# Programming for Science
# Workshop 6

This workshop gives you some more practice with functions, random numbers and organisation of programs.

**1.** Implement the `dice` module described in lectures; that is, the functions `roll`, `test_roll`, `choose` and `throw`. Extend the `roll` function to work for dice with any (positive) number of sides, with a default of 6; extend `test_roll` to suit.

**2.** Use `choose` to implement a Eureka machine as described in lectures. A file containing the definitions of the `drums` lists is on the module resources site. Make sure that you keep `choose` in `dice.py` and `drums` in `drums.py`, and you use `import` to access them.

**3.** The goal of this exercise is to write a "business phrase" generator, which each time it is called produces lines of business speak, such as:

> *It's time that we became uber-efficient with our interactive policy hardware*
>
> *At base level, this just comes down to holistic relative consulting*
>
> *Only geeks stuck in the 90s still go for functional modular capability*
>
> *We need to get on-message about our 'Outside the box' modular matrix approaches*

The idea is very similar to the "Eureka Machine" discussed in lectures: to produce a business speak phrase, your program should randomly select and print a phrase from each of four lists in turn. These lists are stored in files named `beginning.txt`, `adjective.txt`, `inflate.txt` and `noun.txt`, which can be downloaded from the resources website; a business speak phrase is constructed by selecting a phrase from each of these in the order `beginning.txt`, `adjective.txt`, `inflate.txt` and `noun.txt`.

Write a program that will read the required words and phrases from the files and print **five** business speak phrases. The program should be organised so that it can easily be changed to print any number of phrases.

*Hints:*

- Use the `dice.choose` function discussed in lectures.
- As before, you can read all the lines in a file with

  ```
  lines = open('myfile.txt', 'r').readlines()
  ```

  `myfile.txt` must be in the same directory as your program. Remember that each line will have a newline character at the end.
- You can remove the newline and any other whitespace from a string using strip: for example:

  ```
  >>> s = ' string with spaces at the beginning and end \n'
  >>> s
  ' string with spaces at the beginning and end \n'
  >>> stripped = s.strip()
  ```

```
>>> stripped
’string with spaces at the beginning and end’
```

**4.** High-low game. I think of a (floating point) number between 0 and 100. Your job is to guess the number. If you guess a number $x$, I will answer `True` if my number greater than your number, but `False` if my number is less than your guess.

The function `oracle.oracle(guess)` in the file `oracle.py` on the module resources site is a function that answers `True` or `False` according to the scheme above. The module will choose a new hidden number each time it is imported. Don’t look in `oracle.py` until you’ve finished the exercise.

Devise an algorithm for finding out the hidden number: remember we discussed how to do this in lectures in the context of bracketing for root finding. As the hidden number is a floating point number, you won’t be able to find it exactly, so your algorithm should stop when you get “close enough”. Do **not** start coding until you’ve decided on the algorithm.

*Hints:*

- How will you test the function? It might be worth writing your own `oracle` function that returns a known answer before using it for real.
- The function `oracle.reveal()` will tell you the correct answer; but don’t use it until your algorithm has found the answer.
- Remember the `while` loop. It has the following syntax

  ```
  while <condition>:
      body_of_loop
  ```

  where `<condition>` is a Boolean expression and the body of the loop is executed repeatedly while the condition evaluates as `True`. Thus, for example,

  ```
  >>> n = 1
  >>> while n < 1000:
  ...     print n
  ...     n = n*2
  ...
  1
  2
  4
  8
  16
  32
  64
  128
  256
  512
  >>>
  ```

**5.** I didn't discuss `slices` in lectures, but there is plenty of information on them in the lecture slides. Essentially they are a succinct way of specifying a range of indices for a sequence, that is a list, a tuple or a string. Please make sure you've read those slides and then do this and the next exercise.

Given the list of strings:

```
words = ['one', 'fine', 'day', 'in', 'the', 'middle', 'of', 'the', 'night']
```

Write down slice expressions to generate the following lists and test them with the interpreter:

- `['middle', 'of', 'the', 'night']`
- `['night', 'of', 'the', 'middle']`
- `['in', 'the', 'middle']`
- `['night', 'the', 'of', 'middle', 'the', 'in', 'day', 'fine', 'one']`
- `['one', 'in', 'of']`
- `['day', 'middle', 'night']`
- `['night', 'middle', 'day']`

**6.** The object of this exercise is to write a program `palindrome.py` that finds all the palindromes in a file of words. Recall that a palindrome is a word reads the same in either direction: for example, *anna*, *level* or *madam*. The file of words is the dictionary file `lowercasewords.txt` on the ELE page.

Write a program to print all the palindromes in `words.txt` that are at least three characters long. Use a slice to reverse each candidate word!