

Assignment 1 Solution

Shazil Arif, 400201970

Jan 26th, 2020

Intro blurb ??

1 Testing of the Original Program

As someone who is experienced with unit testing, the motivation behind my test driver file was very similar to that of a real testing framework. The idea was to have a single function (called "compare") that compares an expected to an actual value; if they match, the test passes otherwise it fails. For visual purposes the output to the terminal is "Passed" highlighted in green if a test passes, and "Failed" highlighted in red if a test fails. This removes the need for a user to sit through and read which tests failed/passed. The compare function also checks if the expected/actual values are objects (instances of GPosT or DateT). If they are objects it compares all their state variables and a test passes if all the compared values are the same. The program keeps a count of how many tests are executed and how many fail. Another important design choice was to let the program continue its execution even if a test fails, this way all tests will be executed once and the result for each can be seen.

The rationale behind the test cases was to cover all/majority of possible execution paths given the assumptions and design choices in the actual code files. For example, when testing the next() function for DateT, I wrote tests that would cover several test cases including when the function: returns the next day in the current month, returns first day of the next month, returns the first day of the next year, tests the month of february for leap years and so on. I used this same approach and considered the many different possibilities when writing all the tests.

Initially, some tests failed and some passed. Writing good test cases that covered all execution paths helped me find some flaws in my code. For example, I realized in the arrivaldate() function I did not account for when a 0 speed would be passed as a parameter. I also discovered other minor issues such as returning the current day minus

one, instead of one for the next() function. While testing, I also realized the class state variables were not private which is not ideal, so I decided to encapsulate these fields.

2 Results of Testing Partner's Code

After running my partner's code with my test file, 6 out of 44 test cases failed. The methods that failed were:

- The distance() method for GPosT. My test driver expected 571.44 km because I decided to round distance values to 2 decimal places. My partner's returned 571.44064.. and several other decimals. The test failed due to different choices when it came to rounding distance values.
- The move() method for GPosT. Same reason as above. Due to different choices, regarding how many decimal places to keep, the expected and actual were different. The non-decimal portion is correct however.
- The arrival _date() method. My test driver expected the 19th of January as the returned date, partner's code returned January 20th, 2020. This is due to different design choices. In my code I decided to floor (round down) any decimal values when calculating the number of days required. My partner decided to ceil (round up) decimal values, hence the difference of one day in the expected and actual answer. My code for calculating the number of days required :

```
if (s!=0): num_days = Math.floor(distance/round(s,2))
```

Partner's code for calculating number of days required:

```
day_used = ceil(distance/s)
```

- The arrival _date() again failed for a different set of parameters. It failed for the same reason as above. Due to different choices for handling decimal values for arrival _date().
- The arrival _date() a third time for another set of parameters. It failed for the same reason as above. Due to different design choices for handling decimal values.
- The arrival _date() with a value of 0 for speed. The test failed and gave a division by zero error, it failed because partner's code did not account for a value of 0 and it gave a division by zero.

3 Critique of Given Design Specification

A major advantage of the design specification was the open-endedness. The modules and functions to be implemented were specified with their parameters, return values and a description of what it should do. How the output is achieved is not defined, this gives the developer the ability to implement the required functionality however they want. Whether it's deciding which formula to use for a calculation or whether to use an existing library and tailor it to the assignment requirements, there was a lot of freedom.

Another part of the design specification t

4 Answers to Questions

- (a) One possibility for the state variables is simply having three variables for the day, month and year for DateT and similarly two variables for latitude/longitude in GPosT. Another possibility is using an iterable type such as a list or tuple that contain the latitude/longitude values for GPosT and day, month, year for DateT.
- (b) The interface specified in this assignment does not state the use of any state variables or how to maintain them. This means the state variables can be left private or public. Depending on whether the state variables were made public or private, their values could be mutated. However, in the GPosT module, the move() method modifies the values of the state variables. Although not directly, GPosT objects are then indirectly mutable. There are no setter methods.
- (c) pytest is an automated unit testing framework. It provides several features including:
 - (1) Detailed debugging info on assert statements if a test fails
 - (2) Over 315+ plugins for third party python packages, allowing users to unit test a variety of different applications. See <http://plugincompat.herokuapp.com/> for a list.
 - (3)

E Code for date_adt.py

```
## @file date_adt.py
# @author Shazil Arif
# @brief date_adt.py contains a Class that implements a Date object containing a year, month and day
# @date Jan 20th, 2020

from datetime import datetime
from datetime import timedelta

## @brief DateT is a class that implements a Date object containing a year, month and a day
class DateT:

    ## Represents the months with 31 days
    __odd_month = [1,3,5,7,8,10,12]

    ## enum for representing the maximum number of days in the months contained in odd_month
    __max_odd_month = 31

    ## Represents months with 30 days
    __even_month = [4,6,9,11]

    ## enum for representing the maximum number of days in the months contained in even_month
    __max_even_month = 30

    ## enum for the 12 month
    __december = 12

    ## enum for the month of february
    __february = 2

    ## enum for the first month
    __january = 1

    ## enum for number of days in a leap year in the month of february
    __leap_year_days = 29

    ## enum for number of days in february in a common year (not a leap year)
    __feb_common_days = 28

    ## @brief the constructor method for class DateT
    # @param d The date to be set. Assumes an integer between 1 to 31 (inclusive)
    # @param m the Month to be set. Assumes an integer from 1 to 12 (inclusive)
    # @param y the Year to be set. Assumes a positive integer
    def __init__(self, d, m, y):
        self.__d = d
        self.__m = m
        self.__y = y

    ## @brief returns the day
    # @return the day
    def day(self):
        return self.__d

    ## @brief returns the month
    # @return the month
    def month(self):
        return self.__m

    ## @brief returns the year
    # @return the year
    def year(self):
        return self.__y

    ## @brief Returns a DateT object that is 1 day later than the current object
    # @return DateT object that is set 1 day later
    def next(self):
        #going into new month when current month has 31 days
        if(self.month() in self.__odd_month and self.day() + 1 > self.__max_odd_month and self.month() !=
            self.__december):
            return DateT(1, self.month() + 1, self.year())

        #going into new month when current month has 30 days
        if(self.month() in self.__even_month and self.day() + 1 > self.__max_even_month):
            return DateT(1, self.month() + 1, self.year())

        #going into the new year
        if(self.day() + 1 > self.__max_odd_month and self.month() == self.__december):
```

```

        return DateT(1, 1, self.year() + 1)

# if current month is february
if(self.month() == self.__february):
    #leap year and transitioning into march
    if(self.__is_leap_year() and self.day() + 1 > self.__leap_year_days):
        return DateT(1, self.month() + 1, self.year() )

    #not a leap year but transitioning into march
    elif(not self.__is_leap_year() and self.day() + 1 > self.__feb_common_days):
        return DateT(1, self.month() + 1, self.year())

#otherwise return the next day in the current month and year
return DateT(self.day() + 1, self.month(), self.year())

## @brief returns a DateT object that is 1 day before the current object
# @return DateT object that is set 1 day before
def prev(self):
    #in the case where we go back to the previous month
    if(self.day() - 1 < 1 and self.month() != self.__january):

        #if previous month is not february
        if(self.month() - 1 != self.__february):
            #check if previous month has 31 days
            if(self.month() - 1 in self.__odd_months):
                return DateT(self.__max_odd_month, self.month() - 1, self.year())

            #previous month has 30 days
            if(self.month() - 1 in self.__even_months):
                return DateT(self.__max_even_month, self.month() - 1, self.year())

        #in the case where previous month is february
        #first check if leap year or not
        if(self.__is_leap_year()):
            return DateT(self.__leap_year_days, self.__february, self.year())
        return DateT(self.__feb_common_days, self.__february, self.year())

    #in the case we have to go back to the previous year
    if(self.day() - 1 < 1 and self.month() == self.__january):
        return DateT(self.__max_odd_month, self.__december, self.year() - 1)

    #the simplest case, where there is no month or year transition
    return DateT(self.day() - 1, self.month(), self.year())

## @brief compares if the date represented by the current DateT object is before d (d is also a
    DateT object)
# @param d The DateT object to compare with the current object. Assumes a valid DateT object
# @return A boolean value indicating whether the current objects date is before the date in d (True
    if before, False otherwise)
def before(self, d):
    if(self.year() < d.year()): return True
    if(self.year() == d.year() and self.month() < d.month()): return True
    if(self.year() == d.year() and self.month() == d.month() and self.day() < d.day()): return True
    return False

## @brief compares if the date represented by the current DateT object is after d (d is also a DateT
    object)
# @param d The DateT object to compare with the current object. Assumes a valid DateT object
# @return A boolean value indicating whether the current objects date is after the date in d (True
    if before, False otherwise)
def after(self, d):
    if not self.before(d): return True
    return False

## @brief compares if the current DateT object and another DateT object d represent the same date
# @param d The DateT object to compare with the current object. Assumes a valid DateT object
# @return A boolean value indicating whether the two objects represent the same data (True if
    equal, False otherwise)
def equal(self, d):
    return self.__dict__ == d.__dict__

## @brief adds n days to the date represented by the current DateT object
# @param n The number of days to add. Assumes an integer greater than or equal to 0
# @return A DateT object with its date set n days later than the original
def add_days(self, n):
    temp = datetime(self.year(), self.month(), self.day())
    temp = temp + timedelta(days=n)
    return DateT(temp.day, temp.month, temp.year)

```

```

## @brief calculates the number of days between the current DateT object and DateT object d
# @param d The DateT object to calculate the number of days in between with. Assumes a valid DateT
object
# @return An integer value indicating the number of days between the two DateT objects
def days_between(self,d):
    date_one = datetime(self.year(), self.month(), self.day())
    date_two = datetime(d.year(),d.month(),d.day())
    difference = date_one - date_two
    return abs(difference.days)

## @brief returns whether or not the year in the current DateT object is a leap year
# @details private method, not accessible from external interface
# @return a boolean value indicating whether or not the year is a leap year (True if leap year,
False otherwise)
def __is_leap_year(self):
    if(self.year() % 400 == 0): return True
    if(self.year() % 100 == 0): return False
    if(self.year() % 4 == 0): return True
    return False

```

F Code for pos_adt.py

```
## @file pos_adt.py
# @author Shazil Arif
# @brief pos_adt.py implements a class for global position coordinates
# @date January 20th, 2020

import math as Math
import date_adt as Date
## @brief GPosT is class that implements an ADT to represent coordinates using longitude and latitude values
class GPosT:
    ## @brief the constructor method for class GPosT
    # @details Assumes a valid latitude and longitude value as signed decimal degrees
    # @param phi The latitude to be set for the GPosT object
    # @param _lambda the longitude value to be set for the GPosT object
    def __init__(self, phi, _lambda):
        self.__latitude = phi
        self.__longitude = _lambda

    ## @brief returns the latitude for the current GPosT object
    # @return the latitude value
    def lat(self):
        return self.__latitude

    ## @brief returns the longitude for the current GPosT object
    # @return the longitude value
    def long(self):
        return self.__longitude

    ## @brief returns whether the coordinates of the current GPosT object are west of those in object p
    # @param p the GPosT object to compare. Assumes a valid GPosT object p as a parameter
    # @return a boolean value indicating whether the current objects coordinates are west of p (True if they are west of p, False otherwise)
    def west_of(self, p):
        return self.long() < p.long()

    ## @brief returns whether the coordinates of the current GPosT object are north of those in object p
    # @param p the GPosT object to compare. Assumes a valid GPosT object as a parameter
    # @return a boolean value indicating whether the current objects coordinates are north of coordinates in p (True if they are west of p, False otherwise)
    def north_of(self, p):
        return self.lat() > p.lat()

    ## @brief returns whether the current GPosT object and a GPosT object p represent the same position
    # @details considered to represent the same location if the distance between their coordinates is less than 1 km
    # @param p the GPosT object to compare against. Assumes a valid GPosT object p as a parameter
    # @return a boolean value indicating whether the two objects represent same location(i.e if their distance is less than 1km). True if same location, False otherwise
    def equal(self, p):
        return self.distance(p) < 1

    ## @brief moves the position represented by the current GPosT object in direction of bearing b with total distance d
    # @param b A real number indicating the bearing/direction to move in. Assumes a valid number
    # @param d A real number indicating the distance to move in units of kilometres (km). Assumes a valid number
    def move(self, b, d):
        radius = 6371

        phi_one = Math.radians(self.lat())
        angular_dist = d/radius

        new_lat = Math.asin(Math.sin(phi_one) * Math.cos(angular_dist) + Math.cos(phi_one)*Math.sin(angular_dist)*Math.cos(Math.radians(b)) )
        new_long = self.long() + Math.degrees(Math.atan2(Math.sin(Math.radians(b))*Math.sin(angular_dist)*Math.cos(phi_one), Math.cos(angular_dist) - Math.sin(phi_one) * Math.sin(new_lat) ))

        self.__latitude = round(Math.degrees(new_lat),2)
        self.__longitude = round(new_long,2)

    ## @brief calculates the distance between the positions represented by current GPosT object and another GPosT object 'p'. Calculates to 2 decimal places
    # @details Applies the spherical law of cosines formula to calculate the distance. See https://www.movable-type.co.uk/scripts/latlong.html under the heading 'Spherical Law of
```

```

    Cosines'
# @param p A GPosT object containing the lat/long coordinates to calculate the distance to.
# Assumes a valid GPosT object p as a parameter
# @return an integer value representing the distance between the current object and p in units of
    kilometres(km)
def distance(self,p):
    #earth's approximate radius in kilometres
    radius = 6371

    lat_one = Math.radians(self.lat())
    lat_two = Math.radians(p.lat())

    long_diff = Math.radians(p.long() - self.long())

    distance = Math.acos(Math.sin(lat_one)*Math.sin(lat_two) +
        Math.cos(lat_one)*Math.cos(lat_two)*Math.cos(long_diff)) * radius

    return round(distance,2)

## @brief calculates the number of days required to travel from the position represented by
current GPost object to another position represented by a GPost object while travelling at a
specific speed and starting on a specific day
# @details please note that the speed parameter will be rounded to two decimal places to keep it
consistent with distance values. If the speed is 0 the current DateT object will be returned
# @param p A GPosT object representing the position to travel to. Assumes a valid GPosT object p
as a parameter
# @param d a DateT object representing the date to begin travelling on . Assumes a valid DateT
object d as a parameter
# @param s A real number indicating the speed to travel at in units of km/day. It is assumed to
be positive real number
# @return an integer value representing the distance between the current object and p in units of
    kilometres(km)
def arrival_date(self,p,d,s):
    distance = self.distance(p)

    #number of days required to cover the distance travelling at speed s
    num_days = 0
    if(s!=0): num_days = Math.floor(distance/round(s,2))

    return d.add.days(num_days)

```


G Code for test_driver.py

```
## @file test_driver.py
# @author Shazil Arif
# @brief this test driver module is used to test modules DateT and GPost
# @date January 20th, 2020
from date_adt import DateT
from pos_adt import GPosT
import time

failed = []
count = 0

def compare(description, expected, actual):
    print("Description: {description}\n".format(description=description))
    global count
    count = count + 1

    #if expected value is instance of DateT or GPost class
    if(isinstance(expected, DateT) or isinstance(expected, GPosT)):
        expected_keys = expected.__dict__
        actual_keys = actual.__dict__
        print("Expected properties")
        for i in expected_keys:
            print("{key} : {value}".format(key=i, value=expected_keys[i]))
        print("\nActual properties")
        for i in actual_keys:
            print("{key} : {value}".format(key=i, value=actual_keys[i]))
        if(expected.__dict__ == actual.__dict__):
            print('\nResult: ' + '\x1b[6;30;42m' + 'Passed' + '\x1b[0m') #source:
                                https://stackoverflow.com/questions/287871/how-to-print-colored-text-in-terminal-in-python
        else:
            failed.append({"Description": description,
                           "Expected": expected,
                           "Actual": actual})
            print('\nResult: ' + '\x1b[1;37;41m' + 'Failed' + '\x1b[0m')

    else: #comparing other types...string, int, float etc.
        print("Expected: {expected}".format(expected=expected))
        print("Actual: {actual}".format(actual=actual))
        if(expected == actual): print('\nResult: ' + '\x1b[6;30;42m' + 'Passed' + '\x1b[0m')
        else:
            failed.append({"Description": description, "Expected": expected, "Actual": actual})
            print('\nResult: ' + '\x1b[1;37;41m' + 'Failed' + '\x1b[0m')
    print("\n-----\n")

def test_date_adt():
    #testing date_adt.py

    #2020 is a leap year!
    start = time.time()
    test = DateT(1,1,2020)

    #test getter
    compare("testing getter method for day", 1, test.day())
    compare("testing getter method for month", 1, test.month())
    compare("testing getter method for year", 2020, test.year())

    #test next function
    #ideally for a functions like this the number of tests to run should be equal to or greater than
    #the number of execution paths
    #there are 6 cases
    # i) simply the next day within current month and year
    # ii) Transition into The next month where the current month has 30 days
    # iii) Transition into The next month where the current month has 31 days
    # iv) Transition into The next year
    # v) Transition into the next month when the current month is february and it a leap year
    # vi) Transition into the next month when current month is february and it is not a leap year

    compare("testing next method, it should return January 2nd 2020 and
            pass", DateT(2,1,2020), test.next())

    test = DateT(31,1,2020)
    compare("test for transitioning into next month with current month having 31 days. It should
            return february 1st 2020", DateT(1,2,2020), test.next())

    test = DateT(30,4,2020) #April 30th, 2020
```

```

compare("test for transitioning into next month with current month having 30 days. It should
return May 1st 2020",DateT(1,5,2020),test.next())

test = DateT(28,2,2020)
compare("test for transitioning into next month with current month being february and the year is
a leap year. It should return Feb 29th 2020",DateT(29,2,2020),test.next())

test = DateT(28,2,2021)
compare("test for transitioning into next month with current month being february and the year is
NOT leap year. It should return March 1st 2021",DateT(1,3,2021),test.next())

test = DateT(31,12,2020)
compare("test for transitioning into next year. It should return Jan 1st 2021 and
pass",DateT(1,1,2021),test.next())

#test prev method
test=DateT(2,1,2020)
compare("test for prev method, it should return January 1st 2020 and
pass",DateT(1,1,2020),test.prev())

test=DateT(1,5,2020)
compare("test for transitioning into previous month with current month having 31 days. It should
return April 30th 2020",DateT(30,4,2020),test.prev())

test=DateT(1,6,2020)
compare("test for transitioning into previous month with current month having 30 days. It should
return May 31st 2020",DateT(31,5,2020),test.prev())

test=DateT(1,3,2020)
compare("test for transitioning back into february and the year is a leap year. It should return
Feb 29th 2020",DateT(29,2,2020),test.prev())

test=DateT(1,3,2021)
compare("test for transitioning back into february and the year is NOT leap year. It should return
Feb 28th 2021",DateT(28,2,2021),test.prev())

test=DateT(1,1,2020)
compare("test for transitioning into previous year. It should return Dec 31st 2019 and
pass",DateT(31,12,2019),test.prev())

#test for before method
test = DateT(1,1,2020)
test2 = DateT(1,5,2020)
compare("test for before method , it should return True and pass",True,test.before(test2))
compare("test for before method , it should return False and pass",False,test2.before(test))

#test for after method
compare("test for after method , it should return True and pass",True,test2.after(test))
compare("test for after method , it should return False and pass",False,test.after(test2))

#test equals method
test = DateT(1,1,2020)
test2 = DateT(1,1,2020)
test3 = DateT(1,2,2020)
compare("test for equals method, it should return True and pass",True,test.equal(test2))
compare("test for equals method, it should return False and pass",False,test.equal(test3))

#test add_days method
test = DateT(31,1,2020)
compare("test add days method, it should return Feb 1st 2020 and
pass",DateT(1,2,2020),test.add_days(1))
compare("test add days method, it should return Feb 29, 2020", DateT(29,2,2020),test.add_days(29))

test = DateT(31,1,2021)
compare("test add days method, it should return March 1st, 2021",
DateT(1,3,2021),test.add_days(29))

test = DateT(1,1,2021)
compare("test add days method, add 365 days when current year is NOT leap year, it should return
january 1st 2022",DateT(1,1,2022),test.add_days(365))

test = DateT(1,1,2020)
compare("test add days method, add 365 days when current year IS LEAP YEAR. it should return Dec
31st, 2020",DateT(31,12,2020),test.add_days(365))

test = DateT(1,1,2020)
compare("test add days method, add 366 days when current year IS LEAP YEAR. it should return Jan
1st, 2021",DateT(1,1,2021),test.add_days(366))

#test days_between method

```

```

test = DateT(31,1,2020)
test2 = DateT(1,3,2020)

compare("test days_between method with March and January when current year is leap year, it should
return 30 days",30,test2.days_between(test))

test = DateT(31,1,2021)
test2 = DateT(1,3,2021)
compare("test days_between method with March and January when current year is NOT leap year, it
should return 29 days",29,test2.days_between(test))

def test_post_adt():
    test = GPosT(45,45)
    compare("test getter method for latitude",45,test.lat())
    compare("test getter method for longitude",45,test.long())

    test = GPosT(43.580605, -79.625668)
    test2 = GPosT(40.723606, -73.860514)
    compare("test distance method, it should return 571.44 km rounded to the 2 decimal
places",571.44,test2.distance(test))

    test = GPosT(43.261897, -79.921433)
    test2 = GPosT(43.262545, -79.922549)
    compare("test equal method for distance < 1 km, it should return True",True, test2.equal(test))

    test2 = GPosT(43.250880, -79.920292)
    compare("test equal method for distance > 1km, it should return False",False,test2.equal(test))

    test = GPosT(45,45)
    test2 = GPosT(45,-45)
    compare("test west_of method, it should return True",True,test2.west_of(test))

    compare("test west_of method, it should return False",False,test.west_of(test2))

    test = GPosT(45,45)
    test2 = GPosT(50,-45)
    compare("test north_of method, it should return True",True,test2.north_of(test))
    compare("test north_of method, it should return False",False,test.north_of(test2))

    test = GPosT(43,-75)
    test2 = GPosT(44.078061, -73.170068)

    test.move(45,100)
    compare("test move method",GPosT(43.63, -74.12),test)

    date = DateT(18,1,2020)
    test = GPosT(43,-75)
    val = test.arrival_date(test2,date,180) #starting from 43,-75 travel to test2 at 180km/day
starting on date
    compare("test arrival date while travelling at speed that allowed to reach within the same (i.e
days required < 1), it should return Jan 19th, 2020",DateT(19,1,2020),val)

    test.distance = test.distance(test2)
    val = test.arrival_date(test2,date,test.distance)
    compare("test arrival date with travelling speed that takes exactly 1 day in total, it should
return Jan 19th 2020",DateT(19,1,2020),val)

    val = test.arrival_date(test2,date,14.62)
    compare("test arrival date with travelling speed that takes until the 31st of the month, it should
return Jan 31st 2020",DateT(31,1,2020),val)

    date = DateT(1,2,2020)
    val = test.arrival_date(test2,date,6.55)
    compare("test arrival date that takes 29 days to travel in feb, it should return March 1st
2020",DateT(1,3,2020),val)

    val = test.arrival_date(test2,date,0)
    compare("test arrival date with 0 speed it should return the current date",DateT(1,2,2020),val)

def main():
    start = time.time()
    print("Tests for date_adt.py")
    test_date_adt()

    print("\nTESTS FOR pos_adt.py")
    test_post_adt()
    end = time.time()
    end = round(end - start,5)

```

```

if (len(failed)!=0):
    print("\x1b[1;37;41m{num} tests failed\nTests highlighted above in red failed\nThe following
          tests failed: \x1b[0m".format(num=len(failed)))
    for i in range(len(failed)):
        print("\n{num}) Description:
              {Description}".format(num=(i+1),Description=failed[i]["Description"]))
    else: print("\x1b[6;30;42m All tests passed. Executed {count} tests in {time} seconds
              \x1b[0m".format(count=count,time=end))

main()

```

H Code for Partner's CalcModule.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 East, negative long is West
    # @exception ValueError shows if latitude or longitude is out of range.
    def __init__(self, lat, long):
        if lat > 90 or lat < -90 or long > 180 or long < -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 GPosT 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 GPosT 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)

```