Assignment 1 Solution

Madhi Nagarajan, nagarajm

January 28, 2020

This purpose of this assignment is based on creating ADTs for a DateT and GPosT data-type. The DateT data-type stores a date and completes date-related calculations. The GPosT data-type stores a geographical coordinate and completes geographical calculations. Testing of these ADTs were also done to verify our personal code and our partner's code.

1 Testing of the Original Program

My approach to testing my program was to utilize "assert" with unittest to provide whether there were any failures or not. When testing the move and distance function, I used https://www.movable-type.co.uk/scripts/latlong.html as a basis for my test case answers. For the arrival_date function, I first calculated the distance. I then calculate the number of days, using the distance calculated and speed (in km/days). For the final date result, used https://www.timeanddate.com/date/dateadd.html.
When testing my code, I noticed that when testing functions in pos_adt that returned decimal values, these test cases would fail unless if both numbers were identical. Since this is unreasonable and inconvenient for testing, I used "self.assertAlmostEqual" and utilized a range parameter in which both values can differ at max between each other (eg. within 0.1 of an expected value).

2 Results of Testing Partner's Code

When testing Almen's code, all test functions, but "test_arrival_date", passed. In this test function, the test cases failed by one day. This was due to the arrival_date of my partner's pos_adt, as she used the ceil() function. However in my code, I used the round() function. While I believe that both ways of rounding could be valid, there was still some bias in my test cases. The Assignment Specification did not specify the rounding of days, thus I made the assumption that day value should just be rounded normally.

3 Critique of Given Design Specification

I liked Almen's implementation, as she used the datetime package. This made the date_adt code simpler, as datetime took care of the date/calendar implementation. One feedback of Almen's design is to have some variable names shorter, as it would it make it easier for her when writing out code.

4 Answers to Questions

(a) The state variables used for the DateT ADT were day, month, and year. The state variables used for the GPosT ADT were latitude and longitude.

(b) DateT is immutable. This is because its methods are encapsulated. There is no way its methods can unintentionally be modified by another class. GPosT is mutable because of the move() function. When move is called, the GPosT object passed through is directly modifed.

(c) Pytest offers a more detailed explanation of where a failure occurs for test cases. The test script could use just "assert" rather than "self.assert". While I did not use pytest, I utilized unittest (a demo of pytest) which can be used right out of the box in python. I found unittest also uses the "assert" syntax, however it does not offer an as detailed explanation for failures.

(d) Some past examples of Software Engineering failures are NASA's Spirit Rover (shutdown because it ran out of flash memory) and the 2003 Northeast Blackout (due to a race condition).
Software quality and high cost is still a major challenge because software quality directly correlates with cost. To improve the quality of software, there is generally higher cost associated with that improvement because it's very difficult to make software correct, reliable and robust. Oftentimes, companies usually undervalue the cost and effort for a software project and as a result, it could actually lead to unwanted outcomes and costs as seen in the examples above. I believe the best way to address the challenge of software quality while reducing costs is to follow and maintain a design process best suited for a specific project. This ensures that a team is organized and efficient, while also mainitaining software quality while keeping the cost a minimal.

(e) The rational design process is a waterfall model of how software should be designed and implemented. While the process is a rational one, it not so reasonable in many

software environments. This is because there are often revisions, changes, and improvements to not only the documentation, but even the software & code itself.
The advantages of following documentation in a rational design process is that it gives a organized procedure such that a whole team can all understand and follow. It makes design reviews easier, increases efficiency, and the overall progress of a software project.

(f) Correctness is related to how a software meets its requirements and specification. It is generally difficult to acheive. Robustness is how well it behaves when put under unanticipated situations, and is usually accomplished by satisfying unstated requirements. Reliability measures how reliable the software does what it is intended to do. It is more of a relative quality.

(g) Seperation of concerns is when we wish to isolate different concerns such that they won't be able to directly interact with each other. Modularity is when a system is divided into smallar subsystems called modules. Modularity utlizes seperation of concerns, as each module is considered seperate and unique. When the modules are put together, they are considered a different system.

E Code for date_adt.py

```
## @file date_adt.py
#  @title Date ADT
#  @author Madhi Nagarajan
#  @brief This file is meant to act as an ADT for a calendar and to perform date-related calculations
#  @date January 20, 2020

## @brief The class, DateT, represents an ADT of a calendar date
#  @details This class represents represents an ADT of a calendar with the ability
#  to perform date-related calculations, utilizing the day (d), month (m), and year (y)
class DateT:

    ## dictionary that stores the corresponding number of days for each month
    calendar = {1: 31, 2: 28, 3: 31, 4: 30, 5: 31, 6: 30, 7: 31, 8: 31, 9: 30, 10: 31, 11: 30, 12: 31}

    ## @brief Constructor for DateT
    #  @details Constructor accepts three parameters for the day, month, and year.
    #  @param d is an int value for the respective day.
    #  @param m is an int value for the respective month.
    #  @param y is an int value for the respective year.
    def __init__(self, d, m, y):
        self.d = d
        self.m = m
        self.y = y

    ## @brief Getter method for returning day
    #  @returns The d value for the day
    def day(self):
        return self.d

    ## @brief Getter method for returning month
    #  @returns The m value for the month
    def month(self):
        return self.m

    ## @brief Getter method for returning year
    #  @returns The y value for the year
    def year(self):
        return self.y

    ## @brief This function calculates the next date of the given date
    #  @returns The next day of the given date
    def next(self):
        date = DateT(self.day(), self.month(), self.year())
        if DateT.calendar[date.m] == date.d:
            if date.m == 12:
                date.y = date.y + 1
                date.m = 1
                date.d = 1
            elif date.m == 2 and date.y % 4 == 0:
                if self.y % 100 == 0 and self.y % 400 != 0:
                    date.m = 3
                    date.d = 1
                else:
                    date.d = 29
            else:
                date.m = date.m + 1
                date.d = 1
        elif date.m == 2 and date.d == 29:
            date.m = date.m + 1
            date.d = 1
        else:
            date.d = date.d + 1
        return date

    ## @brief This function calculates the previous date of the given date
    #  @return Returns the previous day of the given date
    def prev(self):
        date = DateT(self.day(), self.month(), self.year())
        if date.d == 1:
            if date.m == 1:
                date.y = date.y - 1
                date.m = 12
                date.d = 31
            elif date.m == 3 and date.y % 4 == 0:
                if (date.y % 100 == 0) and (date.y % 400 != 0):
                    date.d = 28
```

```
                date.m = date.m - 1
            else:
                date.d = 29
                date.m = date.m - 1
        else:
            date.d = DateT.calendar[date.m - 1]
            date.m = date.m - 1
    else:
        date.d = date.d - 1
    return date

## @brief This function checks if the current date is before the given date
# @param d is a given date
#  @return Returns a boolean based on whether or not the current date is before the given date
def before(self, d):
    if d.y == self.year():
        if d.m == self.month():
            if d.d > self.day():
                return True
            else:
                return False
        elif d.m > self.month():
            return True
        else:
            return False
    elif d.y > self.year():
        return True
    else:
        return False

## @brief This function checks if the current date is after the given date
# @param d is a given date
#  @return Returns a boolean based on whether or not the current date is after the given date
def after(self, d):
    if d.y == self.year():
        if d.m == self.month():
            if d.d < self.day():
                return True
            else:
                return False
        elif d.m < self.month():
            return True
        else:
            return False
    elif d.y < self.year():
        return True
    else:
        return False

## @brief This function checks if the current date is equal to the given date
# @param d is a given date
#  @return Returns a boolean based on whether or not the current date is equal to the given date
def equal(self, d):
    if d.d == self.day() and d.m == self.month() and d.y == self.year():
        return True
    else:
        return False


## @brief This function adds a certain number of days to the current date and returns the
     calculated date
# @param n is a given number of days,
#  @return d returns the calculated date after the days have been added
def add_days(self, n):
    d = DateT(self.day(), self.month(), self.year())
    for i in range(n):
        d = d.next()
    return d


## @brief This function finds the number of days between the current and given dates
# @param d is a given date
#  @return n returns an int value, the number of days between the current & given dates
def days_between(self, d):
    n = 0
    if self.after(d):
        while not (self.equal(d)):
            d = d.next()
            n = n + 1
    else:
```

```
    while not (self.equal(d)):
        d = d.prev()
        n = n + 1
return n
```

F Code for pos_adt.py

```
## @file pos_adt.py
#  @title Pos ADT
#  @author Madhi Nagarajan
#  @brief This file is meant to act as an ADT for a global coordinate and to perform location-related
     calculations
#  @date January 20, 2020

import math
from date_adt import DateT

## @brief The class, GPosT, represents an ADT of a global coordinate
#  @details This class represents represents an ADT of a global coordinate with the ability
#  to perform location-related calculations, utilizing the latitude and longitude
class GPosT:

    ## @brief Constructor for GPosT
    #  @details Constructor accepts two parameters for the latitude and longitude.
    #  @param y is a double value for the respective latitude.
    #  @param x is a double value for the respective longitude.
    def __init__(self, y, x):
        self.latitude = y
        self.longitude = x

    ## @brief Getter method for returning latitude
    #  @returns The latitude value
    def lat(self):
        return self.latitude

    ## @brief Getter method for returning longitude
    #  @returns The longitude value
    def long(self):
        return self.longitude

    ## @brief The function calculates if the current coordinate is west of the given coordinate
    #  @param p is a given coordinate.
    #  @returns A boolean depending on if the current coordinate is west of the given coordinate
    def west_of(self, p):
        if p.long() > self.long():
            return True
        else:
            return False

    ## @brief The function calculates if the current coordinate is north of the given coordinate
    #  @param p is a given coordinate.
    #  @returns A boolean depending on if the current coordinate is north of the given coordinate
    def north_of(self, p):
        if p.lat() < self.lat():
            return True
        else:
            return False

    ## @brief The function calculates if the current coordinate is equal to the given coordinate
    #  @param p is a given coordinate.
    #  @returns A boolean depending on if the current coordinate is equal to the given coordinate
    def equal(self, p):
        if p.lat() == self.lat() and p.long() == self.long():
            return True
        else:
            return False

    ## @brief The function calculates a resultant coordinate given a certain bearing & distance
    #  @param b is a given bearing (in degrees).
    #  @param d is a given distance (in km).
    def move(self, b, d):
        R = 6371
        rlat = math.radians(self.lat())
        rlong = math.radians(self.long())
        rb = math.radians(b)
        d2 = d / R
        rlat2 = math.asin((math.sin(rlat) * math.cos(d2)) + (math.sin(d2) * math.cos(rlat) *
             math.cos(rb)))
        rlong2 = rlong + math.atan2(math.sin(rb) * math.sin(d2) * math.cos(rlat),
                                    math.cos(d2) - math.sin(rlat) * math.sin(rlat2))
        self.latitude = math.degrees(rlat2)
        self.longitude = math.degrees(rlong2)
```

```
## @brief The function calculates the distance between the current point and another point
#  @param p is a given coordinate.
#  @returns dist, the resultant distance (in km)
def distance(self, p):
    R = 6371
    rlat1 = math.radians(self.lat())
    rlat2 = math.radians(p.lat())
    latdiff = math.radians(p.lat() - self.lat())
    longdiff = math.radians(p.long() - self.long())

    a = math.sin(latdiff / 2) * math.sin(latdiff / 2) + math.cos(rlat1) * math.cos(rlat2) *
        math.sin(
        longdiff / 2) * math.sin(longdiff / 2)
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    dist = R * c
    return dist

## @brief The function calculates the final date based on the speed, destination point, starting
    date
#  @param The parameters passed through is the given destination coordinate, p,
#  @param d is the starting date, and s is the speed in (km/day)
#  @returns d2, the resultant date
def arrival_date(self, p, d, s):
    dist = GPosT.distance(self, p)
    days = round(dist / s)
    d2 = DateT.add_days(d, days)
    return d2
```

G Code for test_driver.py

```
## @file test_driver.py
#  @author Madhi Nagarajan
#  @brief Test Driver for DateT and GPosT
#  @date Jan 20, 2020

import unittest
from date_adt import DateT
from pos_adt import GPosT


d1 = DateT(1, 1, 2020)
d11 = DateT(1, 1, 2021)
d2 = DateT(31, 12, 1500)
d22 = DateT(1, 2, 1501)
d3 = DateT(25, 4, 1764)
d4 = DateT(6, 10, 787)
d44 = DateT(6, 10, 787)
d5 = DateT(29, 11, 1999)
d6 = DateT(28, 2, 2220)

p1 = GPosT(43.5, 79.1)
p2 = GPosT(-12.357, -169.599)
p22 = GPosT(-12.357, -169.599)
p3 = GPosT(83.3, -12.47)
p4 = GPosT(23.77, 99.521)


class TestT(unittest.TestCase):

    def main(self):
        self.test_days()
        self.test_months()
        self.test_years()
        self.test_prev()
        self.test_next()

    def test_days(self):
        self.assertEqual(d1.day(), 1)

        assert d1.day() == 1
        assert d2.day() == 31
        assert d3.day() == 25
        assert d4.day() == 6

    def test_months(self):
        assert d1.month() == 1
        assert d2.month() == 12
        assert d3.month() == 4
        assert d4.month() == 10

    def test_years(self):
        assert d1.year() == 2020
        assert d2.year() == 1500
        assert d3.year() == 1764
        assert d4.year() == 787

    def test_next(self):
        assert d2.next().month() == 1
        assert d2.next().day() == 1
        assert d2.next().year() == 1501
        assert d4.next().month() == 10
        assert d4.next().day() == 7
        assert d4.next().year() == 787

    def test_prev(self):
        assert d1.prev().month() == 12
        assert d1.prev().day() == 31
        assert d1.prev().year() == 2019
        assert d4.prev().month() == 10
        assert d4.prev().day() == 5
        assert d4.prev().year() == 787

    def test_after(self):
        assert d1.after(d2) == True
        assert d1.after(d3) == True
        assert d2.after(d3) == False
```

```
    assert d2.after(d4) == True

def test_before(self):
    assert d1.before(d2) == False
    assert d4.before(d1) == True
    assert d3.before(d1) == True
    assert d3.before(d4) == False

def test_equal(self):
    assert d1.equal(d2) == False
    assert d4.equal(d44) == True

def test_add_days(self):
    assert d2.add_days(7).month() == 1
    assert d2.add_days(7).day() == 7
    assert d2.add_days(7).year() == 1501
    assert d1.add_days(30).month() == 1
    assert d1.add_days(30).day() == 31
    assert d1.add_days(30).year() == 2020
    assert d4.add_days(3653).month() == 10
    assert d4.add_days(3653).day() == 6
    assert d4.add_days(3653).year() == 797

def test_days_between(self):
    assert d1.days_between(d11) == 366
    assert d2.days_between(d22) == 32

def test_lat(self):
    assert p1.lat() == 43.5
    assert p2.lat() == -12.357
    assert p3.lat() == 83.3

def test_long(self):
    assert p1.long() == 79.1
    assert p2.long() == -169.599
    assert p3.long() == -12.47

def test_west_of(self):
    assert p1.west_of(p2) == False
    assert p2.west_of(p4) == True
    assert p3.west_of(p1) == True

def test_north_of(self):
    assert p1.north_of(p2) == True
    assert p3.north_of(p4) == True
    assert p1.north_of(p3) == False

def test_equal(self):
    assert p2.equal(p22) == True
    assert p1.equal(p3) == False

def test_move(self):
    p1.move(14.5, 452)
    self.assertAlmostEqual(p1.lat(), 47.4261, delta=0.1)
    self.assertAlmostEqual(p1.long(), 80.603, delta=0.1)
    p2.move(46.5, 985.33)
    self.assertAlmostEqual(p2.lat(), -6.1925, delta=0.1)
    self.assertAlmostEqual(p2.long(), -163.145, delta=0.1)
    p3.move(-33.61, 2673.67)
    self.assertAlmostEqual(p3.lat(), 71.1877, delta=0.1)
    self.assertAlmostEqual(p3.long(), -148.089, delta=0.1)

def test_distance(self):
    self.assertAlmostEqual(p1.distance(p2), 12660, delta=5)
    self.assertAlmostEqual(p2.distance(p4), 10650, delta=5)
    self.assertAlmostEqual(p3.distance(p1), 5232, delta=5)

def test_arrival_date(self):
    res = p1.arrival_date(p2, d1, 263)
    self.assertEqual(res.year(), 2020)
    self.assertEqual(res.month(), 2)
    self.assertEqual(res.day(), 18)

    res2 = p4.arrival_date(p3, d5, 1.64)
    self.assertEqual(res2.year(), 2012)
    self.assertEqual(res2.month(), 9)
    self.assertEqual(res2.day(), 11)

    res3 = p2.arrival_date(p3, d4, 17.4)
    self.assertEqual(res3.year(), 789)
```

```
        self.assertEqual(res3.month(), 8)
        self.assertEqual(res3.day(), 29)


if __name__ == "__main__":
    unittest.main()
```

H Code for Partner's date_adt.py

```
## @file date_adt.py
#  @author Almen Ng
#  @brief Provides the DateT ADT class for representing dates
#  @date January 20, 2020

from datetime import date, timedelta

## @brief An ADT that represents a date
class DateT:

  ## @brief DateT constructor
  #  @details Initializes a DateT object with day, month, and year of date
  #  @param d The day of the date
  #  @param m The month of the date
  #  @param y The year of the date
  def __init__(self, d, m, y):
      self.__d = d
      self.__m = m
      self.__y = y

  ## @brief Gets the day of the date
  #  @return The day of the date
  def day(self):
      return self.__d

  ## @brief Gets the month of the date
  #  @return The month of the date
  def month(self):
      return self.__m

  ## @brief Gets the year of the date
  #  @return The year of the date
  def year(self):
      return self.__y

  ## @brief Gets the date one day later than the current object
  #  @return Date one day later than the current object
  def next(self):
      next_date = date(self.__y, self.__m, self.__d) + timedelta(days=1)
      return DateT(next_date.day, next_date.month, next_date.year)

  ## @brief Gets the date one day before the current object
  #  @return Date one day before the current object
  def prev(self):
      prev_date = date(self.__y, self.__m, self.__d) - timedelta(days=1)
      return DateT(prev_date.day, prev_date.month, prev_date.year)

  ## @brief Checks to see if the current date is before d
  #  @param d Date object of type DateT to compare the current date with
  #  @return True if the current date is before d. False otherwise.
  def before(self, d):
      if date(self.__y, self.__m, self.__d) < date(d.__y, d.__m, d.__d):
          return True
      else:
          return False

  ## @brief Checks to see if the current date is after d
  #  @param d Date object of type DateT to compare the current date with
  #  @return True if the current date is after d. False otherwise.
  def after(self, d):
      if date(self.__y, self.__m, self.__d) > date(d.__y, d.__m, d.__d):
          return True
      else:
          return False

  ## @brief Checks to see if the current date is equal to d
  #  @param d Date object of type DateT to compare the current date with
  #  @return True if the current date is equal to d. False otherwise.
  def equal(self, d):
      if date(d.__y, d.__m, d.__d)  == date(self.__y, self.__m, self.__d):
          return True
      else:
          return False

  ## @brief Adds a certain amount of dates to the current date (assuming that negative days cannot be
       added)
```

```
#  @param n Integer representing the number of days needed to add to the current date
#  @return The date that is n days later than the date of the current object
def add_days(self, n):
    days_added = date(self.__y, self.__m, self.__d) + timedelta(days=n)
    return DateT(days_added.day, days_added.month, days_added.year)

## @brief Calculates the number of days between 2 dates
#  @param d Date object of type DateT to compare the current date with
#  @return The number of days between 2 dates, the current and d
def days_between(self, d):
    return abs((date(self.__y, self.__m, self.__d) - date(d.__y, d.__m, d.__d)).days)
```

I Code for Partner's pos_adt.py

```
## @file pos_adt.py
#  @author Almen Ng
#  @brief Provides the GPosT ADT class for representing position using latitude and longitude
#  @date January 20, 2020

from math import asin, sin, cos, radians, atan2, sqrt, degrees, ceil

## @brief An ADT that represents a position
class GPosT:

    ## @brief GPosT constructor
    #  @details Initializes a GPosT object with latitude and longitude of a position
    #  @param y The latitude of the position (assuming that the latitude does not exceed +- 90 degrees)
    #           North is positive, South is negative
    #  @param x The longitude of the position (assuming that the longitude does not exceed +-180
        degrees)
    #           East is positive, West is negative
    def __init__(self, y, x):
        self.__x = x
        self.__y = y

    ## @brief Gets the latitude of the position
    #  @return The latitude of the position
    def lat(self):
        return self.__y

    ## @brief Gets the longitude of the position
    #  @return The longitude of the position
    def long(self):
        return self.__x

    ## @brief Checks to see if the current position is to the west of another position
    #  @param p Position of object of type GPosT to compare the current position with
    #  @return True if the current position is to the west of p. False otherwise.
    def west_of(self, p):
        if self.__x < p.__x:
            return True
        else:
            return False

    ## @brief Checks to see if the current position is to the north of another position
    #  @param p Position of object of type GPosT to compare the current position with
    #  @return True if the current position is to the north of p. False otherwise.
    def north_of(self, p):
        if self.__y > p.__y:
            return True
        else:
            return False

    ## @brief Checks to see if the current position is "equal" to another position
    #         Has to be to within 1 km from the current position to be considered "equal"
    #  @param p Position of object of type GPosT to compare the current position with
    #  @return True if the current position is to "equal" to p. False otherwise.
    def equal(self, p):
        if (self.distance(p)) < 1:
            return True
        else:
            return False

    ## @brief Changes the position of the current object (longitude and latitude by starting from the
        current position and
    #         moving at bearing b for a distance of d
    #  @param b Signed decimal degree of type real representing bearing
    #  @param d The distance travelled in units of km
    def move(self, b, d):
        R = 6371
        angular_distance = d/R
        latitude_original = self.__y
        longitude_orignal = self.__x

        self.__y =  degrees(asin(sin(radians(latitude_original)) * cos(angular_distance) +
            cos(radians(latitude_original)) * sin(angular_distance) * cos(radians(b))))
        self.__x = degrees(radians(longitude_orignal) + atan2(sin(radians(b)) * sin(angular_distance)
            * cos(radians(latitude_original)), cos(angular_distance) -
            sin(radians(latitude_original)) * sin(radians(self.__y))))
```

```
## @brief Calculates the distance between the current position and p
#  @param p Position of object of type GPosT to find the distance from
#  @return Distance (in km) between the current position and p
def distance(self, p):
    delta_lat = radians(p.__y) - radians(self.__y)
    delta_long = radians(p.__x) - radians(self.__x)
    R = 6371

    a = (sin(delta_lat/2))**2 + cos(radians(self.__y)) * cos(radians(p.__y)) *
        (sin(delta_long/2))**2
    c = 2 * atan2(sqrt(a), sqrt(1-a))

    return R * c

## @brief Calculates the arrival date for someone starting at the current position on a certain
    date and moving
#         to another position at a certain speed
#  @param p Position of object of type GPosT to move to
#  @param d Date of type DateT of the starting date of the travel
#  @param s Speed of type real at which someone is moving at in km/day
#  @return The arrival date after travelling from the current position on date d and moving to
    position p at a speed of s
def arrival_date(self, p, d, s):
    days_past = ceil(self.distance(p) / s)
    return d.add_days(days_past)
```