```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Oct  5 12:51:52 2020

@author: jimmy
"""
# Import relevant modules/packages
import numpy as np
from astropy import units as u
from astropy import constants as const
import matplotlib.pyplot as plt

# Function to make a list of a descending geometric series
def DescendingGeometric(length):

    # Make list of coefficients that are all 1
    c = np.ones(length)

    # Multiply each component by another factor of 1/2
    for i in range(1,len(c)):
        c[i] *= .5/i

    return(c)

# Function to numerically solve differentiable equation
# Resource that helped me: https://www.math.ubc.ca/~pwalls/math-python/roots-optimization/newtor
def NewtonRaphson(f,df,x0,precision,numSteps):
    # Inputs:
    #    f: function to evaluate
    #    df: derivative of function
    #    x0: initial guess at solution
    #    precision: answer won't exactly be 0, so set a tolerance
    #    numSteps: maximum number of times to iterate

    # Establish first guess at solution
    x = x0

    # Iterate over number of steps
    for i in range(0,numSteps):

        # Evaluate function
        func = f(x)

        # If f(x) is within precision, declare that value of x as the solution
        if abs(func) <= precision:
            #print('A solution of {0:.2e} was found in {1} iterations'.format(x,i))
            break

        # If f(x) is not within precision, continue searching for solution
        elif abs(func) > precision:

            # Evaluate derivative
            deriv = df(x)

            # Adjust guess of solution by subtracting quotient of function and derivative from t
            x -= func/deriv

    return(x)

# Function to compute Chi Squared and reduced Chi Squared to compare models to obserations
def ChiSquared(model,observation,error,free):
    # Inputs:
    #     model = list of values from model
    #     observation = list of values from actual observations
    #     error = list of errors (sigma) for each observation
    #     free = number of free parameters in the model
```

```
    # Returns:
    #     Chi Squared and reduced Chi squared to indicate goodness of fit for the model

    # Initialize Chi Squared as 0
    ChiSq = 0.0

    # Calculate number of degrees of freedom (# of data points - free)
    nu = len(model) - free

    # For each data point:
    for i in range(len(model)):
        # Calculate the difference between the obsrevation and model (residual)
        residual = observation[i] - model[i]

        # Calculate square of quotient of residual and error value for particular data point
        term = (residual/error[i])**2

        # Add this term to the overall Chi Squared value
        ChiSq += term

    # Calculate reduced Chi Squared (just Chi Squared / # of DoF)
    RedChiSq = ChiSq/nu

    return(ChiSq,RedChiSq,nu)

# Function to calculate Gaussian
def Gaussian(x,offset,amplitude,mean,stddev,wavelength=5000.0):
    # Inputs:
    #   x: point at which to calculate Gaussian (can be a list of values)
    #   offset: set continuum level of Gaussian
    #   amplitude: peak depth of function
    #   mean: center of Gaussian
    #   stddev: width of Gaussian
    #   wavelength: reference wavelength for spectrum

    # Returns:
    #   Value of Gaussian function at x

    # Define exponent
    exponent = (-1.0*(x-mean-wavelength)**2)/(2*stddev)

    # Calculate function value
    function = offset-(amplitude*np.exp(exponent))

    return(function)

# Function to calculate non-relativistic doppler shift
def NonRelDoppler(new_value,rest=5000.0):

    # Convert speed of light to km/s
    c = const.c.to(u.km/u.s).value

    # Calculate new velocity
    velocity = ((new_value/rest)-1)*c

    return(velocity)

# Trapezoidal integration function
def TrapIntegrate(f,a,b,N=500):
    # Inputs:
    #   a: lower bound
    #   b: upper bound
    #   N: # of trapezoids
    # Returns:
    #   s*h: value of integral
    # Example:
    # f = lambda x: 3*x**2
```

```
    # >> TrapIntegrate(f,0,1,10000)
    # >>>> Integral = 1.0000000050000002

    # Step size
    h = (b-a)/float(N)

    # First and last terms of trapezoidal sum calculated directly
    s = 0.5*(f(a)+f(b))

    # Integrate over number of trapezoids
    for theta in range(0,N):

        # Evaluate function at one step further from previous step
        s += f(a+(theta*h))
    print('Integral = {0}'.format(s*h))
    return(s*h)

# Function to calculate radiative equilibrium tempertature
def EquilTemp(dist,albedo=0.3,Rstar=1.0*const.R_sun,Teff=5780*u.K,show=False):
    # Inputs:
    #   albedo: albedo of body of interest (usually planet; default=0.3)
    #   Rstar: radius of host star (in Rsun; default is 1)
    #   dist: distance of body of interest from host star (usually SMA in au)
    #   Teff: effective temperature of host star (in K; default is Sun temp.)
    # Returns:
    #   Teq: radiative equilibrium temperature of body of interest

    # Convert Rstar and dist to same units (m)
    Rstar = Rstar.to(u.m).value
    dist = dist.to(u.m).value

    # Calculate radiative equilibrium temperature
    Teq = (((1-albedo)*Rstar**2)/(4*dist**2))**(1/4)*Teff

    # Print temperature if desired
    if show == True:
        print("Body Temperature = {0:.2f} K".format(Teq))

    return(Teq)

# Function to calculate apparent magnitude difference
def MagDiff(flux1,flux2):
    # Inputs:
    #   flux1,2: integrated fluxes over different freq/wavelen. baselines
    # Returns:
    #   delta_m: apparent magnitude difference (m1-m2)

    # Calculate delta_m
    delta_m = -2.5*np.log10(flux1/flux2)

    print("Apparent magnitude difference = {0:.3f}".format(delta_m))
    return(delta_m)

# Function to calculate main part Planck function at given wavelength
def Planck(x,T,units=True):
    # Inputs:
    #   x: value of x-variable to calculate Planck function at
    #   T: temperature of blackbody (in K)
    #   units: boolean to decide if quantities should have units
    #           (no units is preferable if using func. to integrate)
    # Returns:
    #   B: value of Planck function at that wavelength

    # Define temperature with Kelvin units
    #T = T*u.K

    # Calculate 2h/c^2 (prefactor in Planck's function)
```

```
prefactor = 2*const.h/const.c**2

# Calculate h/kT (constant in exponential of Planck's function)
exp_factor = const.h/(const.k_B*T)

if units == False:
    # Calculate value of Planck function at this wavelength
    B = prefactor.value*x.value**3/(np.exp(exp_factor.value*x.value)-1)
else:
    B = prefactor*x**3/(np.exp(exp_factor*x)-1)

return(B)
```