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

- Speed and distance (for the arrival date and move function) are measured in terms of km and hours, and may be negative (indicating 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

Most of the errors in the DateT module were simple things like wrong return types and mispelled variable names. However, for GPosT, it took me a while to understand that my calculations weren't working because I had forgotten to convert my units to radians. After the fixes however, everything ran smoothly. Here's a summary of the pytest log for the test driver run on my code:

```
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 I detail each failure and provide and explanation as to why the unit test failed.

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 that negative numbers for days was allowed in the input (which would indicate moving backwards in time), while my partner assumed that negative days were not allowed. Instead he simply returns a ValueError.

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 object's date, 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 would make his function return a date that was almost always one day ahead of mine.

3 Critique of Given Design Specification

3.1 Disadvantages

The design specification was not very complete, leaving a lot of room for assumptions and ambiguity. The first module DateT didn't even specify what type of calendar to use, which meant I could've used any calendar that had a notion of days, months and years and it would've been a valid implementation. Similarly, I had to make a lot of assumptions about what to do in certain situations when implementing certain functions. 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.

3.2 Advantages

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 could approach the problem. It also puts the focus on the module interface rather than biasing how we should implement said interface.

3.3 Improvements

One simple improvements for the design would be to be more clear and specific. What type of calendar are we implementing? Are negative distances allowed? What is the range on this input value? These are all questions that should be answered by the design specification.

4 Answers to Questions

(a) For the DateT module, you could store the date using state variables for year, month, and day. This is the most trivial implementation as that's how the interface for the constructor is defined. However, you could also implement the DateT module using a single state variable for the number of days, as both month and year can be expressed in terms of days. In this version of the implementation, you would need some anchor date for which the number of days are expressed relative to this anchor date. For example, if the anchor date $days = 1$ is Jan 1st, 2000, then Jan 1st, 1999 would have the state variable $days = -366$, and Jan 1st, 2001 would have the state variable $days = 377$.
For the GPosT module, you could store the longitude and latitude in terms of radians or in terms of decimal degrees. Both have their advantages and disadvanteges. For example, storing it in terms of radians means you don't have to convert it everytime you want to perform a computation, but it also means you'll have to convert it to degrees anytime the user wants to access longitude or latitude. Alternatively, you could also use the DMS (degrees, minutes, seconds) representation for position.

(b) GPosT is mutable because the inner state variables are edited when you call the move function. In contrast, DateT is immutable because when you call prev, next, and add days, it returns a new DateT object rather than editing the current one. With the given interface, there would be no way to edit a DateT object's value after it's been initialized.

(c) Unit testing via pytest abstracted some of the boiler plate code I had to create when I wasn't using pytest (which you can see via my commit history). It also encourages you to organize your test cases into isolated functions. This was extremely helpful for me to visually understand how rigorously each function was being tested and where I needed to add more test cases/more elaborate test cases. Finally, since it provides a traceback of the values as an assertion is evaluated, you don't need to manually break down a test case with print statements, which saved me a ton of time.

(d) In 2015, twitter failed for a few hours due to a date/time issue in their code. They were using the Gregorian calendar for the day and month, but accidentally used the ISO 8601 week-based system for the year. Because of how each of these calendars work, the Gregorian and ISO years lined up perfectly until 2015, at which point there was a 2 day mismatch and twitter was recording the wrong date on their platform. Although the impact of this bug was minimal, you can imagine if a similar bug existed for say a record keeping system for digital evidence for criminal cases, that vital evidence could be considered void because the date of said evidence would be under dispute. Software quality and high cost is a major challenge because of the project management triangle. To summarize, it states that any software project cannot be cheap, high quality, and fast to develop at the same time. In a business oriented world, with deadlines and a drive to maximize profits and reduce costs, it's easy to see why cost and speed are prioritized over quality. Ideally, you would be given more time and financial resources to create and test a quality product. (Cite: `https://www.youtube.com/watch?v=D3jxx8Yyw1c`)

(e) The ration design process is a waterall design process for software development with 7 steps: problem statement, development plan, requirements, design documentation, code, and verification. However this process is not realistic. Software development is almost always never this linear, as along the process you'll learn you may need to relook at the problem statement, or update the requirements as you're in the later stages of the process. As humans, we also make errors and have imperfect communication, which means we need to continually reevaluate each step of the process. Hence, we "fake" the design process.

(f) Let's start by defining each term (Cite: `https://accu.org/index.php/journals/1585`).

- Correctness: The ability of software products to perform their exact tasks, as defined by their specification.
- Robustness: The ability of software systems to react appropriately to abnormal conditions.
- Reliability: A concern encompassing correctness and robustness. The ability of a system to perform its requested functions under stated conditions whenever required - having a long mean time between failures.

To understand how these three are distinguished better, let's take the GPosT and DateT moduels as examples. Correctness would describe how the software's ability to return the correct output and behaviour that was specified. When I call the next method does the date change by one day? Does it work for edge cases, say when the month and year change? What about leap years? Robustness would be the ability of the software to work when invalid inputs are provided, or when errors occur. For example, although my implementation for arrive date was correct in GPosT, it was not robust, as I didn't stop the user from inputing 0 for the speed, causing a DivisionByZero error. Finally, reliability describes both of these terms in general (ie how often does the software do what it is intended to do).

(g) Modularity is the idea of seperating a software systems componenets into organized parts. This can be done for a variety of reasons. Readability, flexibility, maintenance, communication, collaboration, and for seperation of concerns. Seperation of concerns is a software design principle for seperating componenets into independent parts. To understand this better, let's take GPosT as an example. The arrival date function uses the distance function in it's implementation. This modularity allows us to seperate the problem of finding out the arrival date, and calculating distance. Here, we are free to edit the distance function (maybe using a different approximation formula) without breaking the code.

E Code for date_adt.py

```
## @file date_adt.py
#  @author Jay Mody
#  @brief Provides the DateT ADT class for representing dates.
#  @date 20/01/20 (dd/mm/yy)

import datetime

## @brief An ADT that represents a date.
#  @details An ADT for an idealized naive date, assuming the current Gregorian calendar always was,
    and always will be, in effect.
class DateT:

    ## @brief Constructor for DateT objects.
    #  @param d The day of the month (integer from 1-31)
    #  @param m The month of the year (integer from 1-12)
    #  @param y The year (integer from 1-9999)
    def __init__(self, d, m, y):
        self.__date = datetime.date(y, m, d)

    ## @brief Gets the day of the month.
    #  @return The day of the month.
    def day(self):
        return self.__date.day

    ## @brief Gets the month of the year.
    #  @return The month of the year.
    def month(self):
        return self.__date.month

    ## @brief Gets the year.
    #  @return The year.
    def year(self):
        return self.__date.year

    ## @brief Returns a date that is 1 day ahead.
    #  @return A DateT object that is 1 day ahead.
    def next(self):
        new_date = self.__date + datetime.timedelta(days=1)
        return DateT(new_date.day, new_date.month, new_date.year)

    ## @brief Returns a date that is 1 day behind.
    #  @return A DateT object that is 1 day behind.
    def prev(self):
        new_date = self.__date - datetime.timedelta(days=1)
        return DateT(new_date.day, new_date.month, new_date.year)

    ## @brief Determines if this date comes before date d.
    #  @param d A DateT object.
    #  @return A boolean that is True if this date comes before date d, else False.
    def before(self, d):
        return self.__date < d.__date

    ## @brief Determines if this date comes after date d.
    #  @param d A DateT object.
    #  @return A boolean that is True if this date comes after date d, else False.
    def after(self, d):
        return self.__date > d.__date

    ## @brief Determines if this date and date d are equal.
    #  @param d A DateT object.
    #  @return A boolean that is True if this date and date d are equal, else False.
    def equal(self, d):
        return self.__date == d.__date

    ## @brief Returns a date that is n days ahead.
    #  @param n An integer representing the number of days to skip ahead.
    #  @return A date that is n days ahead.
    def add_days(self, n):
        new_date = self.__date + datetime.timedelta(days=n)
        return DateT(new_date.day, new_date.month, new_date.year)

    ## @brief Returns the number of days between this day and date d.
    #  @param d A DateT object.
    #  @return The number of days between this day and date d (negative if d comes before this date).
    def days_between(self, d):
        return (d.__date - self.__date).days
```

F Code for pos_adt.py

```
## @file pos_adt.py
#  @author Jay Mody
#  @brief Provides the GPosT ADT class for representing latitude/longitude points on Earth.
#  @date 20/01/20 (dd/mm/yy)

from math import cos, sin, asin, atan2, radians, degrees
from date_adt import DateT

## @brief An ADT that represents latitude/longitude positions.
#  @details An ADT for signed decimal degree latiitude and longitude GPS positions on Earth, assuming
      Earth's radius to be 6371km.
class GPosT:
    ## Earth's radius in km
    __R = 6371

    ## @brief Constructor for GPosT objects.
    #  @param lat Latitude as a signed decimal degree (float from -90 to 90), with + as north and - as
        south.
    #  @param long Longitude as a signed decimal degree (float from -180 to 180), with + as east and -
        as west.
    #  @throws ValueError Thrown if longitude or latitude values are not in the correct ranges.
    def __init__(self, lat, long):
        if not (-180 <= long and long <= 180):
            raise ValueError("long (longitude) must be between -180 and 180")
        if not (-90 <= lat and lat <= 90):
            raise ValueError("lat (latitude) must be between -90 and 90")

        self.__lat = radians(lat)
        self.__long = radians(long)

    ## @brief Get's the latitude (as a signed decimal degree).
    #  @return The latitude.
    def lat(self):
        return degrees(self.__lat)

    ## @brief Get's the longitude (as a signed decimal degree).
    #  @return The longitude.
    def long(self):
        return degrees(self.__long)

    ## @brief Determins if this position is west of position p.
    #  @param p A GPosT object.
    #  @return  A boolean that is True if this position is west of p, else False.
    def west_of(self, p):
        return self.__long < p.__long

    ## @brief Determines if this position is north of position p.
    #  @param p A GPosT object.
    #  @return  A boolean that is True if this position is north of p, else False.
    def north_of(self, p):
        return self.__lat > p.__lat

    ## @brief Determines if this position equal (within 1km distance) to position p.
    #  @param p A GPosT object.
    #  @return  A boolean that is True if this position is equal to p, else False.
    def equal(self, p):
        distance = self.distance(p)
        return distance < 1.0

    ## @brief Moves the current position by d distance at b bearing.
    #  @param b The bearing of the move, as a signed decimal degree.
    #  @param d The distance (in km) to move.
    def move(self, b, d):
        b = radians(b)

        angular_dist = d / self.__R
        target_lat = asin(sin(self.__lat) * cos(angular_dist) + cos(self.__lat) * sin(angular_dist) *
             cos(b))

        y = sin(b) * sin(angular_dist) * cos(self.__lat)
        x = cos(angular_dist) - sin(self.__lat) * sin(target_lat)
        target_long = self.__long + atan2(y, x)

        self.__lat = target_lat
        self.__long = target_long
```

```python
## @brief Gets the distance (in km) between this position and position p.
#  @return The distance (in km)
def distance(self, p):
    delta_lat = p.__lat - self.__lat
    delta_long = p.__long - self.__long

    a = sin(0.5 * delta_lat)**2 + cos(self.__lat) * cos(p.__lat) * sin(0.5 * delta_long)**2
    c = 2 * atan2(a**0.5, (1-a)**0.5)
    distance = self.__R * c
    return distance

## @brief Calculates the arrival date to get to position p from this position, given a start date
#    and speed.
#  @param p The target position (as a GPosT object).
#  @param d The start date (as a DateT object).
#  @param s The speed (in km/day).
#  @return The arrival date (as a DateT object).
def arrival_date(self, p, d, s):
    distance = self.distance(p)
    days = distance / s
    return d.add_days(n=days)
```

G Code for test_driver.py

```
## @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
```

```python
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))
```

```python
    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)
```