Programming for Science
Possible answers for workshop 6

Here are possible answers to the exercises in the Workshop 6. Usually these are not the only way of solving the problem, but they are relatively straightforward and clean ways.

1. Here's an extended `roll()` and `test_roll()`

```
"""
Functions for rolling dice
"""

from random import random

def roll():
    """
    Return a random number from [1, 2, 3, 4, 5, 6]
    """
    # random() returns a random number between 0 and 1, so multiply the
    # random number by 6 and round down to produce a random number [0, 1,
    # 2, 3, 4, 5] then add 1.
    x = random()
    x = int(x*6) + 1
    return x

def test_roll(N=6000):
    """
    Test roll() by calling it N (default 6000) times and counting the
    number of times each integer is rolled.
    """
    count = [0]*7                        # 0 and 1 to 7
    for i in range(N):
        j = roll()
        # Throw an exception if not
        assert j in [1, 2, 3, 4, 5, 6]
        count[j] = count[j] + 1

    print 'Counts'
    for i in range(7):
        print i, count[i]


def choose(L):
    """
    Return an item at random from the list L
    """
    if len(L) == 0:
        return None
```

```
    x = random()
    i = int(x*len(L))
    assert 0 <= i < len(L)
    return L[i]

def throw(dice=1):
    """
    Throw dice (default 1) dice, returning the sum
    of the faces that show.
    """
    sum = 0
    for n in range(dice):
        sum = sum + roll()
    return sum


print 'In dice.py. __name__ is', __name__

if __name__ == "__main__": # when run as a program
    test_roll()
```

2. Here's an implementation of the Eureka machine straight from the lecture slides:

```
import dice

import drums

for drum in drums.drums:
    print dice.choose(drum),
print
```

3. Here's code to generate business speak.

```
from dice import choose

def readlines(file):
    """
    Read the lines in a file, returning a list where each element
    corresponds to a (whitespaced stripped) line in the file.
    """
    lines = open(file, 'r').readlines()
    stripped = [line.strip() for line in lines]
    return stripped

def make_phrases(filenames):
    """
    Given a list of filenames, return a list of lists where each sublist
    consists of the lines in the given file.
```

```
    """
    linelists = []
    for f in filenames:
        linelists.append(readlines(f))
    return linelists

def business_speak(phraselists):
    """
    Generate and print a random business speak phrase by
    choosing a phrase at random from each of the lists of phrases in
    phraselists.
    """
    # Use choose to construct an entire sentence by selecting a phrase
    # from each wordlist in turn.
    for L in phraselists:
        print choose(L),
    print

if __name__ == "__main__":
    files = ['beginning.txt', 'adjective.txt', 'inflate.txt', 'noun.txt']
    Nphrases = 4                              # Number of phrases to print
    phrases = make_phrases(files)
    for n in range(Nphrases):
        business_speak(phrases)
```

There are several ways of organising this code. Here I've written functions to read the lines from a single file (`readlines`) and return a list of the lines with the newlines stripped. A second function `make_phrases` uses `readlines` to construct a list of lists; each of the sublists is the list of phrases from one of the files. The function `business_speak` is just like the Eureka machine – it randomly chooses a phrase from each of the lists of phrases and prints the constructed phrase. The choosing is done with `choose` imported from the `dice` module. In the main part of the program these functions are used to first read the phrases and then print out as many business speak sentences as required.

The underlying principle is: one function for one job. This leads to a fairly natural decomposition into these functions.

Note that on line 9 I've used a *list comprehension* to construct a list of the stripped lines from the original list of lines in a file:

```
stripped = [ line.strip() for line in lines ]
```

This could have been written in the usual way as:

```
stripped = []
for line in lines:
    strip.append(line.strip())
```

4. The idea here is to keep a *bracket* consisting of the largest number that we know our hidden number is greater than `low` and the smallest number that we know it is less than, `high`. Then

we guess the value in the middle of the bracket `guess` and adjust the bracket to (`low`, `guess`) or (`guess`, `high`) depending on whether the oracle says that the hidden number is bigger or smaller than `guess`. We keep doing this until the bracket is small enough, measured in the code below by the tolerance, `tol`.

```
import oracle

def highlow(low=0.0, high=100.0, tol=0.001):
    """
    Find a the number oracle.oracle is thinking of in the
    range (low, high), by bisection of the interval.
    Returns the best estimate when the width of the bracket
    is less than tol.
    """
    while high - low > tol:
        guess = (high+low)/2.0
        print low, guess, high
        if oracle.oracle(guess):
            # Hidden > guess, so adjust bracket to be
            # (guess, high)
            low = guess
        else:
            # Adjust bracket to be (low, guess)
            high = guess
    return (low+high)/2.0


if __name__ == "__main__": # when run as a script
    x = highlow()
    print 'High-low estimate:', x
    print 'Oracle says:', oracle.reveal()
```

5. Slices to extract parts of the list are:

```
>>> words = ['one', 'fine', 'day', 'in', 'the', 'middle', 'of', 'the', 'ni
>>> words[5:]
['middle', 'of', 'the', 'night']
>>> words[:-5:-1]
['night', 'the', 'of', 'middle']
>>> words[3:6]
['in', 'the', 'middle']
>>> words[::-1]
['night', 'the', 'of', 'middle', 'the', 'in', 'day', 'fine', 'one']
words[::3]
['one', 'in', 'of']
>>> words[2::3]
['day', 'middle', 'night']
>>> words[::-3]
['night', 'middle', 'day']
```

6. Here's a program to find palindromes. Note that the testing for whether a word is a palindrome has been isolated into the function `is_palindrome` that returns a Boolean depending on whether the word is a palindrome. Although this is very simple test (using slices to reverse the string), this sort of organisation makes the way the code operates clear.

```
# Find all the palindromes in words.txt

def is_palindrome(s):
    """
    Return True if `s` is a palindrome
    """
    return s == s[::-1]


lines = open('lowercasewords.txt', 'r').readlines()

for line in lines:
    word = line.strip()
    if len(word) >= 3 and is_palindrome(word):
        print word,
print                                    # Finish the line
```

This code prints all the palindromes on a single long line and here they are:

```
python palindrome.py
aba acca adda affa aga aha ajaja aka ala alala alula ama amma
   ana anana anna apa ara arara atta ava awa bib bob boob bub
   civic dad deed deedeed degged did dod dud eke elle eme ere
   eve ewe eye gag gig gog hah hallah huh ihi imi immi kakkak
   kayak keek kelek lemel level maam madam mem mesem mim minim
   mum murdrum nan non noon nun oho otto pap peep pep pip poop
   pop pup radar redder refer repaper retter rever reviver
   rotator rotor siris sis sooloos tat tebbet teet tenet terret
   tit toot tot tst tut tyt ulu ululu umu utu waw wow yaray yoy
```