = 0
```

```
first = 32
last = 84
for n in range(first, last+1):
    sum = sum + n

print 'Sum of integers from', first, 'to', last, 'is', sum
```

8. This program (typed at the interpreter) generates the cumulative sum of the numbers in `I`. The program iterates over the `I`, calculating the running sum and adding it to the `cumulative` list.

```
>>> I = range(20)
>>> cumulative = []
>>> sum = 0
>>> for n in I:
...     sum = sum + n
...     cumulative.append(sum)
...
>>> cumulative
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153,
```

9. Approximations to the Golden Ratio can be calculated with the following code. As you can see from the direct calculation at the end, the approximation rapidly becomes very good.

`prev` is used to remember the last approximation in the list. Note that the use of `x` and `prev` is unnecessary — a single variable would be sufficient, but this clearer.

```
>>> from math import sqrt  # So we can use the square root function
>>> approx = [prev]
>>> for n in range(19):
...     x = sqrt(prev + 1)
...     approx.append(x)
...     prev = x
...
>>> for g in approx:
...     print g
...
1
1.41421356237
1.55377397403
1.59805318248
1.61184775413
1.61612120651
1.61744279853
1.61785129061
1.61797753093
1.61801654223
1.61802859747
1.61803232275
1.61803347393
```

```
1.61803382966
1.61803393959
1.61803397356
1.61803398406
1.6180339873
1.6180339883
1.61803398861
>>> phi = (1+sqrt(5))/2  # Golden ratio is often denoted by the Greek lett
>>> for g in approx:
...     diff = g - phi
...     error.append(diff)
...     print g, diff
...
1 -0.61803398875
1.41421356237 -0.203820426377
1.55377397403 -0.0642600147199
1.59805318248 -0.0199808062713
1.61184775413 -0.00618623462464
1.61612120651 -0.00191278224178
1.61744279853 -0.000591190222504
1.61785129061 -0.00018269814022
1.61797753093 -5.64578151556e-05
1.61801654223 -1.74465184073e-05
1.61802859747 -5.39127966248e-06
1.61803232275 -1.66599789497e-06
1.61803347393 -5.14821744124e-07
1.61803382966 -1.59088676011e-07
1.61803393959 -4.91611051867e-08
1.61803397356 -1.51916170754e-08
1.61803398406 -4.6944679255e-09
1.6180339873 -1.45067047264e-09
1.6180339883 -4.4828185608e-10
1.61803398861 -1.3852674563e-10
```

10. The Fibonacci series can be generated using a similar paradigm to the Golden Ratio problem, but as each successive Fibonacci number depends on the previous two, we remember them in `prev` and `prevprev`:

```
#  Generate the first few elements of the Fibonacci series from a list of


Fibonacci = [0, 1]        # Initialise with the first two elements of the
prev = 1                  # The element before the element being generated
prevprev = 0              # The one before that

for n in range(18):       # Two elements already generated, so need anothe
    F = prev + prevprev   # New element
    Fibonacci.append(F)
```

```
        # Move on to the next element
        prevprev = prev
        prev = F

for F in Fibonacci:
    print F
```

11. The code produces

```
[1]
[1, 2]
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
....
```

The list `L` is modified *in-place* and so grows from within the loop; there's always another item in the list and the loop never terminates unless interrupted with `ctrl-C`.

12. This exercise gives an example of using *nested* `for` loops to process a list of lists. The outer loop (loop variable `line`) iterates over the 5 lines (each a list itself) of the limerick, while the inner loop (loop variable `word`) iterates over the items of each `line` list, that is the words themselves.

```
limerick = [
    ['There','was', 'a', 'young', 'lady', 'named', 'Wright'],
    ['Whose', 'speed', 'was', 'much', 'faster', 'than', 'light'],
    ['She', 'left', 'home', 'one', 'day'],
    ['In', 'a', 'relative', 'way'],
    ['And', 'returned', 'on', 'the', 'previous', 'night']
]

Nwords = 0
Nletters = 0

for line in limerick:
    for word in line:
        print word,                          # Comma prevents newline
        Nletters = Nletters + len(word)
        Nwords = Nwords + 1
    print                                    # Print a newline

print Nwords, 'words and', Nletters, 'letters'
```