Assignment 1 Solution

Jay Mody

January 28, 2020

This report outlines the results of implementing and testing two modules in python: DateT, an ADT that represents date, and GPosT, and ADT that represents position. These modules are implemented using a given design specification. This report will also discuss critiques of the given design specifications, and answer questions about software practice and engineering as a discipline in general.

1 Testing of the Original Program

1.1 Assumptions

1.1.1 DateT ADT

I based my assumptions for the DateT ADT off of the python datetime module implementation (taken from `docs.python.org`), namely:

> ”An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. ”

Here's a summary of what this includes:

- The calendar has three main attributes, a year, month, and day.
- The first year is 1, and the last year is 9999.
- A year contains 12 months, with each month containing the following number of days (in order):
  - January (31 days)
  - February (28 days, 29 on a leap year)
  - March (31 days)

  - April (30 days)
  - May (31 days)
  - June (30 days)
  - July (31 days)
  - August (31 days)
  - September (30 days)
  - October (31 days)
  - November (30 days)
  - December (31 days)

- As a result of the above, a year contains 365 days, except on leap years, where there is an additional day in February, making a leap year contain 366 days.

- Leap years happen every 4 years, starting from year 4. Leap years do not occur on years that are a multiple of 100, unless they are also a multiple of 400 (ie 300 is not a leap year but 800 is).

In addition, I also had to make a couple assumptions about a couple of the functions:

- The add days function allows negative inputs, which would represent travelling back in time from the current date.

- The return value of the days between method may be negative, which would indicate that the inputed date comes $|n|$ days before the current one.

1.1.2 GPosT ADT

I based my assumptions for the GPosT ADT off of the website `https://www.movable-type.co.uk/scripts/latlong.html`. Namely:

- The longitude and latitude are represented as signed decimal degrees, where longitude (represented by the symbol λ) must be on the range [-180, 180], and latitude (represented by the symbol ϕ) must be on the range [-90, 90].

- The distance and move functions are modeled by the equations provided by the website (with the distance function specifically using the Haversine formula).

In addition to these basic assumptions, I also had to make a couple of assumptions about a couple of the functions:

- Speed and distance (for the arrival date and move function) are measured in terms of km and hours, and may be negative (indicates opposite direction) with no restriction on the input range.
- The bearing paramter in the move function has no restriction on the input range, and is represented as a signed decimal degree (ie 360 is equivalent to 720 which is equivalent to 0).
- For the arrival date function, the start time is assumed to be 12:00 AM, meaning if a decimal number of days pass, the decimal is dropped. It also does not take into account time zones.

1.2 Approach

My test approach involved creating 2-4 test cases for each function. I tried to include the following types of test cases for each function:

- A trivial "normal" test case
- A trivial edge case (-1, 0, max limit, boundary testing, month changes, etc ...)
- A non-trivial edge case (leap year)

Additionally, for the constructors, I tested a range of both valid and invalid inputs to make sure the correct errors were being raised for invalid inputs, and aren't being raised for valid ones.

1.3 Results

Below I put the log of the pytest results after running the test driver on my code (spoiler alert, I passed them all):

```
src/test_driver.py::test_DateT_init PASSED
src/test_driver.py::test_DateT_day PASSED
src/test_driver.py::test_DateT_month PASSED
src/test_driver.py::test_DateT_year PASSED
src/test_driver.py::test_DateT_equal PASSED
src/test_driver.py::test_DateT_next PASSED
src/test_driver.py::test_DateT_prev PASSED
src/test_driver.py::test_DateT_before PASSED
src/test_driver.py::test_DateT_after PASSED
src/test_driver.py::test_DateT_add_days PASSED
```

```
src/test_driver.py::test_DateT_days_between PASSED
src/test_driver.py::test_GPosT_init PASSED
src/test_driver.py::test_GPosT_lat PASSED
src/test_driver.py::test_GPosT_long PASSED
src/test_driver.py::test_GPosT_west_of PASSED
src/test_driver.py::test_GPosT_north_of PASSED
src/test_driver.py::test_GPosT_distance PASSED
src/test_driver.py::test_GPosT_equal PASSED
src/test_driver.py::test_GPosT_move PASSED
src/test_driver.py::test_GPosT_arrival_date PASSED
20 of 20 tests passed
```

2 Results of Testing Partner's Code

Here's the summary of the pytest results after running my partners code on the test driver:

```
src/test_driver.py::test_DateT_init PASSED
src/test_driver.py::test_DateT_day PASSED
src/test_driver.py::test_DateT_month PASSED
src/test_driver.py::test_DateT_year PASSED
src/test_driver.py::test_DateT_equal PASSED
src/test_driver.py::test_DateT_next PASSED
src/test_driver.py::test_DateT_prev PASSED
src/test_driver.py::test_DateT_before PASSED
src/test_driver.py::test_DateT_after PASSED
src/test_driver.py::test_DateT_add_days FAILED
src/test_driver.py::test_DateT_days_between FAILED
src/test_driver.py::test_GPosT_init PASSED
src/test_driver.py::test_GPosT_lat PASSED
src/test_driver.py::test_GPosT_long PASSED
src/test_driver.py::test_GPosT_west_of PASSED
src/test_driver.py::test_GPosT_north_of PASSED
src/test_driver.py::test_GPosT_distance PASSED
src/test_driver.py::test_GPosT_equal PASSED
src/test_driver.py::test_GPosT_move PASSED
src/test_driver.py::test_GPosT_arrival_date FAILED
17 of 20 tests passed
```

3 of the 20 function tests failed, below are pytest log for the failure and an explanation for the error.

2.1 add days

```
  def test_DateT_add_days():
      assert DateT(13, 12, 2021).add_days(12).equal(DateT(25, 12, 2021))
>     assert DateT(29, 1, 1600).add_days(-100).equal(DateT(21, 10, 1599)) # mo

  self = <date_adt.DateT object at 0x1084cc210>, n = -100

  def add_days(self, n):
    if n < 0:
>       raise ValueError("ERROR: Days to add cannot be a negative number")
E       ValueError: ERROR: Days to add cannot be a negative number
```

For this function, we made two different assumptions. I assumed a negative number of days to add were allowed (which would indicate moving backwards in time), while my partner assumed that negative days were not allowed, and to raise a ValueError if a negative number was passed to the function.

2.2 days between

```
  def test_DateT_days_between():
    assert DateT(10, 12, 2000).days_between(DateT(20, 12, 2000)) == 10 # betw
>   assert DateT(29, 3, 2014).days_between(DateT(29, 3, 2013)) == -365 # 365
E   assert 365 == -365
E     -365
E     +-365
```

Once again, we made two different assumptions. I asssumed that if the inputed date d came before the date of the object, the days between would be negative. In contrast, my partner assumed that days between is always positive, and took the absolute value of the difference.

2.3 arrival date

```
  def test_GPosT_arrival_date():
      start_date = DateT(1, 1, 2000)
      start_pos = GPosT(0, 0)
```

```
        target_pos = GPosT(25, 25)

>       assert start_pos.arrival_date(target_pos, start_date, 100).equal(DateT(
E       assert False
E        +  where False = <bound method DateT.equal of <date_adt.DateT object at
E        +    where <bound method DateT.equal of <date_adt.DateT object at 0x108
E        +      where <date_adt.DateT object at 0x10853df90> = <bound method GPo
E        +        where <bound method GPosT.arrival_date of <pos_adt.GPosT objec
E        +          and <date_adt.DateT object at 0x10853dfd0> = DateT(8, 2, 200
```

For this test, I assumed that if the computed number of days it would take to get from point A to point B was a decimal number, to simply ignore the decimal. My rational for this was that if you left at 12:00 AM, and it took you 0.99 days to get to your destination, the date has still yet to change. In contrast, my partner decided that if the computed number of days was a decimal number, to take the ceiling of that number via math.ciel, which is why the test failed (his arrival date was one ahead of mine). We both ignored time zones.

3 Critique of Given Design Specification

The design specification was not very complete, leaving a lot of room for assumptions and ambiguity. This is evident with the differences in the test results between me and my partners code. Of the 3 failed tests, all of them were a result of different assumptions and not incorrect implementations.

One element I did like about the design specification is that the implementation details were not specified (like what state variables to use), which gave us flexibility in how we may approac the problem. It also puts the focus on the module interface rather than biasing how we should implement the interface.

4 Answers to Questions

(a)

E Code for date_adt.py

```
## @file date_adt.py
#  @title date_adt
#  @author Reneuel Dela Cruz
#  @date 2020-01-20

from datetime import date, timedelta

## @brief An ADT for date-related comparisons and calculations.
#  @details This class creates an ADT for date-related comparisons and
#  calculations given the day, month, and year as integers.
class DateT:

    ## @brief Constructor for DateT.
    #  @details Constructor accepts three parameters to initialize the date.
    #  @param d Integer for the day of the month.
    #  @param m Integer representing the month.
    #  @param y Integer representing the year.
    #  @throws ValueError Error if the date entered cannot exist in a calendar, such as negative dates.
    #  @throws TypeError Error if the date values are not integers.
    def __init__(self, d, m, y):
        try:
            """
            Single-line in if statement to check all() parameters for int by Cory Kramer
            Cited from:
                https://stackoverflow.com/questions/25297272/how-to-make-sure-if-parameter-is-a-list-of-numbers-python
            """
            nums = [d, m, y]
            if not all(isinstance(i, int) for i in nums):
                raise TypeError ("ERROR: Date values can only be integers")

            #Creates datetime.date object into self.__date
            self.__date = date(y, m, d)
        except ValueError:
            raise ValueError("ERROR: Entered date values cannot exist together")

    ## @brief This function gets the day of the month.
    #  @return Integer value for the day.
    def day(self):
        return self.__date.day

    ## @brief This function gets the numeric value of the month.
    #  @return Integer value for the month.
    def month(self):
        return self.__date.month

    ## @brief This function returns the numeric year value.
    #  @return Integer value for the year.
    def year(self):
        return self.__date.year

    ## @brief Private function that converts a datetime object into a DateT object.
    #  @param d Datetime object that will be converted from.
    #  @return Converted DateT object.
    def __convert_datetime_to_DateT(self, d):
        return DateT(d.day, d.month, d.year)

    ## @brief Gets the immediate date after the current date in question.
    #  @return DateT object of the day after current date.
    def next(self):
        next_date = self.__date + timedelta(days=1)
        return self.__convert_datetime_to_DateT(next_date)

    ## @brief Gets the date immediately before the current date in question.
    #  @return DateT object of the day before current date.
    def prev(self):
        prev_date = self.__date - timedelta(days=1)
        return self.__convert_datetime_to_DateT(prev_date)

    ## @brief Checks if current date in question is before another date.
    #  @return True if the date is before the other date; false otherwise.
    def before(self, other):
        return self.__date < other.__date

    ## @brief Checks if current date in question is after another date.
    #  @return True if the date is after the other date; false otherwise.
    def after(self, other):
        return self.__date > other.__date

    ## @brief Special method to represent a DateT object as a string.
    #  @return String of the date formatted as DD/MM/YYYY.
    def __str__(self):
        """
        date.strftime(format)
        Cited from: https://docs.python.org/3/library/datetime.html#module-datetime
        """
        return self.__date.strftime("%d/%m/%Y")

    ## @brief Special method to compare if two DateT objects are the same.
    #  @return True if both objects have the same day, month, and year; false otherwise.
    def __eq__(self, other):
        return self.equal(other)

    ## @brief Determines if two DateT objects are equal.
    #  @return True if the objects have the same numeric day, month, and year.
    def equal(self, other):
        return self.__date == other.__date

    ## @brief Adds a specified number of days to a given DateT object.
    #  @param n Integer used to determine the number of days to add.
    #  @throws ValueError Error if added days is a negative number.
    #  @return DateT object after the specified number of days from current date.
    def add_days(self, n):
        if n < 0:
            raise ValueError("ERROR: Days to add cannot be a negative number")

        future_date = self.__date + timedelta(days=n)
        return self.__convert_datetime_to_DateT(future_date)

    ## @brief Calculates the number of days between two DateT objects.
    #  @return Integer of total number of days between the specified dates.
    def days_between(self, other):
        """
        timedelta = date1 - date2
        Cited from: https://docs.python.org/3/library/datetime.html#module-datetime
        """
        return abs((self.__date - other.__date).days)
```

F Code for pos_adt.py

```
## @file pos_adt.py
#  @title pos_adt
#  @author Reneuel Dela Cruz
#  @date 2020-01-20

import math
from date_adt import DateT

## @brief An ADT for representing global position coordinates.
#  @details This class creates an ADT for global position coordinates using
#  latitude and longitude as signed decimal degrees.
class GPosT:

    ## @brief Constructor for GPosT.
    #  @details Constructor accepts two parameters to initialize the global position coordinate.
    #  @param latitude Float for the latitude in signed decimal degrees.
    #  @param longitude Float for the longitude in signed decimal degrees.
    #  @throws ValueError Error if the latitude or longitude exceeds the maximum possible values.
    def __init__(self, latitude, longitude):
        if abs(latitude) > 90 or abs(longitude) > 180:
            raise ValueError("ERROR: Maximum latitude or longitude values exceeded")

        self.__latitude = latitude
        self.__longitude = longitude

    ## @brief This function gets the position's latitude.
    #  @return Float value for latitude.
    def lat(self):
        return self.__latitude

    ## @brief This function gets the position's longitude.
    #  @return Float value for longitude.
    def long(self):
        return self.__longitude

    ## @brief Checks if current position is west of another position.
    #  @details Checks if current longitude is less than another position's, since navigation
    #  convention has negative longitudes for the western hemisphere and positive for the eastern.
    #  @return True if the current longitude is west of the other longitude; false otherwise.
    def west_of(self, other):
        return self.__longitude < other.__longitude

    ## @brief Checks if current position is north of another position.
    #  @details Checks if current latitude is more than another position's, since navigation
    #  convention has positive latitudes for the northern hemisphere and negative for the southern.
    #  @return True if the current latitude is north of the other latitude; false otherwise.
    def north_of(self, other):
        return self.__latitude > other.__latitude

    ## @brief Special method to represent a GPosT object as a string.
    #  @return String of the coordinate formatted as [latitude, longitude].
    def __str__(self):
        return '[{}, {}]'.format(self.__latitude, self.__longitude)

    ## @brief Special method to compare two GPosT objects.
    #  @return True if both positions are within 1 km of each other; false otherwise.
    def __eq__(self, other):
        return self.equal(other)

    ## @brief Checks if two GPosT objects are equal.
    #  @return True if the coordinates have less than 1 km of distance between each other.
    def equal(self, other):
        if self.distance(other) <= 1:
            return True

        return False

    ## @brief Calculates distance between two coordinates.
    #  @details Calculates the distance between two GPosT objects
    #  in km using the Haversine Formula.
    #  @return Float distance between the two positions in km.
    def distance(self, other):
        """
        Haversine Formula:
        a = sin(delta_lat/2)^2 + cos lat1 * cos lat2 * sin(delta_long/2)^2
        c = 2 * atan2(sqrt(a),sqrt(1-a))
    d = R * c

    Cited from: https://www.movable-type.co.uk/scripts/latlong.html
    """

    #Degrees changed to radians to work with math library
    lat1 = math.radians(self.__latitude)
    lat2 = math.radians(other.__latitude)
    delta_lat = math.radians(other.__latitude - self.__latitude)
    delta_long = math.radians(other.__longitude - self.__longitude)

    a = math.sin(delta_lat/2) ** 2 + math.cos(lat1) * math.cos(lat2) * math.sin(delta_long/2) ** 2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
    #Average radius of the Earth is 6371 km according to www.movable-type.co.uk
    return 6371 * c

## @brief Moves a GPosT object in a specified direction and distance.
#  @details This function changes the longitude and latitude of a GPosT object towards a
#  degrees bearing direction over a specified distance in km.
#  @param bearing Number representing the bearing direction in degrees.
#  @param distance Number for the distance to travel in km.
#  @throws ValueError Error if the bearing exceeds 360 degrees
#  @throws ValueError Error if the distance is negative
def move(self, bearing, distance):
    if abs(bearing) > 360:
        raise ValueError("ERROR: Bearing cannot exceed 360 degrees")
    if distance < 0:
        raise ValueError("ERROR: Distance travelled cannot be negative")

    """
    Destination Using Bearing and Distance:
    lat2 = asin(sin lat1 * cos d + cos lat1 * sin d * cos b)
    long2 = long1 + atan2(sin b * sin d * cos lat1, cos d - sin lat1 * sin lat2)

    Cited from: https://www.movable-type.co.uk/scripts/latlong.html
    """
    latitude = math.radians(self.__latitude)
    longitude = math.radians(self.__longitude)
    angular_dist = distance / 6371
    bearing = math.radians(bearing)

    new_lat = math.asin(math.sin(latitude) * math.cos(angular_dist) + math.cos(latitude) *
        math.sin(angular_dist) * math.cos(bearing))
    new_long = longitude + math.atan2(math.sin(bearing) * math.sin(angular_dist) *
        math.cos(latitude),
                                      math.cos(angular_dist) - math.sin(latitude) *
                                          math.sin(new_lat))

    self.__latitude = math.degrees(new_lat)
    self.__longitude = math.degrees(new_long)

## @brief Determines the arrival date based on starting point and speed.
#  @details This function calculates the date of arrival to a specified position given the
#  starting date and the speed of travel in km/day.
#  @param position GPosT object for the destination.
#  @param date DateT object for the starting date.
#  @param speed Travelling speed in km/day.
#  @throws ValueError Error if speed is negative.
#  @throws ZeroDivisionError Error if speed is zero which will cause division by zero.
#  @return DateT object representing date of arrival.
def arrival_date(self, position, date, speed):
    if speed < 0:
        raise ValueError("ERROR: Speed cannot be negative")
    if speed == 0:
        raise ZeroDivisionError("ERROR: Speed cannot be zero")

    #Distance in km
    distance = self.distance(position)
    #Fractional days are rounded up
    num_of_days = math.ceil(distance/speed)
    return date.add_days(num_of_days)
```

G Code for test_driver.py

```python
## @file test_driver.py
#  @author Jay Mody
#  @brief Tests driver for the DateT ADT and GPosT ADT.
#  @date 20/01/20 (dd/mm/yy)

from date_adt import DateT
from pos_adt import GPosT

import pytest

# DateT tests
def test_DateT_init():
    with pytest.raises(ValueError):
        DateT(-1, 1, 2000)
    with pytest.raises(ValueError):
        DateT(0, 1, 2000)
    with pytest.raises(ValueError):
        DateT(100, 1, 2000)
    with pytest.raises(ValueError):
        DateT(10, -1, 2000)
    with pytest.raises(ValueError):
        DateT(10, 0, 2000)
    with pytest.raises(ValueError):
        DateT(-1, 13, 2000)
    with pytest.raises(ValueError):
        DateT(1, 1, 10000)
    with pytest.raises(ValueError):
        DateT(1, 1, 0)

def test_DateT_day():
    assert DateT(23, 2, 2012).day() == 23
    assert DateT(1, 2, 2012).day() == 1

    assert DateT(31, 1, 2012).day() != 30
    assert DateT(14, 2, 2012).day() != -14

def test_DateT_month():
    assert DateT(23, 2, 2012).month() == 2
    assert DateT(1, 12, 2012).month() == 12

    assert DateT(31, 1, 2012).month() != 2
    assert DateT(14, 2, 2012).month() != -2

def test_DateT_year():
    assert DateT(23, 2, 200).year() == 200
    assert DateT(1, 12, 2031).year() == 2031
    assert DateT(1, 12, 10).year() == 10

    assert DateT(31, 1, 2012).year() != -2012
    assert DateT(14, 2, 2012).year() != 0
    assert DateT(14, 2, 2012).year() != 20120

def test_DateT_equal():
    assert DateT(31, 12, 2021).equal(DateT(31, 12, 2021))
    assert DateT(1, 1, 1).equal(DateT(1, 1, 1))

    assert not DateT(1, 1, 1).equal(DateT(2, 1, 1))
    assert not DateT(31, 12, 2021).equal(DateT(30, 12, 2020))

def test_DateT_next():
    assert DateT(1, 2, 2012).next().equal(DateT(2, 2, 2012))
    assert DateT(28, 2, 2020).next().equal(DateT(29, 2, 2020)) # leap year
    assert DateT(28, 2, 2021).next().equal(DateT(1, 3, 2021)) # non leap year
    assert DateT(31, 12, 2021).next().equal(DateT(1, 1, 2022)) # month + year change

def test_DateT_prev():
    assert DateT(31, 12, 2021).prev().equal(DateT(30, 12, 2021))
    assert DateT(1, 3, 1600).prev().equal(DateT(29, 2, 1600)) # 400 divisible leap year
    assert DateT(1, 3, 1700).prev().equal(DateT(28, 2, 1700)) # 100 divisible non leap year
    assert DateT(1, 2, 2012).prev().equal(DateT(31, 1, 2012)) #  month change

def test_DateT_before():
    assert DateT(30, 12, 2021).before(DateT(31, 12, 2021)) # days before
    assert DateT(1, 2, 1600).before(DateT(1, 3, 1600)) # months before
    assert DateT(1, 3, 1).before(DateT(1, 3, 1700)) # years before
```

```
def test_DateT_after():
    assert DateT(13, 12, 2021).after(DateT(12, 12, 2021)) # days after
    assert DateT(29, 3, 1600).after(DateT(29, 1, 1600)) # months after
    assert DateT(1, 3, 1701).after(DateT(28, 2, 1700)) # years after

def test_DateT_add_days():
    assert DateT(13, 12, 2021).add_days(12).equal(DateT(25, 12, 2021))
    assert DateT(29, 1, 1600).add_days(-100).equal(DateT(21, 10, 1599)) # month + year change

def test_DateT_days_between():
    assert DateT(10, 12, 2000).days_between(DateT(20, 12, 2000)) == 10 # between years
    assert DateT(29, 3, 2014).days_between(DateT(29, 3, 2013)) == -365 # 365 (year) negative days
        between


# GPosT tests
def test_GPosT_init():
    with pytest.raises(ValueError):
        GPosT(-90.0001, 0)
    with pytest.raises(ValueError):
        GPosT(90.0001, 0)
    with pytest.raises(ValueError):
        GPosT(0, -180.0001)
    with pytest.raises(ValueError):
        GPosT(0, 180.0001)

    assert GPosT(-90., 0)
    assert GPosT(90., 0)
    assert GPosT(0, -180.)
    assert GPosT(0, 180.)

def test_GPosT_lat():
    assert GPosT(23., 0).lat() == 23
    assert GPosT(-12.1231, 1.).lat() == -12.1231

    assert GPosT(23.000001, 23).lat() != 23
    assert GPosT(23, -23).lat() != -23

def test_GPosT_long():
    assert GPosT(23., 0).long() == 0
    assert GPosT(2.1231, 1.).long() == 1.

    assert GPosT(1.01, 1.01).long() != 1.
    assert GPosT(23, -23).long() != 23

def test_GPosT_west_of():
    assert GPosT(1, 0).west_of(GPosT(2, 1))
    assert not GPosT(28, -2).west_of(GPosT(21, -20))

def test_GPosT_north_of():
    assert GPosT(31, 12).north_of(GPosT(30, 14))
    assert not GPosT(-21, 3).north_of(GPosT(-20, 2))

## @cite used https://www.movable-type.co.uk/scripts/latlong.html to find expected distance outputs
def test_GPosT_distance():
    assert (abs(1805.5 - GPosT(-1, 2).distance(GPosT(10, -10))) < 1)
    assert not (abs(212 - GPosT(-11, 2).distance(GPosT(10, -10))) < 1)

def test_GPosT_equal():
    assert GPosT(-1, 2).equal(GPosT(-1, 2))
    assert GPosT(-1.001, 2).equal(GPosT(-1, 2.001))
    assert not GPosT(-1.2, 2).equal(GPosT(0, 0))
    assert not GPosT(-20, 20).equal(GPosT(20, -20))

## @cite used https://www.latlong.net/degrees-minutes-seconds-to-decimal-degrees to calculate expected
    output
def test_GPosT_move():
    pos = GPosT(10, 0)
    pos.move(30, 1000)

    # test if within 1m of target lat/long
    assert abs(17.74805556 - pos.lat()) < 0.001
    assert abs(4.70722222 - pos.long()) < 0.001

def test_GPosT_arrival_date():
    start_date = DateT(1, 1, 2000)
    start_pos = GPosT(0, 0)
    target_pos = GPosT(25, 25)

    assert start_pos.arrival_date(target_pos, start_date, 100).equal(DateT(8, 2, 2000))
```

H Code for Partner's CalcModule.py

```
## @file pos_adt.py
#  @title pos_adt
#  @author Reneuel Dela Cruz
#  @date 2020-01-20

import math
from date_adt import DateT

## @brief An ADT for representing global position coordinates.
#  @details This class creates an ADT for global position coordinates using
#  latitude and longitude as signed decimal degrees.
class GPosT:

    ## @brief Constructor for GPosT.
    #  @details Constructor accepts two parameters to initialize the global position coordinate.
    #  @param latitude Float for the latitude in signed decimal degrees.
    #  @param longitude Float for the longitude in signed decimal degrees.
    #  @throws ValueError Error if the latitude or longitude exceeds the maximum possible values.
    def __init__(self, latitude, longitude):
        if abs(latitude) > 90 or abs(longitude) > 180:
            raise ValueError("ERROR: Maximum latitude or longitude values exceeded")

        self.__latitude = latitude
        self.__longitude = longitude

    ## @brief This function gets the position's latitude.
    #  @return Float value for latitude.
    def lat(self):
        return self.__latitude

    ## @brief This function gets the position's longitude.
    #  @return Float value for longitude.
    def long(self):
        return self.__longitude

    ## @brief Checks if current position is west of another position.
    #  @details Checks if current longitude is less than another position's, since navigation
    #  convention has negative longitudes for the western hemisphere and positive for the eastern.
    #  @return True if the current longitude is west of the other longitude; false otherwise.
    def west_of(self, other):
        return self.__longitude < other.__longitude

    ## @brief Checks if current position is north of another position.
    #  @details Checks if current latitude is more than another position's, since navigation
    #  convention has positive latitudes for the northern hemisphere and negative for the southern.
    #  @return True if the current latitude is north of the other latitude; false otherwise.
    def north_of(self, other):
        return self.__latitude > other.__latitude

    ## @brief Special method to represent a GPosT object as a string.
    #  @return String of the coordinate formatted as [latitude, longitude].
    def __str__(self):
        return '[{}, {}]'.format(self.__latitude, self.__longitude)

    ## @brief Special method to compare two GPosT objects.
    #  @return True if both positions are within 1 km of each other; false otherwise.
    def __eq__(self, other):
        return self.equal(other)

    ## @brief Checks if two GPosT objects are equal.
    #  @return True if the coordinates have less than 1 km of distance between each other.
    def equal(self, other):
        if self.distance(other) <= 1:
            return True

        return False

    ## @brief Calculates distance between two coordinates.
    #  @details Calculates the distance between two GPosT objects
    #  in km using the Haversine Formula.
    #  @return Float distance between the two positions in km.
    def distance(self, other):
        """
        Haversine Formula:
        a = sin(delta_lat/2)^2 + cos lat1 * cos lat2 * sin(delta_long/2)^2
        c = 2 * atan2(sqrt(a),sqrt(1-a))
        d = R * c

        Cited from: https://www.movable-type.co.uk/scripts/latlong.html
        """

        #Degrees changed to radians to work with math library
        lat1 = math.radians(self.__latitude)
        lat2 = math.radians(other.__latitude)
        delta_lat = math.radians(other.__latitude - self.__latitude)
        delta_long = math.radians(other.__longitude - self.__longitude)

        a = math.sin(delta_lat/2) ** 2 + math.cos(lat1) * math.cos(lat2) * math.sin(delta_long/2) ** 2
        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
        #Average radius of the Earth is 6371 km according to www.movable-type.co.uk
        return 6371 * c

    ## @brief Moves a GPosT object in a specified direction and distance.
    #  @details This function changes the longitude and latitude of a GPosT object towards a
    #  degrees bearing direction over a specified distance in km.
    #  @param bearing Number representing the bearing direction in degrees.
    #  @param distance Number for the distance to travel in km.
    #  @throws ValueError Error if the bearing exceeds 360 degrees
    #  @throws ValueError Error if the distance is negative
    def move(self, bearing, distance):
        if abs(bearing) > 360:
            raise ValueError("ERROR: Bearing cannot exceed 360 degrees")
        if distance < 0:
            raise ValueError("ERROR: Distance travelled cannot be negative")

        """
        Destination Using Bearing and Distance:
        lat2 = asin(sin lat1 * cos d + cos lat1 * sin d * cos b)
        long2 = long1 + atan2(sin b * sin d * cos lat1, cos d - sin lat1 * sin lat2)

        Cited from: https://www.movable-type.co.uk/scripts/latlong.html
        """
        latitude = math.radians(self.__latitude)
        longitude = math.radians(self.__longitude)
        angular_dist = distance / 6371
        bearing = math.radians(bearing)

        new_lat = math.asin(math.sin(latitude) * math.cos(angular_dist) + math.cos(latitude) *
            math.sin(angular_dist) * math.cos(bearing))
        new_long = longitude + math.atan2(math.sin(bearing) * math.sin(angular_dist) *
            math.cos(latitude),
                                         math.cos(angular_dist) - math.sin(latitude) *
                                             math.sin(new_lat))

        self.__latitude = math.degrees(new_lat)
        self.__longitude = math.degrees(new_long)

    ## @brief Determines the arrival date based on starting point and speed.
    #  @details This function calculates the date of arrival to a specified position given the
    #  starting date and the speed of travel in km/day.
    #  @param position GPosT object for the destination.
    #  @param date DateT object for the starting date.
    #  @param speed Travelling speed in km/day.
    #  @throws ValueError Error if speed is negative.
    #  @throws ZeroDivisionError Error if speed is zero which will cause division by zero.
    #  @return DateT object representing date of arrival.
    def arrival_date(self, position, date, speed):
        if speed < 0:
            raise ValueError("ERROR: Speed cannot be negative")
        if speed == 0:
            raise ZeroDivisionError("ERROR: Speed cannot be zero")

        #Distance in km
        distance = self.distance(position)
        #Fractional days are rounded up
        num_of_days = math.ceil(distance/speed)
        return date.add_days(num_of_days)
```