Assignment 1 Solution

Dhruv Bhavsar, bhavsd1

January 28, 2020

This report discusses the testing results of DateT and GPosT programs written for Assignment 1. It also includes the test results of partner's code for these programs. I also discuss the quality of the given specifications and answer the given discussion questions.

1 Testing of the Original Program

Test were written in a similar format as one of the previous years'. This was the format I looked at: `https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/blob/master/Assignments/PreviousYears/2017/A1/A1Soln/src/testCircles.py`. Essentially, I made multiple objects with different information, for example for the DateT object, I made it to be the last day of the year, another to be the first day of the year and one to be a day in February that only occurs on a leap year. My objects helped me test edge cases. So, I would test each function written in the specification using these objects and comparing it with the actual answer with assertions. I would track the number of tests I would run and how many passed and display them at the end of the tests for DateT and GPosT. All of my test cases passed, doing testing on a program is something a good software engineer should always do. During my testing, I was able to discover a bug in my distance function where I forgot to use a previous calculated result of the latitude instead of the current latitude.

Some of assumptions include:

- Following the Georgian Calendar for Dates
- For the `move` function, the distance entered cannot be negative
- For the `arrival_date` function, I made a couple assumptions such as the speed provided cannot be negative as it doesn't make sense. The distance also cannot be negative. I also ceiled the days for arrival because I wanted to be accurate as possible by providing them with a Date that they will reach by.

2 Results of Testing Partner's Code

Running my `test_driver,py` for my partner's code gave almost a fully successfully testing but it failed the `days_between` function. Upon further examination, I figured out that my partner adds an extra day to their calculation. I figured this out by printing out how many days are between the same day and it returned 1. From my understanding, the days between the same day is 0. However, all the other test cases passed for my partner. It seems that we both had similar assumptions and understanding of the specifications. All of my tests passed for my partner's `pos_adt.py`.

3 Critique of Given Design Specification

The given specification was written in Natural Language, which means there is room for ambiguity if it isn't specific enough as people will have different views on it. But for the most part, I think the specification was specific enough for it to get that correct implementation. However, the specification could have been more clear on the way to implement `west_of` and `north_of` functions. Also the specification did not specify how it wants the state variables, so people could have done this in several different ways. Overall, natural language specifications have its advantages and disadvantages such as it is sometimes easier to understand what is needed to implement but sometimes it is not clear enough.

4 Answers to Questions

(a) One option for the DateT is to have one state variable that is the `datetime` object instead of storing the day, month and year separately. This option would have been efficient as I would not have to keep on make new `datetime` objects for the current object to use the `datetime` functions such as `timedelta`. Another option could have been to hold the date in a string, though this would have been really inefficient as I would need to convert and match based on the day and month to its corresponding number. For GPosT, I used two state variables to store the latitude and longitude separately. Instead of doing that, another option would be to use x, y, z coordinates but this would be really hard to use the methods as the math becomes more complicated. in addition, using degrees, minutes, seconds would be a more feasible option as it is easily converted to and from latitude and longitude.

(b) DateT is immutable because once the constructor sets the object with the date, it is not possible to change that afterwards. The functions for DateT also don't mutate

the current object, the functions create a new object and return that. But for GPosT, it is mutable because the function `move` modifies the current object based on the given parameters. Therefore, DateT is immutable and GPosT is mutable.

(c) The testing framework `pytest` provides a way to ensure the program behaves in a robust way. Overall, it helps writing better programs. The framework makes writing small tests easier but it can be scalable to more complex programs. For this assignment, I have not used `pytest` but in the future it can help me with scalability as future assignments will be more complex.

(d) One example of a past software engineering failure is Ariane 5 Flight 501. This was a new rocket that reused the same software from its predecessor, Ariane 4. Ariane 5 had to self-destruct thirty-five seconds into it's launch because of several computer failures in the engines. They were trying to cram a 64-bit number into a 16-bit size, which caused an overflow, crashing the computers onboard. They used the same software as the previous rocket without checking if it was compatible, no testing was done to make sure no software errors occurred. Ariane 5 had cost about $8 billion to develop and it was all lost in a matter of seconds. Source

Another example of a past software engineering failure is NASA's Mars Climate Orbiter. A sub contractor on the engineering team forgot to convert from English units to metric. This mistake costed $125 million for such a simple conversion. This goes to show that no matter how big the project it is always important to test software thoroughly since a simple mistake can cost a lot. Source

Software quality and high cost is still a major challenge because quality takes time and has additional costs involved. According to Wikipedia's article on Project Management Triangle, a software can only have two traits from quality, cost and time. The quality of work is restricted by the budget and deadline, as it takes time and money to produce quality software. I think making software open sourced is a way to address this issue, as the software can be reused and can add proprietary features afterwards. This way it will take less time to build the project and have more time to thoroughly test it, thus increasing quality of the work.

(e) The rational design process includes many steps similar to the waterfall method. This process is often faked because the clients who are requesting the software don't know what they want and need from the beginning, so often one goes back and forth with the specification. It is not only the specification that keeps changing, the implementation, the design, can all change during a lifecycle of the software product. It is

necessary to fake this process because it provides a clean timeline of the whole software development process as supposed to constantly moving back and forth each step. The advantages of writing documentation that follows the rational design process is that all the information about the development is organized, understandable.

(f) Correctness is when a software product meets the requirements specification, though this is very difficult to achieve because the specifications might be ambiguous, to measure correctness and achieve bug free software. Robustness is when the software handles what ever is thrown at it. What I mean by this is that it has an output for all input even the ones that seem unexpected or unanticipated. A correct software does not have to be robust as it satisfies the requirements, while the robustness satisfies the requirements that were not in the specification. Reliability is when the software is robust for a period of time. It also means the likelihood of it functioning correctly which can be measured. Not all programs are correct and robust but all correct programs are reliable. Source

(g) The principle of Separation of Concerns is to isolate different concerns and deal with them separately. Essentially, it takes a complex problem and breaks it down into smaller achievable problems. The motivation behind this principle is to enable parallelization of effort as you will be working on different tasks and trying to merge them together. The principle of Modularity and Separation of Concerns are similar because in both principles you are breaking down a complex problem into smaller easier problems. In other terms, you are looking at each part of the system separately. Source

E Code for date_adt.py

```
## @file date_adt.py
#  @author Dhruv Bhavsar
#  @brief An abstract data type that represents Date and allows for other functions to manipulate Date
#  @date Jan 15 2020

import datetime

## @brief An ADT that represents Date
class DateT:


    ## @brief Constructor that inializes the object with year, month and day
    #  @details Using a try and except, the inputted date is tested with datetime to see if its valid
      then save them in
    #  class variables else raise error.
    #  @param m - Month, d - Day, y- Year
    def __init__(self, d, m, y):
      try:
        datetime.date(y, m, d)
        self.y = y
        self.d = d
        self.m = m
      except:
        raise


    ## @brief Getter for day
    #  @return The day as an int
    def day(self):
      return self.d


    ## @brief Getter for month
    # @return The month as an int
    def month(self):
      return self.m


    ## @brief Getter for year
    #  @return The year as an int
    def year(self):
      return self.y


    ## @brief Find the next day from the current object
    #  @details Create a datetime object with current object and use timedelta function to add one day
      and extract
    #           year, month and day
    #  @return DateT object of next day
    def next(self):
      cur_date = datetime.date(self.y, self.m, self.d)
      next_date = cur_date + datetime.timedelta(days=1)
      return DateT(next_date.day, next_date.month, next_date.year)


    ## @brief Find the day before current object
    #  @details Creates a datetime object with current object and use timedelta function to subtract
      one day and extract
    #           year, month, day
    #  @return DateT object of the previous day
    def prev(self):
      cur_date = datetime.date(self.y, self.m, self.d)
      prev_date = cur_date - datetime.timedelta(days=1)
      return DateT(prev_date.day, prev_date.month, prev_date.year)


    ## @brief Finds if the current date is before the d object
    #  @details Creates a datetime object with current object and a datetime object with the parameter
      d. Then compare the
    #           datetime objects
    #  @param d - DateT object to compare with
    #  @return True if the current object is before param d else False
    def before(self, d):
      cur_date = datetime.date(self.y, self.m, self.d)
      new_date = datetime.date(d.y, d.m, d.d)
```

```
  return cur_date < new_date     # Use less than since we want to find out if date is before


## @brief Finds if the current date is after the d object
#  @details Creates a datetime object with current object and a datetime object with the parameter
    d. Then compare the
#           datetime objects
#  @param d - DateT object to compare with
#  @return True if the current object is after param d else False
def after(self, d):
  cur_date = datetime.date(self.y, self.m, self.d)
  new_date = datetime.date(d.y, d.m, d.d)
  return cur_date > new_date    # Use greater than since we want to find out if date is after


## @brief Checks if the dates are the same
#  @details Compares the day, month and year among the two objects
#  @param d - DateT object to compare with
#  @return True if the date are the same else False
def equal(self, d):
  return self.y == d.y and self.m == d.m and self.d == d.d


## @brief Adds n days to the current DateT object
#  @details Creates a datetime object with the current object, then uses the timedelta function to
    add n days and
#           finally extracts the year, month and day. If you enter a decimal number for n it will
    round to nearest
#           whole number and then add those many days. Accepts negative days (to go back days)
#  @param n - number of days to add on to the current date (int)
#  @return a DateT object with the new date
def add_days(self, n):
  cur_date = datetime.date(self.y, self.m, self.d)
  new_date = cur_date + datetime.timedelta(days=n)
  return DateT(new_date.day, new_date.month, new_date.year)


## @brief Find the days between the current object and parameter object
#  @details Creates a datetime object with the current object and a datetime object with the
    parameter object. Then
#           find the difference between the two dates by subtracting.
#  @param d - DateT object find the difference in days from
#  @return the absolute value of the difference since the user can enter a date that is before the
    current date
def days_between(self, d):
  cur_date = datetime.date(self.y, self.m, self.d)
  new_date = datetime.date(d.y, d.m, d.d)
  return abs(new_date - cur_date).days
```

F Code for pos_adt.py

```
## @file pos_adt.py
#  @author Dhruv Bhavsar
#  @brief An abstract date type for global position coordinates and allows for several functions on
    them
#  @date Jan 16 2020

import math
from date_adt import DateT

## @brief An ADT that represents global position coordinates
class GPosT:

    ## @brief Constructor that creates a new GPosT object with latitude and longitude (in degrees)
    #  @details First check if latitude and longitude are in range then save them to the class
        variables else throw
    #  value error
    #  @param latitude and longitude (in degrees) are real numbers
    def __init__(self, latitude, longitude):
        if not -90 <= latitude <= 90:
            raise ValueError("Latitude must be in -90..90")
        if not -180 <= longitude <= 180:
            raise ValueError("Longitube must be in -180..180")

        self.latitude = latitude
        self.longitude = longitude

    ## @brief Getter for latitude
    #  @return the latitude as a double
    def lat(self):
        return self.latitude

    ## @brief Getter for longitude
    #  @return the longitude as a double
    def long(self):
        return self.longitude
    ## @brief Checks if current position is west of parameter position
    #  @details
    #  @param p - GPosT object which you are comparing with
    #  @return True if the current position is west of parameter position else False
    def west_of(self, p):
        return self.longitude < p.longitude

    ## @brief Checks if current position is north of parameter position
    #  @details
    #  @param p - GPosT object which you are comparing with
    #  @return True if the current position is north of parameter position else False
    def north_of(self, p):
        return self.latitude > p.latitude

    ## @brief Checks if the distance is equal by calculating the distance
    #  @details Using the formula provided in specifications
        https://www.movable-type.co.uk/scripts/latlong.html
    #  @param p - GPosT object comparing with
    #  @return True if the distance is less than 1 km else False
    def equal(self, p):
        radius = 6371
        lat = math.radians(self.latitude)
        lat2 = math.radians(p.latitude)

        lat_diff = math.radians(p.latitude - self.latitude)
        long_diff = math.radians(p.longitude - self.longitude)

        a = math.sin(lat_diff/2) * math.sin(lat_diff/2) + math.cos(lat) * math.cos(lat2) \
            * math.sin(long_diff/2) * math.sin(long_diff/2)

        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))

        distance = radius * c

        return distance < 1

    ## @brief Moves the current object d (km) towards b (bearing)
    #  @details Using the formula provided in the specifications
        https://www.movable-type.co.uk/scripts/latlong.html
    #           under Destination point given distance and bearing from start point then the new
        position gets updated
```

```
#           to be the current. Firstly convert all degrees into radians as the formula requires
    radians. Also I
#           check if the new positions are greater than their ranges, if so then I apply a formula
    to normalize them
#           which can be found from the website given in the specifications
#           https://www.movable-type.co.uk/scripts/latlong.html
#   @param b - bearing type real and d - distance in km type real
def move(self, b, d):
    radius = 6371
    rad_lat = math.radians(self.latitude)
    rad_long = math.radians(self.longitude)
    rad_bearing = math.radians(b)

    self.latitude = math.asin(math.sin(rad_lat) * math.cos(d / radius)
                              + math.cos(rad_lat) * math.sin(d / radius) * math.cos(rad_bearing))

    self.longitude = rad_long + math.atan2(math.sin(rad_bearing) * math.sin(d / radius) *
        math.cos(rad_lat),
                                           math.cos(d / radius) - math.sin(rad_lat) *
                                               math.sin(self.latitude))
    new_lat = math.degrees(self.latitude)
    new_long = math.degrees(self.longitude)
    self.latitude = new_lat if abs(new_lat) < 90 else (new_lat + 270) % 180 - 90
    self.longitude = new_long if abs(new_long) < 180 else (new_long + 540) % 360 - 180

## @brief Calculate the distance between current position and position p
#   @details Using the formula provided in the specifications
    https://www.movable-type.co.uk/scripts/latlong.html
#           under Distance
#   @param p - GPosT object that we are finding the distance between
#   @return the distance in kilometers between the two positions
def distance(self, p):
    radius = 6371
    lat = math.radians(self.latitude)
    lat2 = math.radians(p.latitude)

    lat_diff = math.radians(p.latitude - self.latitude)
    long_diff = math.radians(p.longitude - self.longitude)

    a = math.sin(lat_diff/2) * math.sin(lat_diff/2) + math.cos(lat) * math.cos(lat2) \
        * math.sin(long_diff/2) * math.sin(long_diff/2)

    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))

    return radius * c

## @brief Calculate the arrival date starting from the current position and traveling towards
    position p at s
#         kilometers per day
#   @details First calculate the distance between current position and desired position (p), then
    divide the distance
#           by s to get days it takes to reach that position. Finally use the DateT method
    add_days to add the days
#           needed to travel the distance. The reason I took the ceil of days is because I want to
    give the date
#           that they will reach the position by. Also, it doesnt make sense to input a negative
    speed in this case.
#   @param p - GPosT object that we need to find the distance between, d - distance in km, s -
    speed in km/day
#   @return a new DateT object that represents when the arrival date is
def arrival_date(self, p, d, s):
    distance = self.distance(p)
    days = distance / s
    return d.add_days(math.ceil(days))
```

G Code for test_driver.py

```
## @file test_driver.py
#  @author Dhruv Bhavsar
#  @brief Test cases for DateT and GPosT
#  @date Jan 19 2020

#  Reference for testing format:
#
     https://gitlab.cas.mcmaster.ca/smiths/se2aa4-cs2me3/blob/master/Assignments/PreviousYears/2017/A1/A1Soln/src/testCircle

from date_adt import *
from pos_adt import *


date1 = DateT(28, 2, 2020)
date2 = DateT(3, 4, 1983)
date3 = DateT(29, 2, 2020)
date4 = DateT(22, 7, 2022)
date5 = DateT(1, 1, 2019)
date6 = DateT(31, 12, 2005)
date7 = DateT(23, 4, 2000)
date8 = DateT(29, 7, 1988)
date9 = DateT(1, 3, 2004)
date10 = DateT(22, 7, 2022)



def test_day():
    global test_total, passed
    test_total += 1
    try:
        try:
            date_invalid = DateT(29, 2, 2019)
            print("invalid DateT test FAILED")
        except ValueError:
            print("invalid DateT test PASSED")
        assert date1.day() == 28
        assert date2.day() == 3
        assert date3.day() == 29
        assert date4.day() == 22
        assert date5.day() == 1
        passed += 1
        print("day test PASSED")
    except AssertionError:
        print("day test FAILED")


def test_month():
    global test_total, passed
    test_total += 1
    try:
        assert date6.month() == 12
        assert date7.month() == 4
        assert date8.month() == 7
        assert date1.month() == 2
        passed += 1
        print("month test PASSED")
    except AssertionError:
        print("month test FAILED")


def test_year():
    global test_total, passed
    test_total += 1
    try:
        assert date1.year() == 2020
        assert date2.year() == 1983
        assert date8.year() == 1988
        passed += 1
        print("year test PASSED")
    except AssertionError:
        print("year test FAILED")


def test_next():
    global test_total, passed
    test_total += 1
```

```
    try:
        next_date = date1.next()
        assert next_date.day() == 29 and next_date.month() == 2 and next_date.year() == 2020
        next_date = date3.next()
        assert next_date.day() == 1 and next_date.month() == 3 and next_date.year() == 2020
        next_date = date6.next()
        assert next_date.day() == 1 and next_date.month() == 1 and next_date.year() == 2006
        passed += 1
        print("next test PASSED")
    except AssertionError:
        print("next test FAILED")


def test_previous():
    global test_total, passed
    test_total += 1
    try:
        prev_date = date5.prev()
        assert prev_date.day() == 31 and prev_date.month() == 12 and prev_date.year() == 2018
        prev_date = date9.prev()
        assert prev_date.day() == 29 and prev_date.month() == 2 and prev_date.year() == 2004
        prev_date = date7.prev()
        assert prev_date.day() == 22 and prev_date.month() == 4 and prev_date.year() == 2000
        new_date = date1.add_days(-365)
        passed += 1
        print("previous test PASSED")
    except AssertionError:
        print("previous test FAILED")


def test_before():
    global test_total, passed
    test_total += 1
    try:
        assert date2.before(date1)
        assert date1.before(date2) == False
        assert date1.before(date3)
        passed += 1
        print("before test PASSED")
    except AssertionError:
        print("before test FAILED")


def test_after():
    global test_total, passed
    test_total += 1
    try:
        assert date1.after(date2)
        assert date8.after(date7) == False
        assert date5.after(date2)
        passed += 1
        print("after test PASSED")
    except AssertionError:
        print("after test FAILED")


def test_equal():
    global test_total, passed
    test_total += 1
    try:
        assert date10.equal(date4)
        assert date5.equal(date6) == False
        passed += 1
        print("equal test PASSED")
    except AssertionError:
        print("equal test FAILED")


def test_add_days():
    global test_total, passed
    test_total += 1
    try:
        new_date = date1.add_days(365)
        assert new_date.day() == 27 and new_date.month() == 2 and new_date.year() == 2021
        new_date = date1.add_days(-365)
        assert new_date.day() == 28 and new_date.month() == 2 and new_date.year() == 2019
        new_date = date9.add_days(30)
        assert new_date.day() == 31 and new_date.month() == 3 and new_date.year() == 2004
        passed += 1
        print("add days test PASSED")
```

```
    except AssertionError:
        print("add days test FAILED")


def test_days_between():
    global test_total, passed
    test_total += 1
    try:
        assert date5.days_between(date5) == 0
        assert date1.days_between(date7) == 7250
        assert date1.days_between(date3) == 1
        assert date6.days_between(date5) == 4749
        passed += 1
        print("days between test PASSED")
    except AssertionError:
        print("days between test FAILED")


test_total = 0
passed = 0


test_day()
test_month()
test_year()
test_next()
test_previous()
test_before()
test_after()
test_equal()
test_add_days()
test_days_between()

print(passed, "out of", test_total, "tests passed for DateT")
print("")

gpost1 = GPosT(0, 0)
gpost2 = GPosT(1, 4)
gpost3 = GPosT(-90, 0)
gpost4 = GPosT(90, 0)
gpost5 = GPosT(-45.435, 67.54)
gpost6 = GPosT(-90, 180)
gpost7 = GPosT(90, -180)
gpost8 = GPosT(-89.54, -179.534)
gpost9 = GPosT(23.566, 5.77)
gpost10 = GPosT(65.23, -137.99)
gpost11 = GPosT(23.566, 5.77)
gpost12 = GPosT(0, 0.007)


test_total = 0
passed = 0

def isClose(test_ans, actual_ans):
    # Gives errors up to 0.3% Source: https://www.movable-type.co.uk/scripts/latlong.html
    return abs(actual_ans - test_ans) <= actual_ans * 0.003




def test_lat():
    global test_total, passed
    test_total += 1
    try:
        try:
            invalid_gpost = GPosT(-56,-181)
            print("invalid GPosT test FAILED")
        except ValueError:
            print("invalid GPosT test PASSED")
        assert gpost1.lat() == 0
        assert gpost2.lat() == 1
        assert gpost3.lat() == -90
        passed += 1
        print("latitude test PASSED")
    except AssertionError:
        print("latitude test FAILED")


def test_long():
    global test_total, passed
```

```
    test_total += 1
    try:
        assert gpost4.long() == 0
        assert gpost5.long() == 67.54
        assert gpost6.long() == 180
        assert gpost7.long() == -180
        passed += 1
        print("longitude test PASSED")
    except AssertionError:
        print("longitude test FAILED")


def test_west_of():
    global test_total, passed
    test_total += 1
    try:
        assert gpost7.west_of(gpost6)
        assert gpost8.west_of(gpost7) == False
        assert gpost3.west_of(gpost4) == False
        passed += 1
        print("west of test PASSED")
    except AssertionError:
        print("west of test FAILED")


def test_north_of():
    global test_total, passed
    test_total += 1
    try:
        assert gpost1.north_of(gpost2) == False
        assert gpost3.north_of(gpost7) == False
        assert gpost8.north_of(gpost6)
        passed += 1
        print("north of test PASSED")
    except AssertionError:
        print("north of test FAILED")


def test_equal():
    global test_total, passed
    test_total += 1
    try:
        assert gpost11.equal(gpost9)
        assert gpost1.equal(gpost2) == False
        assert gpost5.equal(gpost5)
        assert gpost1.equal(gpost12)
        passed += 1
        print("equal test PASSED")
    except AssertionError:
        print("equal test FAILED")


def test_move():
    global test_total, passed
    test_total += 1
    try:
        gpost1.move(145.435, 35)
        assert gpost1.equal(GPosT(-0.25916667, 0.17861111))
        gpost7.move(34.54, 577)
        assert gpost7.equal(GPosT(84.81083333, -90))
        gpost5.move(-43.523, -43.53)
        assert gpost5.equal(GPosT(-45.71833333, 67.92611111))
        passed += 1
        print("move test PASSED")
    except AssertionError:
        print("move test FAILED")


def test_distance():
    global test_total, passed
    test_total += 1
    try:
        gpost13 = GPosT(2.3, 43.5)
        assert isClose(gpost2.distance(gpost13), 4393)
        assert isClose(gpost3.distance(gpost10), 17260)
        assert isClose(gpost8.distance(gpost12), 10060)
        passed += 1
        print("distance test PASSED")
    except AssertionError:
        print("distance test FAILED")
```

```
def test_arrival_date():
    global test_total, passed
    test_total += 1
    try:
        gpost14 = GPosT(34.564, 100.5)
        gpost15 = GPosT(-65.9, -45.5)
        arrival_date = gpost14.arrival_date(gpost15, DateT(1, 1, 2020), 567.65)
        assert arrival_date.equal(DateT(29, 1, 2020))
        arrival_date = gpost3.arrival_date(gpost10, DateT(2, 5, 2018), 1234.54)
        assert arrival_date.equal(DateT(16, 5, 2018))
        arrival_date = gpost2.arrival_date(gpost12, DateT(31, 12, 2004), 600)
        assert arrival_date.equal(DateT(1, 1, 2005))
        arrival_date = gpost8.arrival_date(gpost11, DateT(20, 1, 2020), 10034.33)
        assert arrival_date.equal(DateT(22, 1, 2020))
        passed += 1
        print("arrival date test PASSED")
    except AssertionError:
        print("arrival date test FAILED")


test_lat()
test_long()
test_west_of()
test_north_of()
test_equal()
test_move()
test_distance()
test_arrival_date()

print(passed, "out of", test_total, "tests passed for GPosT")
```

H Code for Partner's date_adt.py

```
## @file date_adt.py
# @author Bruce He
# @brief ADT for DateT
# @date 2020/01/14

from datetime import *


# @brief create ADT for date related calculation
class DateT:

    ## @brief DateT constructor
    #  @details initialize DateT object with integers d, m, y as input
    #           If any invalid input of dates is given, a ValueError will shown
    #  @param datetime represents current date in datetime type
    #  @param d correspond to day
    #  @param m correspond to month
    #  @param y correspond to year
    #  @exception ValueError throws when inputs are not valid for datetime
    def __init__(self, d, m, y):
        try:
            self.__datetime = datetime(y, m, d)
        except:
            raise ValueError("Please input valid datetime.")
        self.__d = d
        self.__m = m
        self.__y = y

    ## @brief shows the value of current day
    #  @return the value of current day
    def day(self):
        return self.__d

    ## @brief shows the value of current month
    #  @return the value of current month
    def month(self):
        return self.__m

    ## @brief shows the value of current year
    #  @return the value of current year
    def year(self):
        return self.__y

    ## @brief shows the day after current date
    #  @detail First create a datetime object 'add1date' by adding self.__datetime and timedelta(1),
    #          which represents 1 day. Then extract the values of new day, month and year by using
     Class Attributes.
    #          Finally, return a DateT object with the new date.
    #  @param add1date adding one day in current date
    #  @return DateT object that is one day later than current date
    def next(self):
        add1date = self.__datetime + timedelta(1)
        self.__d = add1date.day
        self.__m = add1date.month
        self.__y = add1date.year

        return DateT(self.__d, self.__m, self.__y)

    ## @brief shows the day before current date
    #  @detail With a similar to the next() method, prev() method instead return one day before
     current date.
    #  @param minus1date showing one day earlier than current date
    #  @return DateT object that is one day earlier than current date
    def prev(self):
        minus1date = self.__datetime + timedelta(-1)
        self.__d = minus1date.day
        self.__m = minus1date.month
        self.__y = minus1date.year

        return DateT(self.__d, self.__m, self.__y)

    ## @brief determine if current date is before the target date d
    #  @detail Transfer target date d as datetime type. Then, use diff to store the difference in days
     between
    #          current date and target date, in value of days.Then, convert diff from timedelta to
     integer.
```

```
#           If diff is smaller than 0, current date is before target date, so return True. Return
    false otherwise.
#   @param d target date for comparison
#   @param temp_date represent target date
#   @param diff value of difference between current date to target date, measured in days.
#   @return True if current date is before target date, False otherwise.
def before(self, d):
    temp_date = datetime(d.year(), d.month(), d.day())  # Use day() to get value of day, instead
        of d.day
    diff = self.__datetime - temp_date
    diff = diff.days
    if diff < 0:
        return True
    else:
        return False

## @brief determine if current date is after the target date d
#   @detail Similar process as method before(self, d). This time, return true if diff is greater
    than 0, which means
#           current date is after target date. Return False otherwise.
#   @param d target date for comparison
#   @param temp_date represent target date
#   @param diff value of difference between current date to target date, measured in days.
#   @return True if current date is after target date, False otherwise.
def after(self, d):
    temp_date = datetime(d.year(), d.month(), d.day())
    diff = self.__datetime - temp_date
    diff = diff.days
    if diff > 0:
        return True
    else:
        return False

## @brief determine if current date is equal to the target date d
#   @detail Similar process as method after(self, d) and before (self, d). This time, return true
    if diff is 0, which
#           means current is the same as target date. Return False otherwise.
#   @param d target date for comparison
#   @param temp_date represent target date
#   @param diff value of difference between current date to target date, measured in days.
#   @return True if current date is the same as target date, False otherwise.
def equal(self, d):
    temp_date = datetime(d.year(), d.month(), d.day())
    diff = self.__datetime - temp_date
    diff = diff.days
    if diff == 0:
        return True
    else:
        return False

## @brief take integer n, return DateT that is n days later than current date
#   @detail Create a new datetime object new_date by adding current datetime with n days, in
    timedelta format.
#           I assume that n can either be positive or negative. If n is positive, that means we
    want a new date that
#           is n days after the current date. If n is negative, that means we want a new date that
    is n days earlier
#           than current date. If n is zero, new date is the same as current date.
#           Expect input n as integer with reasonable value.
#   @param n days be added on current date
#   @param new_date represents date to be shown, after calculation
#   @return DateT object, with n days added or subtracted to the current date
def add_days(self, n):
    new_date = self.__datetime + timedelta(n)
    return DateT(new_date.day, new_date.month, new_date.year)

## @brief take DateT object d, return the number of days between current date and date d
#   @detail I assume that the number of difference in days, between two dates, is always
    non-negative.
#           So no matter current date is before or after the date stored in d, the returning value
    is non-negative.
#           First, transfer d from DateT object to datetime type, in new_date.
#           Then, subtracting one date to another date, and change result from timedelta object to
    integer.
#           Return the integer that represents the day difference between current date and date d.
#   @param d DateT object used to compare with current date
#   @param new_date represents date stored in DateT object d
#   @return the number of days between current date and date stored in d
def days_between(self, d):
    new_date = datetime(d.year(), d.month(), d.day())
```

```
if self.__datetime >= new_date:
    return (self.__datetime - new_date).days
else:
    return (new_date - self.__datetime).days
```

I Code for Partner's pos_adt.py

```
## @file pos_adt.py
#  @author Bruce He
#  @brief module that implements and an ADT for global position coordinates and calculations around it
#  @date 2020/1/15

from math import *
from date_adt import *

# @@ brief create ADT for position coordinates related calculation
class GPosT:

    ## @brief GPosT constructor
    #  @detail initialized GPosT object with inputs latitude and longitude.
    #          This module expect users to input reasonable latitude in range of [-90, 90],
    #          and longitude in range of [-180,180].
    #  @param lat corresponds to the latitude, positive lat is North, negative lat is South
    #  @param long corresponds to the longitude, positive long is Ease, negative long is West
    #  @exception ValueError shows if latitude or longitude is out or range.
    def __init__(self, lat, long):
        if lat > 90 or lat < -90 or long > 180 or lat < -180:
            raise ValueError("Value Out of range")
        self.__lat = lat
        self.__long = long

    ## @brief getter for latitude
    # @return the value of latitude
    def lat(self):
        return self.__lat

    ## @brief getter for longitude
    #  @return the value of longitude
    def long(self):
        return self.__long

    ## @brief determine if current position is West of p
    #  @detail Compare the value of longitude of current position to GPostT p. If longitude of current
     position is smaller,
    #          then it is West of p, so return True. Return False otherwise.
    #          One thing worth noticing is: float lose precision when the difference is small.
    #  @return True if the current position is West of p; False otherwise
    def west_of(self, p):
        if self.long() < p.long():
            return True
        else:
            return False

    ## @brief determine if current position is North of p
    #  @detail Compare the value of latitude of current position to GPostT p. If latitude of current
     position is larger,
    #          then it is North of p, so return True. Return False otherwise.
    #          One thing worth noticing is: float lose precision when the difference is small.
    #  @param p GPosT object with latitude and longitude
    #  @return True if the current position is North of p; False otherwise
    def north_of(self, p):
        if self.lat() > p.lat():
            return True
        else:
            return False

    ## @brief determine the distance between current position and argument p(in km)
    #  @detail Followed by the instruction, 'haversine' formula is used directly
    #          to calculate the distance between two points.
    #  @param p GPosT object with latitude and longitude
    #  @param radius Earth's mean radius
    #  @param lat1 latitude of current position in radians
    #  @param lat2 latitude of position p in radians
    #  @param lat_delta difference of latitude between current position and position p, in radians
    #  @param long_delta difference of longitude between current position and position p, in radians
    #  @param a square of half the chord length between 2 points
    #  @param c the angular distance in radians
    #  @return the distance between current position and p with unit of km
    def distance(self, p):
        radius = 6371
        lat1 = radians(self.lat())
        lat2 = radians(p.lat())
        lat_delta = lat1 - lat2
```

```
    long_delta = radians(self.long()) - radians(p.long())
    a = sin(lat_delta/2)**2 + cos(lat1) * cos(lat2) * sin(long_delta/2)**2
    c = 2 * atan2(sqrt(a), sqrt(1-a))
    return radius * c

## @brief determine whether current position is the same as position p
#  @detail use self.distance(p) to get value of distance between current position and position p.
#          Followed by instruction, if the value is less than 1, that means two positions are
    considered equal.
#  @param p GPosT object with latitude and longitude
#  @return True if 2 points are within 1 km; False otherwise.
def equal(self, p):
    if self.distance(p) <= 1:
        return True
    else:
        return False

## @brief change current position with bearing b and distance d
#  @detail With the formula provide in https://www.movable-type.co.uk/scripts/latlong.html,
#          use current position, bearing and distance to calculate the moved position
#  @param b the value of bearing
#  @param d distance moved in unit of km
#  @param ang angular distance, calculated by d/r; d is distance moved, r is Earth's mean radius
#  @param rad_lat latitude of current position in radians
#  @param rad_long longitude of current position in radians
#  @param new_lat latitude of moved position in radians
#  @param new_long longitude of moved position in radians
def move(self, b, d):
    ang = d/6371
    bearing = radians(b)
    rad_lat = radians(self.lat())
    rad_long = radians(self.long())
    new_lat = asin(sin(rad_lat) * cos(ang) + cos(rad_lat) * sin(ang) * cos(bearing))
    new_long = rad_long + atan2(sin(bearing) * sin(ang) * cos(rad_lat), cos(ang) - sin(rad_lat) *
        sin(new_lat))
    self.__lat = degrees(new_lat)     # update the latitude in degree type
    self.__long = degrees(new_long)  # update the longitude in degree type

## @brief return DateT object that shows arrival date
#  @detail start at date d, moving from current position to position p at a speed s.
#          Since DateT.add_days(n) will round off to 1 days if n = 1.9, so the day used for moving
    from current position
#          to point p will round up. If n = 1.9, the actual day used is 2 days.
#  @param p target position in GPosT type
#  @param s speed with units km/day
#  @param d starting date in DateT type
#  @param distance the distance between current position and position p, in unit of km
#  @param day_used day used to finish the trip with speed s, rounding up
#  @return the arrival date
def arrival_date(self, p, d, s):
    distance = self.distance(p)
    day_used = ceil(distance/s)
    return d.add_days(day_used)
```