```python
# Assessed exercises 4
# As before, each question has an associated function, with input arguements
# matching those specified in the question. Your functions will be test for a
# range of different input values, against a model solution, to see if they
# produce the same answers.
import numpy as np
import numpy.random as npr


# At the end of lecture 4 we simulated some 2 dimensional data from a linear
# regression model. In this assignment we're going to try generalise that code
# to higher dimensions.

# The first thing we'll need to to do is simulate the variables xi from a
# uniform distribution

# Q1 Write a function that takes n, a1, a2 and s as inputs, and returns a sample
# of length n, drawn from a uniform distribution U(a1,a2). The seed should be
# set to s.
def exercise1(n, a1, a2, s):
    npr.seed(s)
    return npr.uniform(a1, a2, n)


# A multiple linear regression model is defined as
# y =  b0 + b1*x1 + b2*x2 + b3*x3 + b4*x4 + ... + bpxp + epsilon
# where p is the dimension and {x1,x2,...,xp} are the variables

```

```python
# To fit a linear regression model to a dataset we use the standard equation
# b = (X^T X)^{-1} X^T y, to estimate the coefficients b = [b0, b1, ... , bp].
# Here, y is the dataset (1D array) and X is a matrix where the first column is
# filled entirely with 1s and the subsequent columns are x1, x2, etc.

# Q2 Write a function that takes p and a list S as inputs, and returns the
# matrix X. Use your function from exercise one to create the x1, x2, ... , xp
# variables, with n = 1000, a1 = 0 and a2 =10. The input S = (s0, s1, ... , sp),
# where si corresponds to the seed that should be used to create the variable xi.
# Hint: Python treats all 1D arrays as row vectors. Instead Create the transpose
# of X and return its tranpose ((X^T)^T=X). Also, the function vstack will come
# in useful here.
def exercise2(p, S):
    n = 1000
    a1 = 0
    a2 = 10
    X = exercise1(p, a1, a2, S)
    for i in range(1, n):
        X = np.vstack((X, exercise1(p, a1, a2, S)))
    one_array = np.ones((n, 1))
    X = np.hstack((one_array, X))
    return X


# Q3 Write a function that takes the matirx X and vector y as input, and
# performs a multiple linear regression, using the standard equation
# b = (X^T X)^{-1} X^T y, by calculating the inverse of (X^T X) and multiplying
# the result by (X^T y). The function should return the vector b, which contains
```

```python
# the fits for the intercept and slope parameters (b0, b1, b2, b3, b4)
def exercise3(X, y):
    x_shape = X.shape[0]
    X = np.hstack((np.ones((x_shape, 1)), X))
    return np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)


# Q4 Write another function, with the same inputs and outputs, which uses the
# solve function rather than finding the inverse and then multiplying.
def exercise4(X, y):
    x_shape = X.shape[0]
    X = np.hstack((np.ones((x_shape, 1)), X))
    return np.linalg.solve(X.T.dot(X), np.dot(X.T, y))

# Try testing you functions for different models, e.g.
# y = 3 + 2*x1 - x2 + 0.5*x3 - 0.1*x4 + npr.normal(0,1,n)
# where x1, x2, x3, x4 should be comupted using the function exercise1, with
# different seeds s1, s2, s3, s4. These seeds should be given as a list/array
# into exercise2 to create the matrix X. Running exercise3 and exercise4 should
# give the same result, a vector (1D array) of length p+1, with entries roughly
# equal to the coefficients defined in your multiple linear model,
# e.g. [3,2,-1,0.5,-0.1] for the above example. You can use %timeit to see
# whether exercise3 or exercise4 is quicker for fitting the regression model.

```