Problem: A cannonball is shot into the air. We want its trajectory.

Steps: [cb = cannonball]

Think of cb being shot from (0,0) on x-y plane into the 1st quadrant (x>0, y>0)

Think of x-axis as ground level

x refers to x-axis direction (distance)
y refers to y-axis direction (height of cb above ground)

1. Inputs: angle, velocity, height, time-interval (of position updates)
2. Compute/get (xp, yp), initial position of cb
3. Compute/get initial velocity (xv, yv) of cb

While (cb is in the air)
    Update (xp, yp) for time-interval
    Update (xv, yv) for time-interval
    Output distance traveled (xp)
    Output height above ground (yp)

# How to shoot a cannonball (for homework you can draw the ball in a graphics window;
# display it as soon as you compute its new position at the end of each small time-interval)

#cb.py

from math import sin, cos, radians

def main():
    angle = eval(input("Enter launch angle (degrees): "))
    vel = eval(input("Enter initial velocity (meters/sec): "))
    h0 = eval(input("Enter initial height (meters): "))
\[ t = \text{eval(input("Enter small time-interval between position updates of cb: "))} \]

# sin and cos etc need parameters in radians and not degrees, so convert
degrees to radians
# 360 degrees = 2*pi radians, so 1 degree = pi/180 radians
# so theta degrees = \((\text{theta})\) * pi/180 radians

# It's good to know this, but we can call a function radians() to do
this conversion for us

\[ \text{theta} = \text{radians}(\text{angle}) \]

# get the starting setup of cb

\[ \text{xp} = 0 \]
\[ \text{yp} = 0 \]

#so we are launchig from ground, at (0,0)
of x-y axis

\[ \text{xv} = \text{vel} * \cos(\text{theta}) \] #from actual velocity at angle theta, get x-
direction velocity
\[ \text{yv} = \text{vel} * \sin(\text{theta}) \] #similarly get y-direction velocity

# loop while cb travels upwards and finally hits ground

\[ \text{while (yp >= 0)}: \] # as long as it is up in the air

# keep updating where it is at the end of every "t" units of time (t
is small time-interval)

\[ \text{xp} = \text{xp} + t * \text{xv} \] # it's new posn after time t

# gravity is acting on the ball in the y-direction. It'ss velocity
in the y-direction gets
# cut down (or reduced) at the rate of 9.8 meters/sec (i.e., 9.8
meters in 1 second)

# so in a small time t seconds, its velocity in y-direction is
reduced by t*9.8
# (we simply replaced 1 second by the smaller t seconds)

\[ \text{yv} = \text{yv} - t \times 9.8 \] # now we have the y-direction
velocity after t secs

# Now we have two velocities in the y-direction: the velocity of cb
at the start of the time
# interval (yv) and the velocity of cb at the end of the time
interval (yv_new)

# which velocity should we use to calculate how far it has gone in
the y-direction?

    # Idea: acceleration due to gravity is a constant, we can simply use the
    # average velocity of cb in this particular time-interval t. How to
get that?

    # average velocity = (velocity at start of time interval + velocity
at end of time interval)/2

    # so now we can get the y-position using average velocity

    yp = yp + t * (yv + yv_new)/2  # it looks like the xp update
above

    yv = yv_new  # since cb is at the new location at end of time
t, make sure it has the
    # new veocity at that point

    print("\nCB has traveled {0:0.2f} meters and is now at height
{1:0.2f} meters.".format(xp, yp))

#___________________________________________________________________
#2.py

'''
Look at how clean-looking your program is when you make it more modular. Put
all the
separate work in separate functions (i.e., separation of concerns).

When you need to change something, you likely only need to touch particular
functions
'''

def main():

    angle, vel, h0, t = get_from_user()

    xp, yp = 0, h0

    xv, yv = get_initial_directional_velocities(vel, angle)

    while (yp >= 0):

        xp, yp, yv = Update_CB_position(t, xp, yp, xv, yv)

        # Here you can print the new position of CB
# We can take portions of code from 1.py and easily put them in these functions and then run
# main()

# You already know how to do this and write modular code

'''
Now the next thing we will learn is how to use CLASSES and thus eventually use
OBJECT-ORIENTED PROGRAMMING

The code will look like the main() function above, but will use objects. An object (say, a cannonball) is an instantiation of a class. Which class? Say, the class of all Projectiles which behave the same way on earth due to gravity.

So the new main() might look like this:

```python
def main():
    angle, vel, h0, t = get_from_user()
    # now we will initiate a cannonball from the class of all projectiles
    cb = Projectile(angle, vel, h0)
    # In the above line, Projectile() is called a constructor for the Projectile class.
    # It makes a projectile, gives it some initial values for those parameters and lets you call it "cb"
    while (cb.get_yp()) >= 0:  # get_yp() is a function in the Projectile class which gets the height of the ball.
        Because this function sits in a class, it is called a "method" for that class
        cb.update(t)  # a method that updates position of projectile
        # Here you print cb's updated position
```
```
The second main function looks and feels cleaner and more organized than the first (which is already well organized except for the ugly function call with so many parameters. It's always good to keep parameter sets very small.

Of course, we now have to learn how to write the Projectile class so that the code can actually run!

```

# 3.py

Here we look at a simple example of how to define a class in Python.

The monopoly die you are familiar with is a cube with 6-sides. What if you wanted to be able to toss a die with n sides, for any n >= 1? In essence, you want something that will give you a random integer between 1 and n, both ends included.

Let's call it a "Many-sided Die" (MSD). It will be a Python object.

1. What does such a MSD know? It knows how many sides it has (n) and its current value (i.e., which side faces up). These are called "instance variables". That is, you instantiate an MSD object with those values.

2. What might you like MSD to be able to do? It can do whatever you want it to do as long as you write a function to make it be able to do that. This function is called a "method" for the MSD class. Here are some examples:

   roll the MSD: roll()           get some face value to show "up"
   cheat: setValue()             make some face value you like show "up"
   look:  getValue()             what is the current face
value showing "up"?

...

#msd.py    an n-sided die, using Python class definition

from random import randrange

# notice below how we will define a class, its constructor (a function that initializes it), and # its many methods.

class MSD:

    def __init__(self, n):
        # self is a "special parameter" used to make whatever object you are # defining refer (point) to itself.
        You don't have to use the word "self" # but the convention is to use that word because other languages # also use it, though they do not have to use it as a parameter

        self.n = n
        self.value = 1  #any valid value is okay for an initial value

    def roll (self):
        # note that roll() throws but does not return a value

        self.value = randrange(1,self.n + 1)  # adding 1 because of how randrange works # calling roll is like tossing the MSD object

    def getValue(self):
        # to get the face value after throwing we must call getValue()

        return(self.value)  #notice how we are working with the instance variables defined # in the class init function? __init__() is the CONSTRUCTOR

    def setValue(self, value):
        # since self always has to be there, we'll say setValue() has only # one parameter which is "value"
# Okay. Our class is ready. Let's use it to play a game. 

''' You have an n-sided die, and you control this simple game.

There are 2 players, each has his/her own n-sided die.

1. You throw the die once. Of course, some number between 1 and n will show as face value.

2. Each of the players is given k throws of his/her die. At the end of k throws, the player who
has thrown the value closest to the value that you threw at the start wins the game, counting
over all k games. Each player can choose how many sides his MSD will have.

Here is another rule. We have not coded this in but you can add a couple of if-statements to
add this rule if you like.

3. If either player (or both) hit your initial value on any throw we stop the game. That player wins
with a "bullseye" or its a tie.

... 

def youtoss(msd0):
    return(msd0.roll())

def player1toss(msd1):
    return(msd1.roll())

def player2toss(msd2):
    return(msd2.roll())

def main():
    my_n = eval(input("How many sides on my die?: "))
    my_msd = MSD(my_n)

    pl_n = eval(input("How many sides on die for pl?: "))
    pl_msd = MSD(pl_n)
p2_n = eval(input("How many sides on die for p2?: "))
p2_msd = MSD(p2_n)

# now each of us has his/her own die. We can start to toss.

k = eval(input("how many tossing rounds will we allow?: "))

my_msd.roll()
my_value = my_msd.getValue()

print("I rolled ",my_value,"\n")

p1score = 0
p2score = 0

for j in range(k):
    p1_msd.roll()
p1_throw = p1_msd.getValue()

    p2_msd.roll()
p2_throw = p2_msd.getValue()

    if (p1_throw != p2_throw): # if its a tie no point in updating score
        print("P1 threw ",p1_throw, "  and P2 threw", p2_throw)
        if (abs(p1_throw - my_value) < abs(p2_throw - my_value)):
            p1score = p1score + 1
            print("  and P1 wins \n")
        else:
            p2score = p2score + 1
            print("  and P2 wins \n")

print ("Total score for P1: ",p1score)
print ("Total score for P2: ",p2score)

if (p1score < p2score):
    print("  P1 wins!")
elif (p2score < p1score):
    print("  P2 wins!")
else:
print(" It''s a TIE!")

def __init__(self, angle, velocity, height):
    self.xp = 0
    self.yp = 0
    theta = radians(angle)
    self.xv = velocity * cos(theta)
    self.yv = velocity * sin(theta)

def position_update(self, t):
    self.xp = self.xp + t*self.xv

    yv_new = self.yv - 9.8*t

    average_velocity = (self.yv + yv_new)/2

    self.yp = self.yp + t * average_velocity
self.yv = yv_new

def height(self):
    return(self.yp)

def distance(self):
    return(self.xp)

#___________ code below is not in the class

def get_input_values():

    a = eval(input("Enter launch angle (degrees): "))
    v = eval(input("Enter initial velocity (meters/sec): "))
    h = eval(input("Enter initial height (meters): "))
    t = eval(input("Enter small time-interval between position updates of cb: "))

    return(a, v, h, t)

# Here is a main program that uses the above class definition for projectiles
# Note that you can add as many methods as you need, and you can make them as complex as you need

def main():

    angle, vel, h0, t = get_input_values()

    cb = Projectile(angle, vel, h0) # create a projectile object, call it cb (cannonball)

    while (cb.height() >= 0):
        # as long as cb is in the air
        cb.position_update(t) # how much does it move in a small time-interval t?

        print("CB is now {0:0.2f} meters high and is {1:0.2f} meters away.").format(cb.height(),cb.distance())

main()