```python
# Stock modules
import os
import sys
import logging
import time
import dronekit
import threading
import numpy
import math

# Custom modules
from connect import Connect  # For connecting to the vehicle
from observe import Observe  # For observing the state of the vehicle
from threads import thrd  # For multithreading capabilities
import sound  # For playing sounds on the background (without affecting main thread)
import angle  # For operations with angles (to avoid discontinuities)
from control import Control  # For taking and giving control to the pilot, checking if it was successful
from auto import Auto  # For controling autonomous flight
from sonar import Sonar # For sonar sensors operation

## Set-up logging ##
logFilename=os.path.dirname(os.path.realpath(__file__))+"/logs/"+str(time.strftime("%Y%m%d-%H%M%S"))+".txt"

fid=open(logFilename,"w")   # Open and then close to create a new file
fid.close()

logging.basicConfig(filename=logFilename,level=logging.DEBUG,format='%(asctime)s %(name)s:%(levelname)s %(message)s')


#### Step 1: Connect to vehicle ####

logStr = "\nStart of script"
print logStr
logging.info(logStr)

vehicle = Connect()

logStr = "Vehicle connected"
print logStr
logging.info(logStr)

#### Step 2: Observe state until "take control" condition is met ####

sonars=[Sonar(3,4),Sonar(14,15),Sonar(17,18)]

"""
for c in range(10): # Measure several times to have data on velocity
    for s in range(3):
        sonars[s].measureDistance()
        sonars[s].computeVelocity()
"""

for c in range(10):     # Pre-populate arrays
    print ""

    for s in range(3):
        sonars[s].measureDistance()
        sonars[s].computeVelocity()
        sonars[s].calculateCollision()
```

```

        logStr = "S%d>> Distance: %.3f [m] Velocity: %.2f [m/s]  Tcollision: %.2f [s]  Tsafe: %.2f [s]" % (s,sonars[s].avgDistance,sonars[s].avgVelocity,sonars[s].Tcollision,sonars[s].Tsafe)
        print logStr
        logging.info(logStr)


logStr = "Starting measurements"
print logStr
logging.info(logStr)

while not sonars[s].avgDistance < 2: # (sonars[s].Tsafe < 0 and sonars[s].Tcollision > 0):
#while not avgDistance < 1:

    for s in range(3):

        sonars[s].measureDistance()
        sonars[s].computeVelocity()
        sonars[s].calculateCollision()

        logStr = "S%d>> Distance: %.3f [m] Velocity: %.2f [m/s]  Tcollision: %.2f [s]  Tsafe: %.2f [s]" % (s,sonars[s].avgDistance,sonars[s].avgVelocity,sonars[s].Tcollision,sonars[s].Tsafe)
        print logStr
        logging.info(logStr)

    logStr = ""
    print logStr
    logging.info(logStr)


logStr = "Condition met"
print logStr
logging.info(logStr)

sound.beep(440, 200)


#### Step 3: Take control ####

def changeMode(mode):
    vehicle.mode = dronekit.VehicleMode(mode)


def checkMode(mode):
    return vehicle.mode.name==mode


ctrl = Control(takeFun=changeMode, checkTakeFun=checkMode, giveFun=changeMode, checkGiveFun=checkMode,
               takeArgs="GUIDED", checkTakeArgs="GUIDED", giveArgs="LOITER", checkGiveArgs="LOITER")

logStr = "Taking control"
print logStr
logging.info(logStr)

ctrl.take()
ctrl.checkTake()

while not threading.activeCount() <= 3:
```

```python
    time.sleep(0.02)

logStr = "Control taken"
print logStr
logging.info(logStr)


#### Step 4: Autonomous flight ####

def do_move(distance,tMove,direction=[1,0,0]):

    # def goto_position_target_local_ned(north, east, down):
    #   """
    #   Send SET_POSITION_TARGET_LOCAL_NED command to request the vehicle fly to a specified
    #   location in the North, East, Down frame.

    #   It is important to remember that in this frame, positive altitudes are entered as negative
    #   "Down" values. So if down is "10", this will be 10 metres below the home altitude.

    #   At time of writing, acceleration and yaw bits are ignored.

    #   """
    #   msg = vehicle.message_factory.set_position_target_local_ned_encode(
    #       0,       # time_boot_ms (not used)
    #       0, 0,    # target system, target component
    #       mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
    #       0b0000111111111000, # type_mask (only positions enabled)
    #       north, east, down, # x, y, z positions (or North, East, Down in the MAV_FRAME_BODY_NED frame
    #       0, 0, 0, # x, y, z velocity in m/s  (not used)
    #       0, 0, 0, # x, y, z acceleration (not supported yet, ignored in GCS_Mavlink)
    #       0, 0)    # yaw, yaw_rate (not supported yet, ignored in GCS_Mavlink)
    #   # send command to vehicle
    #   vehicle.send_mavlink(msg)


    def body2ned(frontBody,leftBody,upBody=-vehicle.location.global_relative_frame.alt):

        yaw=vehicle.attitude.yaw
        yawCorrected=yaw+40/180/math.pi # Weird offset. Don't know why, but it works
        north=frontBody*math.cos(yawCorrected)+leftBody*math.sin(yawCorrected)
        east=frontBody*math.sin(yawCorrected)-leftBody*math.cos(yawCorrected)
        down=-upBody
        return [north,east,down]

    def ned2global(original_location, dNorth, dEast, dDown=0):
        """
        Returns a LocationGlobal object containing the latitude/longitude `dNorth` and `dEast` metres from the
        specified `original_location`. The returned LocationGlobal has the same `alt` value
        as `original_location`.

        The function is useful when you want to move the vehicle around specifying locations relative to
        the current vehicle position.

```

```
        The algorithm is relatively accurate over small distances (10m within 1km) except close to the poles.

        For more information see:
        http://gis.stackexchange.com/questions/2951/algorithm-for-offsetting-a-latitude-longitude-by-some-amount-of-meters
        """
        earth_radius=6378137.0 #Radius of "spherical" earth
        #Coordinate offsets in radians
        dLat = dNorth/earth_radius
        dLon = dEast/(earth_radius*math.cos(math.pi*original_location.lat/180))
        dAlt = -dDown

        #New position in decimal degrees
        newlat = original_location.lat + (dLat * 180/math.pi)
        newlon = original_location.lon + (dLon * 180/math.pi)
        newalt = original_location.alt + dAlt
        if type(original_location) is dronekit.LocationGlobal:
            targetlocation=dronekit.LocationGlobal(newlat, newlon, newalt)
        elif type(original_location) is dronekit.LocationGlobalRelative:
            targetlocation=dronekit.LocationGlobalRelative(newlat, newlon, dAlt)
        else:
            raise Exception("Invalid Location object passed")

        return targetlocation;


    vehicle.simple_goto(ned2global(vehicle.location.global_frame,body2ned(distance*direction[0],distance*direction[1],distance*direction[2])[0],body2ned(distance*direction[0],distance*direction[1],distance*direction[2])[1],body2ned(distance*direction[0],distance*direction[1],distance*direction[2])[2]))
    # goto_position_target_local_ned(*body2ned(distance*direction[0],distance*direction[1],distance*direction[2]))
    print "Moving"

    time.sleep(tMove+1)


def wait(seconds):
    time.sleep(seconds)
    return True


autoMove = Auto(do_move, wait, [3,10,[0,0,1]], 10)


print "Starting autonomous flight"
autoMove.fly()
autoMove.stop()

while not threading.activeCount() <= 3:
    time.sleep(0.02)
print "Mission finished"


#### Step 5: Return control to the pilot ####

logStr = "Returning control"
print logStr
logging.info(logStr)

# Recovering ctrl class instance that was created in step 3
ctrl.give()
```

```
ctrl.checkGive()

while not threading.activeCount() <= 3:
    time.sleep(0.02)

logStr = "Control returned"
print logStr
logging.info(logStr)

sound.tripleBeep(700, 150, 600, 150, 500, 300)

logStr = "\nTerminating script\n"
print logStr
logging.info(logStr)
vehicle.close()
```