```python
'''
Assignment:     hw2, MAE 5930 (Optimization)
File name:      optzn_hw2__JHansen.py
Author:         Jared Hansen
Date created:   09/23/2019
Python version: 3.7.3

DESCRIPTION:
    This Python script is used for answering the following questions from
    MAE 5930 (Optimization) hw2:
        = Problem 1
        = Problem 2
        = Problem 3
        = Problem 5: parts E and F
        = Problem 7: parts B, C, and D
'''








#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------
#--- IMPORT STATEMENTS -------------------------------------------------------
#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------
import numpy as np
import numpy.linalg as npla
import random
# Set a seed to make results reproducible
random.seed(1776)











#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------
#--- PROBLEM 1 ---------------------------------------------------------------
#-----------------------------------------------------------------------------
# Program the backtracking algorithm as described by Boyd on page 464.
#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------

#--- PART A ------------------------------------------------------------------
# Inputs to the function are the objective function, gradient function
# evaluated at x, the point x, and the descent direction v.
#--- PART B ------------------------------------------------------------------
# Within the function set alpha = 0.2 and beta = 0.5.
def backtrack(f, grd, x, v):
    """
    This function implements the backtracking algorithm.

    Parameters
    ----------
    f     : a mathematical function (the objective function)
    grd   : a mathematical function (the gradient of f)
    x     : a point in the domain of f (either a scalar or a vector/list/array)
```

```
    v    : descent direction (a floating point number)
    t    : step size parameter
    alpha : adjustadble floating point number (between 0 and 0.5)
    beta  : float, btwn 0-1, granularity of search (large=fine, small=coarse)

    Returns
    -------
    t : descent direction (floating point number between 0 and 1)
    """
    # Define the starting value of t, and values of alpha and beta
    t = 1.0
    alpha = 0.2
    beta = 0.5
    # Implement the condition found in the algorithm defintion
    while(f(x + t*v) >
          f(x) + alpha*t*np.asscalar(np.dot(grd(x).T, v))):
        t *= beta
        #print("t = ", t)
    return t


# Testing out the backtracking algorithm function using the Rosenbrock function
'''
x_val = np.array([1.0,2.0])
backtrack(rosen_f, rosen_grad, x_val, np.array([-1.0,1.0]))
'''


#---------------------------------------------------------------------------
#---------------------------------------------------------------------------
#--- PROBLEM 2 -------------------------------------------------------------
#---------------------------------------------------------------------------
# Program the gradient descent method as described by Boyd on page 466.
#---------------------------------------------------------------------------
#---------------------------------------------------------------------------
#--- PART A ----------------------------------------------------------------
# Inputs to the function are the objective, gradient, and starting point.
#--- PART B ----------------------------------------------------------------
# Within the function set eta = 1e-6

def grad_desc(f, grd, x0):
    """
    This function implements the gradient descent algorithm.

    Parameters
    ----------
    f   : a mathematical function (the objective function)
    grd : a mathematical function (the gradient of f)
    x0  : a starting pt in the domain of f (either a scalar or an array/list)

    Returns
    -------
    grad_output: a tuple containing (final_x, f_final, iters)
    final_x : the point in the domain of f that minimizes f (to eta tolerance)
    f_final : the function f evaluated at final_x
```

```
    iters   : the number of iterations that it took to achieve the minimum
    """
    # Initialize a variable to count the number of iterations
    iters = 1
    # Declare a variable for the max number of iterations
    MAX_ITERS = 10000
    # Define our tolerance, the constant ETA
    ETA = 1e-6
    # Initialize the point we're going to update, x, as the starting point x0
    x = x0
    while(np.linalg.norm(grd(x)) > ETA):
        # Find step size using backtracking
        t = backtrack(f, grd, x, (-1*grd(x)).T.flatten())
        # Update our "more minimizing" point x
        x = (x - (t * grd(x)).reshape(len(x),))
        #x = x - (t*grd(x))
        # Print statements to see what is going on
        print("f(x)  : ", f(x))
        print("iters : ", iters)
        print()
        # Update the number of iterations
        iters += 1
        # Set up the function to quit after reaching MAX_ITERS
        if(iters > MAX_ITERS):
            print("Reached MAX_ITERS (", MAX_ITERS, ") without converging.")
            break
    # After sufficiently approach ETA or reaching MAX_ITERS return a tuple
    # containing the final_x, f_final, and iters
    grad_output = (x, f(x), iters)
    return(grad_output)
















#---------------------------------------------------------------------------
#---------------------------------------------------------------------------
#--- PROBLEM 3 ----------------------------------------------------------------
#---------------------------------------------------------------------------
# Program Newton's Method as described by Boyd on page 487.
#---------------------------------------------------------------------------
#---------------------------------------------------------------------------
#--- PART A ---------------------------------------------------------------
# Inputs to the function are the objective, gradient, Hessian, and starting pt.
#--- PART B ---------------------------------------------------------------
# Within the function set epsilon = 1e-6

def newtons_method(f, grd, hessian, x0):
    """
    This function implements Newton's method.

    Parameters
```

```
    ----------
    f   : a mathematical function (the objective function)
    grd : a mathematical function (the gradient of f)
    x0  : a starting pt in the domain of f (either a scalar or an array/list)

    Returns
    -------
    newtons_output: a tuple containing (final_x, f_final, iters)
    final_x : the point in the domain of f that minimizes f (to eta tolerance)
    f_final : the function f evaluated at final_x
    iters   : the number of iterations that it took to achieve the minimum
    """
    # Initialize a variable to count the number of iterations
    iters = 1
    # Declare a variable for the max number of iterations
    MAX_ITERS = 20000
    # Define our tolerance, the constant EPSILON
    EPSILON = 1e-6
    # Initialize the point we're going to update, x, as the starting point x0
    x = x0
    # Calculate Newton's decrement value
    lmb_sq = np.asscalar(np.dot(np.dot(grd(x).T , npla.pinv(hessian(x))) ,
                         grd(x)))
    # Stay in this while loop until the have sufficiently small lmb_sq
    while((lmb_sq/2.0) > EPSILON):
        # Compute delta_x and Newton's decrement
        dlt_x = np.dot(-npla.pinv(hessian(x)) , grd(x))
        # Line search for t (descent direction)
        t = backtrack(f, grd, x, (np.array(dlt_x).flatten()))
        # Update our "more minimizing" x
        x = np.array(x + np.dot(t, dlt_x).reshape(len(x), )).flatten()
        # Print statements to see how the algorithm is doing
        print("f(x)  : ", f(x))
        print("iters : ", iters)
        print()
        # Update lmb_sq
        lmb_sq = np.asscalar(np.dot(np.dot(grd(x).T , npla.pinv(hessian(x))) ,
                         grd(x)))
        # Update number of iterations
        iters += 1
        # Set up the function to quit after reaching MAX_ITERS
        if(iters > MAX_ITERS):
            print("Reached MAX_ITERS (", MAX_ITERS, ") without converging.")
            break
    # After sufficiently approach EPSILON or reaching MAX_ITERS return a tuple
    # containing the final_x, f_final, and iters
    newt_output = (x, f(x), iters)
    return(newt_output)


#------------------------------------------------------------------------------
```

```python
#---------------------------------------------------------------------------
#--- PROBLEM 5 -------------------------------------------------------------
#---------------------------------------------------------------------------
#---------------------------------------------------------------------------



# Create a 100x3 matrix A and a 100x1 vector b, each containing random values
# uniformly distributed on [0,1].
#---------------------------------------------------------------------------
A = np.random.uniform(low=0.0, high=1.0, size=(100,3))
b = np.random.uniform(low=0.0, high=1.0, size=(100,1))



#--- PART A ----------------------------------------------------------------
# Solve the problem using your analytical solution from Problem 4.
# Our solution from problem 4 is x = (A.T * A)^(-1) * (A.T) * (b)
analytical_soln = np.dot( (np.dot( np.linalg.pinv(np.dot(A.T, A)), A.T) ), b)
analytical_soln

regr_obj(analytical_soln)



#--- FOR PARTS e, f, AND g -------------------------------------------------
#---------------------------------------------------------------------------
#--- DEFINE FUNCTIONS FOR: OBJECTIVE FCTN, GRADIENT, and HESSIAN -------------
def regr_obj(x):
    """ Takes the point x (R^3 vec) and returns the value of the regression
    (objective) function
    """
    return(np.asscalar((1/2.0) * np.dot((np.dot(A, x).reshape(100,1) - b).T,
                                        (np.dot(A, x).reshape(100,1) - b))))

def regr_grad(x):
    """ Takes the point x (R^3 vec) and returns the gradient of the regression
    (objective) function
    """
    return(np.dot(np.dot(A.T, A), x).reshape(3,1) -np.dot(A.T, b).reshape(3,1))

def regr_hess(x):
    """ Takes the matrix A (dim(A) = 100x3) and returns the Hessian of the
    regression (objective) function
    """
    return(np.dot(A.T, A))


#--- PART E ----------------------------------------------------------------
# Solve the problem using your gradient descent program from Problem 2.
#---------------------------------------------------------------------------
# Let's make an initial guess. I'm going to start with all values of x = 0.4
x0 = np.array([0.4, 0.4, 0.4])
# Find the solution using gradient descent
grad_output = grad_desc(regr_obj, regr_grad, x0)
# Output the results to the console
print("\n----------------------------------------------------------------")
print("--- 5E: Gradient Descent for random A, b, x0=[0.4, 0.4, 0.4] ---")
print("----------------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* grad desc      : ", grad_output[0])
print("f(x*)             : ", grad_output[1])
print("iters to converge : ", grad_output[2])


#--- PART F ----------------------------------------------------------------
# Solve the problem using your Newton's method program from Problem 3.
#---------------------------------------------------------------------------
```

```python
# Let's make an initial guess. I'm going to start with all values of x = 0.4
x0 = np.array([0.4, 0.4, 0.4])
# Find the solution using gradient descent
newt_output = newtons_method(regr_obj, regr_grad, regr_hess, x0)
# Output the results to the console
print("\n-----------------------------------------------------------------")
print("---- 5F: Newton's Method for random A, b, x0=[0.4, 0.4, 0.4] ---")
print("-----------------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* Newton's mtd   : ", newt_output[0])
print("f(x*)             : ", newt_output[1])
print("iters to converge : ", newt_output[2])


#--- PART G ----------------------------------------------------------------
# Try different initial guesses and document your observations/thoughts.
#--------------------------------------------------------------------------

# Let's try a starting guess that is far from the correct answer (x0_far)
x0_far = np.array([300.0, -400.0, 120.0])
# With gradient descent
grad_output_FAR = grad_desc(regr_obj, regr_grad, x0_far)
print("\n-------------------------------------------------------------")
print("--- 5G: Gradient Descent (x0_far=[300.0, -400.0, 120.0]) ---")
print("-------------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* grad desc      : ", grad_output_FAR[0])
print("f(x*)             : ", grad_output_FAR[1])
print("iters to converge : ", grad_output_FAR[2])
# With Newton's method
newt_output_FAR = newtons_method(regr_obj, regr_grad, regr_hess, x0_far)
print("\n--------------------------------------------------------------")
print("---- 5G: Newton's Method (x0_far=[300.0, -400.0, 120.0]) ----")
print("--------------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* Newton's mtd   : ", newt_output_FAR[0])
print("f(x*)             : ", newt_output_FAR[1])
print("iters to converge : ", newt_output_FAR[2])

# Let's try a starting guess that is close to the correct answer (x0_close)
x0_close = (analytical_soln -
          np.array([0.01, 0.01, 0.01]).reshape(3,1)).flatten()
# With gradient descent
grad_output_CLOSE = grad_desc(regr_obj, regr_grad, x0_close)
print("\n-----------------------------------------------------------------------")
print("--- 5G: Gradient Descent (x0_close=anltc_sln-[0.01, 0.01, 0.01]) ---")
print("-----------------------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* grad desc      : ", grad_output_CLOSE[0])
print("f(x*)             : ", grad_output_CLOSE[1])
print("iters to converge : ", grad_output_CLOSE[2])
# With Newton's method
newt_output_CLOSE = newtons_method(regr_obj, regr_grad, regr_hess, x0_close)
print("\n-----------------------------------------------------------------------")
print("--- 5G: Newton's Method (x0_close=anltc_sln-[0.01, 0.01, 0.01]) ----")
print("-----------------------------------------------------------------------")
print("x* analytical     : ", analytical_soln.flatten())
print("x* Newton's mtd   : ", newt_output_CLOSE[0])
print("f(x*)             : ", newt_output_CLOSE[1])
print("iters to converge : ", newt_output_CLOSE[2])
```

```python
#---------------------------------------------------------------------------
#---------------------------------------------------------------------------
#--- PROBLEM 7 -------------------------------------------------------------
#---------------------------------------------------------------------------
# For the optimization problem described in Problem 6:
#---------------------------------------------------------------------------


#---------------------------------------------------------------------------
#--- DEFINING ROSENBROCK FUNCTIONS (OBJECTIVE, GRADIENT, HESSIAN) FOR
#--- MINIMIZING WITH GRADIENT DESCENT AND NEWTON'S METHOD
#---------------------------------------------------------------------------


#--- ROSENBROCK FUNCTION ---------------------------------------------------
def rosen_f(x):
    """ Takes the point (R^2 vector) x and returns the Rosenbrock function at x
    """
    return((1 - x[0])**2+ 100*((x[1]-x[0]**2)**2))

#--- ROSENBROCK FUNCTION'S GRADIENT ----------------------------------------
def rosen_grad(x):
    """ Takes the point x and returns the gradient of the Rosenbrock fctn at x
    """
    # The partial derivative of f w.r.t. x[0]
    df1 = -2*(1 - x[0]) - (400*x[0])*(x[1] - (x[0]**2))
    # The partial derivative of f w.r.t. x[1]
    df2 = 200*(x[1] - (x[0]**2))
    # Returns the gradient as a NumPy array
    return(np.array([df1, df2]))

#--- ROSENBROCK FUNCTION'S HESSIAN -----------------------------------------
def rosen_hess(x):
    """ Takes the point x and returns the Hessian of the Rosenbrock fctn at x
    """
    x0 = np.asscalar(x[0])
    x1 = np.asscalar(x[1])
    # The second partial derivative of f w.r.t. x[0]
    d2f_dx2 = 2 - 400*(x1) + 1200 * (x0**2)
    # The off-diagonal entry in the Hessian
    d2f_dydx = -400*x0
    # The second partial derivative of f w.r.t. x[1]
    d2f_dy2 = 200
    # Arrange these partial derivatives in a matrix, return that matrix
    hess_matrix = np.matrix([[d2f_dx2, d2f_dydx], [d2f_dydx, d2f_dy2]])
    return(hess_matrix)


#--- PART B ----------------------------------------------------------------
# Solve the problem using your gradient descent program from Problem 2.
#---------------------------------------------------------------------------
# For x0=[1.3,1.3] grad desc converges to local min [1,1] in 1628 iterations
gd_rosen = grad_desc(rosen_f, rosen_grad, np.array([1.3, 1.3]))
print("\n-------------------------------------------------------------")
print("---- Gradient Descent to minimize Rosenbrock (x0=[1.3,1.3]) ----")
print("--------------------------------------------------------------")
print("x* true          : ", np.array([1.0,1.0]))
```

```python
print("x* grad desc      : ", gd_rosen[0])
print("f(x*)             : ", gd_rosen[1])
print("iters to converge : ", gd_rosen[2])


#--- PART C ---------------------------------------------------------------------
# Solve the problem using your Newton's method program from Problem 2.
#---------------------------------------------------------------------------------
# For x0=[1.3,1.3] Newton's method converges to local min [1,1] in 8 iterations
nm_rosen = newtons_method(rosen_f, rosen_grad, rosen_hess,
                          np.array([1.3, 1.3]))
print("\n----------------------------------------------------------------")
print("---- Newton's Method to minimize Rosenbrock (x0=[1.3,1.3]) ----")
print("----------------------------------------------------------------")
print("x* true           : ", np.array([1.0,1.0]))
print("x* Newton's mtd   : ", nm_rosen[0])
print("f(x*)             : ", nm_rosen[1])
print("iters to converge : ", nm_rosen[2])


#--- PART D ---------------------------------------------------------------------
# Try initial guesses [1.0, -1.0] and [100.0, -1.0]. Document observations
#---------------------------------------------------------------------------------
# Gradient descent converges to local min [1,1] in 2599 iterations
gd_1_neg1 = grad_desc(rosen_f, rosen_grad, np.array([1.0, -1.0]))
print("\n------------------------------------------------")
print("---- Results of Gradient Descent (x0=[1,-1]) ----")
print("------------------------------------------------")
print("x* true           : ", np.array([1.0,1.0]))
print("x* grad desc      : ", gd_1_neg1[0])
print("f(x*)             : ", gd_1_neg1[1])
print("iters to converge : ", gd_1_neg1[2])
# Newton's method converges to local min [1,1] in 2 iterations
nm_1_neg1 = newtons_method(rosen_f, rosen_grad, rosen_hess,
                           np.array([1.0, -1.0]))
print("\n------------------------------------------------")
print("---- Results of Newton's Method (x0=[1,-1]) ----")
print("------------------------------------------------")
print("x* analytical     : ", np.array([1.0,1.0]))
print("x* Newton's mtd   : ", nm_1_neg1[0])
print("f(x*)             : ", nm_1_neg1[1])
print("iters to converge : ", nm_1_neg1[2])
# Gradient descent converges to local min [1,1] in 3125 iterations
gd_100_neg1 = grad_desc(rosen_f, rosen_grad, np.array([100.0, -1.0]))
print("\n--------------------------------------------------")
print("---- Results of Gradient Descent (x0=[100,-1]) ----")
print("--------------------------------------------------")
print("x* true           : ", np.array([1.0,1.0]))
print("x* grad desc      : ", gd_100_neg1[0])
print("f(x*)             : ", gd_100_neg1[1])
print("iters to converge : ", gd_100_neg1[2])
# Newton's method converges to local min [1,1] in 210 iterations
nm_100_neg1 = newtons_method(rosen_f, rosen_grad, rosen_hess,
                             np.array([100.0, -1.0]))
print("\n--------------------------------------------------")
print("---- Results of Newton's Method (x0=[100,-1]) ----")
print("--------------------------------------------------")
print("x* analytical     : ", np.array([1.0,1.0]))
print("x* Newton's mtd   : ", nm_100_neg1[0])
print("f(x*)             : ", nm_100_neg1[1])
print("iters to converge : ", nm_100_neg1[2])
```