Assignment 1 Solution

Utsharga Rozario, rozariou

January 28, 2020

This report discusses testing of two Abstract Data Types, DateT and GPosT written for Assignment 1. The DateT ADT represents the date consisting of day, month and year while the GPosT ADT represents the global positioning system in terms of latitude and longitude. It also discusses testing of the partner's version of the program. The design restrictions for the assignment are critiqued and then various discussion questions related are answered. The program was written in Python programming language with the code for both my own program and my partners program given in the pages below.

1 Testing of the Original Program

Tests for the program was on the basis of normal, edge and large cases. The tests were done on each method, containing 5 test cases intermixed with normal, edge and large cases. If a test case passes a prompt is output that it passes that case, otherwise an output fail is output. Normal cases were done to check if it the program ran properly for normal scenarios. Edge cases were done at the maximum and minimum ends to check if problems occur at the extreme operating parameters. Large cases were done with high input values to check if the program can compute the large data without running into an error.

Some assumptions were made about the program's behaviour for cases that were not fully specified. Specifically, it was assumed that:

- for the `DateT ADT`, it follows the Gregorian calendar,
- leap year occurs every four years (divisible by 4) and skips a leap year every 100 years to account for the time,
- January 1st, year 1 (AD), is assumed to be the first year on the calendar and years before that are not accounted for,

- the months have days according to the Gregorian calendar, and February has 29 days when it is a leap year,
- for `add_days` method in `DateT ADT`, the number of days added are assumed to be input correctly as positive numbers.
- for the `GPosT ADT`, latitude and longitude are taken in terms of degrees with range - 90 to 90 degrees and -180 to 180 degrees respectively,
- for `west_of` method in `GPosT ADT`, the current position is considered west of the parameter position if the longitude of the current position is less than the parameter position,
- for `west_of` method in `GPosT ADT`, the current position is considered north of the parameter position if the latitude of the current position is greater than the parameter position,
- for `equal` method in `GPosT ADT`, if the distance is equal to 1 km, the parameter position is not equal to the current position,
- for `move` method in `pos_adt.py`, the bearing and distance were assumed to be input correctly within range of 0 degrees from the north to 360 degrees, and positive for the distance.
- for `arrival_date` method in `GPosT ADT`, the speed was assumed to be input as a positive number above 0
- for `arrival_date` method in `GPosT ADT`, the assumed departure time was 12:00 am of the current date, hence if the time was less than one day, the current date was returned. Otherwise, the time (number of days) minus one day was added to the current date and then returned.

The Original Program passed all the tests.

2 Results of Testing Partner's Code

Partner's Code passed all test cases expect the test for `arrival_date`. This error occurred due to an assumption I made while writing the program. My partner assumed that if the number of days were more than 0 then the arrival date would be the number of days after the the current date. Hence his assumed departure date would be 11:59 pm of the current day.

`currentPos` = GPosT(0.00, 0.00)
`testArrivalDate` = *currentPos.arrival_date*(GPosT(87.9467, 80.7486), DateT(1,2,2000), 9970.84535815507)

`Expected arrival date:` 1,2,2000
`Partner's arrival date:` 2,2,2000

3 Critique of Given Design Specification

`Advantages` of the design specification included the types chosen to represent some of the data, such as representing the date in day, month and year as integers instead of a string, preventing one from needing to parse a string as a integer when accessing the date data. Similarly, for the global positioning data, the longitude and latitude were in real numbers representing degrees, hence preventing one from needing to parse a string as a real number when accessing the positioning data. Most of the methods specified the return data type with it's appropriate metric units and formats, which minimised the chances of error raised through method calling.

`Disadvantages` of the design specification include the ambiguity of the range of input. There is a lot of date format, e.g. the Gregorian calendar, the Chinese calendar, etc. As well as the global positioning in terms of degrees, degrees/min and degrees/min/sec. There was a lot of space for assumptions in terms of what the methods do.

For example in `days_between`, we can assume to include or exclude the current day and the parameter date, and in `arrival_date`, we can assume the what time the departure was.

4 Answers to Questions

(a) Possible state variables for `DateT ADT` could be a tuple of *day*, *month* and *year* in the format (day, month, year) or create a list of *day*, *month* and *year* in the order of [day, month, year].

Possible sate variable for `GPosT ADT` could be a tuple of *phi* (latitude) and *lamda* (longitude) in the format (phi, lamda) or create a list of *phi* and *lamda* in the order of [phi, lamda].

(b) From the interface specified in the assignment, DateT ADT is immutable as it cannot be modified using methods. All the methods return a new DateT object and does not modify the current object.

The GPosT ADT is mutable as it can be modified using methods from the ADT. The *move* method modifies the phi and the lamda value to a new value. Hence, it is mutable.

(c) Using `pytest`, it is easier to build test cases for abstract data types. It provides many benefits such as automated tests that can be specific and tailored to the API. A big advantage with pytest is that it is easy to use and utilizes Python syntax. As a result, the test cases are expressive and readable. Designing code for unit testing is easier and tidier, as it requires you to split the modules into smaller components.

(d) An expensive example of an software engineering failure is Bank Heist of the Bangldesh Bank in 2016. According to Bangladesh Bank authorities, a printer is set up to automatically print read-outs of transactions made. The glitch in the system, interrupted the automatic printing process, so that is was only several days later that the transfer receipts were even discovered – giving the thieves plenty of time to cover their tracks. A total of $ 81 million was stolen from the funds due to this software bug.

A very recent example of an software engineering failure is a system failure in the British Airways. An IT glitch in the system caused it to fail resulting in more than 100 flights to be cancelled and more than 200 others to delayed.

Software quality and high cost still a major challenge because:

- a software product can almost never be bug free, especially for large scale projects and products,
- problems appear throughout the use of the product, hence maintenance is required on a periodic basis,
- human error in writing the code as well as understanding the requirement specification,
- some errors/problems cannot be foreseen due new development in computation (i.e. using supercomputers to hack a system),
- the rapid change of technology and design systems also pose as a major challenge due to software continually needing to be updated for the change.

To address the challenge in the future, a standard of documentation needs to be established and certain metrics need to be created to have a uniform standard of code. Till then, software engineers and developers need to follow the best and optimized Rational Design Process, one which is focused on change and robustness.

(e) The Rational Design Process discussed in class is:

1. Problem statement
2. Development plan
3. Requirements (SRS)
4. Design Docs (MG) & (MIS)
5. Code
6. V&V Report
7. Maintain

It is necessary to fake this process because the resulting product can be understood, maintained, and reused. Maintainability and reusability are key components of software engineering.

The advantages to faking this process are ease of redesign, ease of redevelopment, increased maintainability and reusability. Faking this ideal process is costly in the short term, but inexpensive in the long term.

(f) `Software correctness`: A software product is correct if satisfies its requirements specification. In theory, it can be hard to achieve due to imprecise, ambiguous, inconsistent, based on incorrect knowledge, or nonexistent but can be possible if the requirement specifications are formal.

`Software Reliability`: A software product is reliable if it usually does what is intended to do. Reliability can be measured.

`Software Robustness`: A software product is robust if it behaves reasonably even in unanticipated or exceptional situations. It accounts for cases unspecified in the requirements.

A robust software product is correct but a correct software product does not need to be robust. Meanwhile, a software product can be reliable and incorrect.

(g) Separation of concerns is the principle that different concerns should be isolated and considered separately.

The motivation behind this principle is to reduce a complex problem into a set of smaller problems. Hence, enables smaller components to be tackled in parallel. This further allows the program to be easily debugged and modules to be switched out in case of error or update. Furthermore, because of this principle, we consider software systems from different view points and have the qualities seperately concerned.

The principles behind Modularity and Separation of Concerns are related because both principles focus on breaking up large complex programs into smaller simple programs that communicate with each other. Separation of concerns encourages modularity hence in turn making the program maintainable, reusable, readable, and interchangeable as much as possible.

E Code for date_adt.py

```
## @file date_adt.py
#  @author Utsharga Rozario
#  @brief Provides the DateT ADT class for representating dates
#  @date 20/1/2020

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

  ## @brief Date Constructor
  #  @details Initializes a DateT object with day, month and year
  #  @param d The day of the date
  #  @param m The month of the date
  #  @param y The year of the date
  #  @exception ValueError throws if the date is not a valid date
  def __init__(self, d, m, y):
    if (y < 0):
      raise ValueError(”Date is not valid date”)
    elif (m < 1 or m > 12):
      raise ValueError(”Date is not valid date”)
    elif (m == 2 and ((y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)) and d > 29):
      raise ValueError(”Date is not valid date”)
    elif (m == 2 and d > 28 and (y%4 != 0)):
      raise ValueError(”Date is not valid date”)
    elif ((m == 1 or m == 3 or m == 5 or m == 7 or m == 8 or m == 10 or m == 12) and d > 31):
      raise ValueError(”Date is not valid date”)
    elif ((m == 4 or m == 6 or m == 9 or m == 11) and d > 30):
      raise ValueError(”Date is not valid date”)
    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 Determines if the year is a leap year or not
  #  @return True if it is leap year, False
  def __leapYear(self):
    return (((self.year() % 4) == 0 and ((self.year() % 100) != 0)) or ((self.year() % 400) == 0))

  ## @brief Calculates the next date in respective to current date
  #  @details Determines which date it is, if it is the last date of the month then it increments the
      month,
  #           and also the day to 1, also considers cases for leap year
  #  @return The next date from the current date in the DateT format
  def next(self):
    if (self.day() < 30):
      if (self.month() != 2):
        return DateT(self.day() + 1, self.month(), self.year())
      elif (self.month() == 2):
        if (self.day() < 28):
          return DateT(self.day() + 1, self.month(), self.year())
        elif (self.day() == 28 and not(self.__leapYear())):
          return DateT(1, 3, self.year())
        elif (self.day() == 28 and self.__leapYear()):
          return DateT(self.day() + 1, self.month(), self.year())
        elif (self.day() == 29 and self.__leapYear()):
          return DateT(1, 3, self.year())
    elif (self.day() == 30 and (self.month() == 4 or self.month() == 6 or self.month() == 9 or
        self.month() == 11)):
      return DateT(1, self.month() + 1, self.year())
    elif (self.month() == 1 or self.month() == 3 or self.month() == 5 or self.month() == 7 or
        self.month() == 8 or self.month() == 10):
      if (self.day() == 30):
        return DateT(31, self.month(), self.year())
      elif (self.day() == 31):
        return DateT(1, self.month() + 1, self.year())
    elif(self.month() == 12):
      if (self.day() == 30):
        return DateT(31, self.month(), self.year())
      elif (self.day() == 31):
        return DateT(1, 1, self.year() + 1)

  ## @brief Calculates the previous date in respective to current date
  #  @details Determines which date it is, if it is the first date of the month then it decrements the
      month,
  #            and also the day to the corresponding month's last day, also considers cases for leap
      year
  #  @return The previous date from the current date in the DateT format
  def prev(self):
    if (self.day() > 1):
      return DateT(self.day() - 1, self.month(), self.year())
    elif (self.day() == 1):
      if (self.month() == 1):
        return DateT(31, 12, self.year() - 1)
      elif (self.month() == 2 or self.month() == 4 or self.month() == 6 or self.month() == 8 or
           self.month() == 9 or self.month() == 11):
        return DateT(31, self.month() - 1, self.year())
      elif (self.month() ==3):
        if (self.__leapYear()):
          return DateT(29, 2, self.year())
        elif(self.__leapYear()):
          return DateT(28, 2, self.year())
      elif (self.month() == 5 or self.month() == 7 or self.month() == 10 or self.month() == 12):
        return DateT(30, self.month() - 1, self.year())

  ## @brief Determines if current date is before to the date d
  #  @param d Date in DateT format
  #  @return Returns true if current date is before the date d, false if not
  def before(self, d):
    if (d.year() > self.year()):
      return True
    elif (self.year() == d.year()):
      if (d.month() > self.month()):
        return True
      elif (self.month() == d.month()):
        if (d.day() > self.day()):
          return True
        else:
          return False
      elif (d.month() < self.month()):
        return False
    elif (d.year() < self.year()):
      return False

  ## @brief Determines if current date is after to d date
  #  @param d Date in DateT format
  #  @return Returns true if current date is after the date d, false if not
  def after(self, d):
    return not(self.before(d))

  ## @brief Compares current Date with another date
  #  @param d Date in DateT format
  #  @return Returns true if current date is equal to the date d, false if not
  def equal(self, d):
    if(self.day() == d.day() and self.month() == d.month() and self.year() == d.year()):
      return True
    else:
      return False

  ## @brief Calculates the Date after n number of days
  #  @details Utilizes the next method form DateT class, performs a for loop calculate the date after
      n number of days
  #  @param n The number of days to be added to the current date
  #  @return Date after n number of days
  def add_days(self, n):
    added_date = DateT(self.day(), self.month(), self.year())
    for m in range(n):
      added_date = added_date.next()
    return added_date

  ## @brief Calculates the number of days between current Date and d Date
  #  @details Determines which of the dates is before and after, and performs appropriate
  #            while loop with conditions to match the date in the parameter. The while loop
  #            is executed on the basis that the next or prev method approaches the d date
  #            while incrementing a variable n, which is the days between them.
  #  @param d Date in DateT format
  #  @return The difference in interger between the current Date and d Date
  def days_between(self, d):
    n = 0
    d_date = DateT(self.day(), self.month(), self.year())
    if (d.year() > d_date.year()):
      while not (d.equal(d_date)):
        d_date = d_date.next()
        n = n + 1
    elif (d_date.year() == d.year()):
      if (d.month() > d_date.month()):
        while not (d.equal(d_date)):
          d_date = d_date.next()
          n = n + 1
      elif (d_date.month() == d.month()):
        if (d.day() > d_date.day()):
          n = d.day() - d_date.day()
        elif (d.day() == d_date.day()):
          n = 0
        elif (d.day() < d_date.day()):
          n = d_date.day() - d.day()
      elif (d.month() < d_date.month()):
        while not (d.equal(d_date)):
          d_date = d_date.prev()
          n = n + 1
    elif (d.year() < d_date.year()):
      while not (d.equal(d_date)):
        d_date = d_date.prev()
        n = n + 1

    return n
```

F Code for pos_adt.py

```
## @file pos_adt.py
#  @author Utsharga Rozario
#  @brief Provides the GPosT ADT for representating global position in degrees
#  @date 13/01/2020

from math import *
from date_adt import *

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

    ## @brief Date Constructor
    #  @details Initializes a GPosT object with phi (latitude in degrees) and lamda (longitude in
         degrees)
    #  @param phi The latitude in degrees
    #  @param lamda The longitude in degrees
    #  @exception ValueError throws if the the latitude or longitude is out of range
    def __init__(self, phi, lamda):
        if (phi < -90 or phi > 90):
            raise ValueError(”Latitude is out of range”)
        elif (lamda < -180 or lamda > 180):
            raise ValueError(”Longitude is out of range”)
        self.phi = phi
        self.lamda = lamda

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

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

    ## @brief Determines whether the parameter position is west of current position
    #  @param p Position of another global position of type GPosT
    #  @return Returns True if the current position is west of the parameter position, False if
         otherwise
    def west_of(self, p):
        if (p.long() > self.long()):
            return True
        else:
            return False

    ## @brief Determines whether the parameter position is north of current position
    #  @param p Position of another global position of type GPosT
    #  @return True if the curent position is north of the parameter position, false if otherwise
    def north_of(self, p):
        if (p.lat() < self.lat()):
            return True
        else:
            return False

    ## @brief Determines whether the parameter position is equal to the current position
    #  @details The method calculates whether the parameter position is equal to or less than 1km
    #           from current position which is then considered equal to the position
    #  @param p Position of another global position of type GPosT
    #  @return True if the parameter position is equal to the current position, false if otherwise
    def equal(self, p):
        if (self.long() == p.long() and self.lat() == p.lat()):
            return True
        elif (self.distance(p) < 1):
            return True
        else:
            return False

    ## @brief Translates the current global position in accordance to the provided parameters
    #  @param b The bearing at which to move the current posistion
    #  @param d The distance to move the current position
    def move(self, b, d):
        R = 6371
        delta = d/R
        phi1 = self.lat() * (pi/180)
        lamda1 = self.long() * (pi/180)
        b_rad = b * (pi/180)
    phi2 = asin(sin(phi1)*cos(delta) + cos(phi1)*sin(delta)*cos(b_rad))
    lamda2 = lamda1 +  atan2( (sin(b_rad)*sin(delta)*cos(phi1)), (cos(delta) -
        sin(phi1)*sin(phi2)))
    self.phi = phi2 * (180/pi)
    self.lamda = lamda2 * (180/pi)

## @brief Calculates the distance between current position and parameter position
#  @param p Position of another global position of type GPosT
#  @return Returns distance between two points
def distance(self, p):
    R = 6371
    phi1 = self.lat() * (pi/180)
    phi2 = p.lat() * (pi/180)
    delphi = (self.lat() - p.lat()) * (pi/180)
    dellamda = (self.long() - p.long()) * (pi/180)

    a = sin(delphi/2)*sin(delphi/2) + cos(phi1)*cos(phi2)*sin(dellamda/2)*sin(dellamda/2)
    c = 2*atan2(sqrt(a), sqrt(1-a))
    d = R*c

    return d

## @brief Calculates the date of arrival from
#  @details Calculates the distance between the parameter position and current position,
#           Calculates time in days required to reach parameter position using linear velocity
     formula (t = d/s),
#           Determines arrival date by by adding number of days to current date using add_days
     method
#  @param p Position of another global position of type GPosT
#  @param d Current date of type DateT
#  @param s Speed of someone moving from current position in km/day
#  @return Returns distance between two points
def arrival_date(self, p, d, s):
    dis = self.distance(p)
    time = ceil(dis/s)
    a_date = d.add_days(time - 1)

    return a_date
```

G Code for test_driver.py

```
## @file test_driver.py
#  @author Utsharga Rozario
#  @brief Provides test samples for pos_adt.py and date_adt,py
#  @date 20/1/2020

from date_adt import *
from pos_adt import *


######Test Cases for date_adt.py#######

# @brief Tests the day method of the DateT class
# @details Checks for equality between actual and expected values for
#          the DateT day method
def test_day():
  global testTot, passed
  testTot += 1
  try:
    assert date1.day() == 1
    assert date2.day() == 28
    assert date3.day() == 29
    assert date4.day() == 31
    assert date5.day() == 1
    passed += 1
    print(”day test PASSED.”)
  except AssertionError:
    print(”day test FAILED.”)

# @brief Tests the month method of the DateT class
# @details Checks for equality between actual and expected values for
#          the DateT month method
def test_month():
  global testTot, passed
  testTot += 1
  try:
    assert date1.month() == 1
    assert date2.month() == 2
    assert date3.month() == 2
    assert date4.month() == 12
    assert date5.month() == 3
    passed += 1
    print(”month test PASSED.”)
  except AssertionError:
    print(”month test FAILED.”)

# @brief Tests the year method of the DateT class
# @details Checks for equality between actual and expected values for
#          the DateT year method
def test_year():
  global testTot, passed
  testTot += 1
  try:
    assert date1.year() == 22
    assert date2.year() == 2222
    assert date3.year() == 2000
    assert date4.year() == 2
    assert date5.year() == 10000
    passed += 1
    print(”year test PASSED.”)
  except AssertionError:
    print(”year test FAILED.”)

# @brief Tests the next method of the DateT class
# @details Checks for equality between actual and expected values for
#          the DateT next method
def test_next():
  global testTot, passed
  testTot += 1
  try:
    assert (date1.next().day() == 2 and date1.next().month() == 1 and date1.next().year() == 22)
    assert (date2.next().day() == 1 and date2.next().month() == 3 and date2.next().year() == 2222)
    assert (date3.next().day() == 1 and date3.next().month() == 3 and date3.next().year() == 2000)
    assert (date4.next().day() == 1 and date4.next().month() == 1 and date4.next().year() == 3)
    assert (date5.next().day() == 2 and date5.next().month() == 3 and date5.next().year() == 10000)
    passed += 1
    print(”next test PASSED.”)
  except AssertionError:
    print("next test FAILED.")

# @brief Tests the prev method of the DateT class
# @details Checks for equality between actual and expected values for
#          the DateT prev method
def test_prev():
  global testTot, passed
  testTot += 1
  try:
    assert (date1.prev().day() == 31 and date1.prev().month() == 12 and date1.prev().year() == 21)
    assert (date2.prev().day() == 27 and date2.prev().month() == 2 and date2.prev().year() == 2222)
    assert (date3.prev().day() == 28 and date3.prev().month() == 2 and date3.prev().year() == 2000)
    assert (date4.prev().day() == 30 and date4.prev().month() == 12 and date4.prev().year() == 2)
    assert (date5.prev().day() == 29 and date5.prev().month() == 2 and date5.prev().year() == 10000)
    passed += 1
    print("prev test PASSED.")
  except AssertionError:
    print("prev test FAILED.")

# @brief Tests the before method of the DateT class
# @details Checks for equality between actual and expected values for
#          the DateT before method
def test_before():
  global testTot, passed
  testTot += 1
  try:
    assert date1.before(DateT(1,1,23))
    assert date2.before(DateT(31,12,3000))
    assert (date3.before(DateT(29,2,2000)) == False)
    assert date4.before(DateT(12,10,213))
    assert date5.before(DateT(29,3,10000))
    passed += 1
    print("before test PASSED.")
  except AssertionError:
    print("before test FAILED.")

# @brief Tests the after method of the DateT class
# @details Checks for equality between actual and expected values for
#          the DateT after method
def test_after():
  global testTot, passed
  testTot += 1
  try:
    assert date1.after(DateT(31,12,21))
    assert date2.after(DateT(4,7,2000))
    assert (date3.after(DateT(12,6,3000)) == False)
    assert date4.after(DateT(30,12,2))
    assert date5.after(DateT(1,1,1))
    passed += 1
    print("after test PASSED.")
  except AssertionError:
    print("after test FAILED.")

# @brief Tests the equal method of the DateT class
# @details Checks for equality between actual and expected values for
#          the DateT equal method
def test_equal():
  global testTot, passed
  testTot += 1
  try:
    assert date1.equal(DateT(1,1,22))
    assert date2.equal(DateT(28,2,2222))
    assert date3.equal(DateT(29,2,2000))
    assert date4.equal(DateT(31,12,2))
    assert date5.equal(DateT(1,3,10000))
    passed += 1
    print("equal test PASSED.")
  except AssertionError:
    print("equal test FAILED.")

# @brief Tests the add_days method of the DateT class
# @details Checks for equality between actual and expected values for
#          the DateT add_days method
def test_add_days():
  global testTot, passed
  testTot += 1
  try:
    assert (date1.add_days(31).day() == 1 and date1.add_days(31).month() == 2 and
        date1.add_days(31).year() == 22)
    assert (date2.add_days(365).day() == 28 and date2.add_days(365).month() == 2 and
         date2.add_days(365).year() == 2223)
    assert (date3.add_days(307).day() == 1 and date3.add_days(307).month() == 1 and
         date3.add_days(307).year() == 2001)
    assert (date4.add_days(90000).day() == 30 and date4.add_days(90000).month() == 5 and
         date4.add_days(90000).year() == 249)
    assert (date5.add_days(70).day() == 10 and date5.add_days(70).month() == 5 and
         date5.add_days(70).year() == 10000)
    passed += 1
    print(”add_days test PASSED.”)
  except AssertionError:
    print(”add_days test FAILED.”)

# @brief Tests the days_between method of the DateT class
# @details Checks for equality between actual and expected values for
#          the DateT days_between method
def test_days_between():
  global testTot, passed
  testTot += 1
  try:
    assert date1.days_between(DateT(1,3,22)) == 59
    assert date2.days_between(DateT(31,12,3000)) == 284465
    assert date3.days_between(DateT(1,3,2000)) == 1
    assert date4.days_between(DateT(31,12,3)) == 365
    assert date5.days_between(DateT(1,3,10000)) == 0
    passed += 1
    print(”days_between test PASSED.”)
  except AssertionError:
    print(”days_between test FAILED.”)


testTot = 0
passed = 0

date1 = DateT(1,1,22)
date2 = DateT(28,2,2222)
date3 = DateT(29,2,2000)
date4 = DateT(31,12,2)
date5 = DateT(1,3,10000)

print(”Test Cases for date_adt.py:”)
test_day()
test_month()
test_year()
test_next()
test_prev()
test_before()
test_after()
test_equal()
test_add_days()
test_days_between()

print (passed, ”of”, testTot, ”date_adt.py tests passed.”)

######Test Cases for pos_adt.py#######

# for testing floating point equality after arithmetic
def isClose(a, b):
    return (abs(a - b) <= 0.1*(abs(a)))

# @brief Tests the lat method of the GPosT class
# @details Checks for equality between actual and expected values for
#          the GPosT lat method
def test_lat():
  global testTotPos, passedPos
  testTotPos += 1
  try:
    assert pos1.lat() == -90.00
    assert pos2.lat() == 90.00
    assert pos3.lat() == 0.00
    assert pos4.lat() == -53.1236739687
    assert pos5.lat() == 45
    passedPos += 1
    print(”lat test PASSED.”)
  except AssertionError:
    print(”lat test FAILED.”)

# @brief Tests the long method of the GPosT class
# @details Checks for equality between actual and expected values for
#          the GPosT l ongmethod
```

```
def test_long():
  global testTotPos, passedPos
  testTotPos += 1
  try:
    assert pos1.long() == -180.00
    assert pos2.long() == 180.00
    assert pos3.long() == 0.00
    assert pos4.long() == 31.5463673564
    assert pos5.long() == -87
    passedPos += 1
    print("long test PASSED.")
  except AssertionError:
    print("long test FAILED.")

# @brief Tests the west_of method of the GPosT class
# @details Checks for equality between actual and expected values for
#          the GPosT west_of method
def test_west_of():
  global testTotPos, passedPos
  testTotPos += 1
  try:
    assert (pos1.west_of(GPosT(-90.00, -179.9999999999)) == True)
    assert (pos2.west_of(GPosT(90.00, 179.9999999999)) == False)
    assert (pos3.west_of(GPosT(0.0000000001, 32.0000000001)) == True)
    assert (pos4.west_of(GPosT(67.70497335, 29.904735907450)) == False)
    assert (pos5.west_of(GPosT(90.00, -180.00)) == False)
    passedPos += 1
    print("west_of test PASSED.")
  except AssertionError:
    print("west_of test FAILED.")

# @brief Tests the north_of method of the GPosT class
# @details Checks for equality between actual and expected values for
#          the GPosT north_of method
def test_north_of():
  global testTotPos, passedPos
  testTotPos += 1
  try:
    assert (pos1.north_of(GPosT(-89.9999999999, -180.00)) == False)
    assert (pos2.north_of(GPosT(89.9999999999, 180.00)) == True)
    assert (pos3.north_of(GPosT(32.0000000001, 0.0000000001)) == False)
    assert (pos4.north_of(GPosT(-67.097060756709, 29.904735907450)) == True)
    assert (pos5.north_of(GPosT(-90.00, -180.00)) == True)
    passedPos += 1
    print("north_of test PASSED.")
  except AssertionError:
    print("north_of test FAILED.")

# @brief Tests the equal method of the GPosT class
# @details Checks for equality between actual and expected values for
#          the GPosT equal method
def test_equal_pos():
  global testTotPos, passedPos
  testTotPos += 1
  try:
    assert pos1.equal(GPosT(-89.9999999999, -179.9999999999)) == True
    assert pos2.equal(GPosT(90.00, 180.00)) == True
    assert pos3.equal(GPosT(0.00, 32.9098678598)) == False
    assert pos4.equal(GPosT(-53.12367370058, 31.55383614889)) == True
    assert pos5.equal(GPosT(45.00645172973, -86.99111585795)) == False
    passedPos += 1
    print("equal test PASSED.")
  except AssertionError:
    print("equal test FAILED.")

# @brief Tests the move method of the GPosT class
# @details Checks for equality between actual and expected values for
#          the GPosT move method
def test_move():
  global testTotPos, passedPos
  testTotPos += 1
  pos1m = GPosT(pos1.lat(), pos1.long())
  pos2m = GPosT(pos2.lat(), pos2.long())
  pos3m = GPosT(pos3.lat(), pos3.long())
  pos4m = GPosT(pos4.lat(), pos4.long())
  pos5m = GPosT(pos5.lat(), pos5.long())
  pos1m.move(90.00, 100000)
  pos2m.move(360,100)
  pos3m.move(45,3564)
  pos4m.move(-79,10000)
  pos5m.move(0,0)
  try:
    assert (isClose(pos1m.lat(), 89.3216) and isClose(pos1m.long(), -90.0000))
    assert (isClose(pos2m.lat(), 89.1007) and isClose(pos2m.long(), 90.0000))
    assert (isClose(pos3m.lat(), 22.0399) and isClose(pos3m.long(), 023.8808))
    assert (isClose(pos4m.lat(), 06.3018) and isClose(pos4m.long(),  -049.4131))
    assert (isClose(pos5m.lat(), 45.00) and isClose(pos5m.long(),  -087.00))
    passedPos += 1
    print("move test PASSED.")
  except AssertionError:
    print("move test FAILED.")

# @brief Tests the distance method of the GPosT class
# @details Checks for equality between actual and expected values for
#          the GPosT distance method
def test_distance():
  global testTotPos, passedPos
  testTotPos += 1
  try:
    assert isClose(pos1.distance(GPosT(90,180)),20020)
    assert isClose(pos2.distance(GPosT(-56.4639764,-76.4875)), 16290)
    assert isClose(pos3.distance(GPosT(87.9467, 80.7486)), 9971)
    assert isClose(pos4.distance(GPosT(90, -180)), 15910)
    assert isClose(pos5.distance(GPosT(45.01, -87)), 1.112)
    passedPos += 1
    print("distance test PASSED.")
  except AssertionError:
    print("distance test FAILED.")

# @brief Tests the arrival_date method of the GPosT class
# @details Checks for equality between actual and expected values for
#          the GPosT arrival_date method
def test_arrival_date():
  global testTotPos, passedPos
  testTotPos += 1
  date1 = pos1.arrival_date(GPosT(90,180), DateT(1,1,2000), 54.5)
  date2 = pos2.arrival_date(GPosT(-56.4639764,-76.4875), DateT(1,2,2000), 212.59123)
  date3 = pos3.arrival_date(GPosT(87.9467, 80.7486), DateT(1,2,2000), 9970.84535815507)
  date4 = pos4.arrival_date(GPosT(90, -180), DateT(1,2,2000), 16000)
  date5 = pos5.arrival_date(GPosT(45.01, -87), DateT(1,2,2000), 0.0371)
  try:
    assert date1.day() == 2 and date1.month() == 1 and date1.year() == 2001
    assert date2.day() == 17 and date2.month() == 4 and date2.year() == 2000
    assert date3.day() == 1 and date3.month() == 2 and date3.year() == 2000
    assert date4.day() == 1 and date4.month() == 2 and date4.year() == 2000
    assert date5.day() == 1 and date5.month() == 3 and date5.year() == 2000
    passedPos += 1
    print("arrival_date test PASSED.")
  except AssertionError:
    print("arrival_date test FAILED.")

pos1 = GPosT(-90.00, -180.00)
pos2 = GPosT(90.00, 180.00)
pos3 = GPosT(0.00, 0.00)
pos4 = GPosT(-53.1236739687, 31.5463673564)
pos5 = GPosT(45.00, -87.00)

testTotPos = 0
passedPos = 0

print("Test Cases for pos_adt.py:")

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

print ( passedPos, "of", testTotPos, "pos_adt.py tests passed.")
```

H Code for Partner's date_adt.py

```python
## @file date_adt.py
#  @author Waseef Nayeem
#  @brief DateT source code file, contains the implementation of the DateT ADT used to represent
     Gregorian calendar dates.
#  @date 2020-01-20


## @brief An ADT that represents a date on the Gregorian calendar.
#  @details An ADT that represents a date on the Gregorian calendar.
#  It also implements various functions related to date calculations
#  Some implementation assumptions that were made include not supporting BCE years. I also chose to
     include
#  parameter validity checking even though it was not explicitly required.
class DateT:

    ## @brief DateT constructor
    #  @details Constructor that builds the DateT object from day, month and year parameters
    #  @param d Day
    #  @param m Month
    #  @param y Year
    #  @throws ValueError Error raised if specified day, month or year are invalid
    def __init__(self, d, m, y):
        self.days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

        if y < 0:
            raise ValueError('Invalid year: {}'.format(y))

        if m < 1 or m > 12:
            raise ValueError('Invalid month: {}'.format(m))

        if d < 1 or d > 28 and self.days_in_month[m - 1] == 28:
            if not (d == 29 and ((y % 4 == 0 and y % 100 != 0) or (y % 400 == 0))):
                raise ValueError('Invalid day {} for month {} of year {}'.format(d, m, y))
        elif d < 1 or d > self.days_in_month[m - 1]:
            raise ValueError('Invalid day {} for month {} of year {}'.format(d, m, y))

        self.__d = d
        self.__m = m
        self.__y = y

    ## @brief Getter for day variable
    #  @return Day variable
    def day(self):
        return self.__d

    ## @brief Getter for month variable
    #  @return Month variable
    def month(self):
        return self.__m

    ## @brief Getter for year variable
    #  @return Year variable
    def year(self):
        return self.__y

    ## @brief Returns a DateADT object that is one day ahead of current object
    #  @return DateADT object one day after current
    def next(self):
        next_d = self.__d + 1
        next_m = self.__m
        next_y = self.__y

        if self.__d == 28 and self.days_in_month[self.__m - 1] == 28:
            if not ((self.__y % 4 == 0 and self.__y % 100 != 0) or (self.__y % 400 == 0)):
                next_d = 1
                next_m = self.__m + 1
        elif self.__d == 29 and self.days_in_month[self.__m - 1] == 28:
            next_d = 1
            next_m = self.__m + 1
        elif self.__d == 30 and self.days_in_month[self.__m - 1] == 30:
            next_d = 1
            next_m = self.__m + 1
        elif self.__d == 31 and self.days_in_month[self.__m - 1] == 31:
            next_d = 1
            next_m = self.__m + 1
            if self.__m == 12:
```

```
            next_m = 1
            next_y = self.__y + 1

    return DateT(next_d, next_m, next_y)

## @brief Returns a DateADT object that is one day behind the current object
#  @return DateADT object one day before current
def prev(self):
    prev_d = self.__d - 1
    prev_m = self.__m
    prev_y = self.__y

    if self.__d == 1 and self.days_in_month[self.__m - 2] == 28:
        prev_m = self.__m - 1
        if (self.__y % 4 == 0 and self.__y % 100 != 0) or (self.__y % 400 == 0):
            prev_d = 29
        else:
            prev_d = 28
    elif self.__d == 1 and self.days_in_month[self.__m - 2] == 30:
        prev_d = 30
        prev_m = self.__m - 1
    elif self.__d == 1 and self.days_in_month[self.__m - 2] == 31:
        prev_d = 31
        prev_m = self.__m - 1
        if self.__m == 1:
            prev_m = 12
            prev_y = self.__y - 1

    return DateT(prev_d, prev_m, prev_y)

## @brief Comparison function that determines if the current object is before the passed date
   object
#  @param date DateT object to be compared against
#  @return True if the current date is before date parameter, False otherwise
def before(self, date):
    if self.__y < date.year():
        return True
    elif self.__y == date.year() and self.__m < date.month():
        return True
    elif self.__y == date.year() and self.__m == date.month() and self.__d < date.day():
        return True

    return False

## @brief Comparison function that determines if the current object is after the date parameter
#  @param date DateT object to be compared against
#  @return True if the current date is after date parameter, False otherwise
def after(self, date):
    if self.__y > date.year():
        return True
    elif self.__y == date.year() and self.__m > date.month():
        return True
    elif self.__y == date.year() and self.__m == date.month() and self.__d > date.day():
        return True

    return False

## @brief Comparison function that determines if the current object is same date as the passed
   parameter
#  @param date DateT object to be compared against
#  @return True if the current date is identical to the date parameter, False otherwise
def equal(self, date):
    return date.day() == self.__d and date.month() == self.__m and date.year() == self.__y

## @brief Function that returns a DateT object N number of days after the current date object
#  @param n Number of days to add to current date object
#  @throws ValueError Raises error if n is negative
#  @return DateADT object N days later
def add_days(self, n):
    if n < 0:
        raise ValueError(”Cannot add negative number of days”)

    new_day = DateT(self.__d, self.__m, self.__y)

    for i in range(n):
        new_day = new_day.next()

    return new_day
    ## @brief Calculates the number of days between the current date object and a given date object
        parameter
    #  @param date Date object used to calculate days between current date object
    #  @return Integer value representing the number of days between the two dates
    def days_between(self, date):
        counter = 0

        before_date = self if self.before(date) else date
        after_date = self if self.after(date) else date

        while not after_date.equal(before_date):
            before_date = before_date.next()
            counter += 1

        return counter
```

I Code for Partner's pos_adt.py

```
## @file pos_adt.py
#  @author Waseef Nayeem
#  @brief GPosT source code file, contains the implementation of the GPosT ADT used to represent
     geographic coordinates.
#  @date 2020-01-20

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


## @brief An ADT that represents geographic coordinates.
#  @details An ADT that represents geographic coordinates as well as various functions for geographic
     calculations,
#  such as distance between two points.
#  Some implementation assumptions that were made include constraining latitude and longitude values
     to valid range
#  instead of converting out of range values.
class GPosT:

    ## @brief GPosT constructor
    #  @details Constructor that builds the GPosT object from latitude and longitude values
    #  @param lat Latitude
    #  @param long Longitude
    #  @throws ValueError Error raised if latitude or longitude values are out of the valid range
    def __init__(self, lat, long):
        if lat < -90 or lat > 90:
            raise ValueError("Invalid latitude: {}".format(lat))

        if long < -180 or lat > 180:
            raise ValueError("Invalid longitude: {}".format(lat))

        self.__lat = lat
        self.__long = long

    ## @brief Getter for latitude variable
    #  @return Latitude variable
    def lat(self):
        return self.__lat

    ## @brief Getter for longitude variable
    #  @return Longitude variable
    def long(self):
        return self.__long

    ## @brief Comparison function that determines if the current position is west of the passed
        position object
    #  @param p GPosT object to be compared against
    #  @return True if the current position is west of passed position parameter, False otherwise
    def west_of(self, p):
        return self.__long < p.long()

    ## @brief Comparison function that determines if the current position is north of the passed
        position object
    #  @param p GPosT object to be compared against
    #  @return True if the current position is north of passed position parameter, False otherwise
    def north_of(self, p):
        return self.__lat > p.lat()
```

```
## @brief Comparison function that determines if the current position is ”equal to” the passed
    position object.
#  This is defined as the two positions being less than 1 km in distance from each other.
#  @param p GPosT object to be compared against
#  @return True if the current position is within 1 km of passed position parameter, False
    otherwise
def equal(self, p):
    return self.distance(p) < 1.0

## @brief Function that moves the current position by a certain distance along a specified bearing.
#  @details Mutator function that moves the current function a specified distance in km along a
    given bearing in degrees.
#  Formula courtesy of https://www.movable-type.co.uk/scripts/latlong.html
#  @param b Bearing in degrees
#  @param d Distance in kilometres (km)
def move(self, b, d):
    r = 6371e3
    d *= 1000
    phi1 = radians(self.__lat)
    lambda1 = radians(self.__long)
    b = radians(b)
    phi2 = asin(sin(phi1) * cos(d / r) + cos(phi1) * sin(d / r) * cos(b))
    lambda2 = lambda1 + atan2(sin(b) * sin(d / r) * cos(phi1), cos(d / r) - sin(phi1) * sin(phi2))

    self.__lat = degrees(phi2)
    self.__long = degrees(lambda2)

## @brief Calculates great-circle distance between two coordinates using the haversine formula.
#  Source: https://www.movable-type.co.uk/scripts/latlong.html
#  @param p GPosT object to calculated distance between
#  @return Distance between the two positions (in km)
def distance(self, p):
    r = 6371e3  # metres
    phi1 = radians(self.__lat)
    phi2 = radians(p.lat())
    delta_phi = radians(p.lat() - self.__lat)
    delta_lambda = radians(p.long() - self.__long)

    a = sin(delta_phi / 2) * sin(delta_phi / 2) + cos(phi1) * cos(phi2) * sin(delta_lambda / 2) *
        sin(delta_lambda / 2)
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    dist = r * c
    return dist / 1000

## @brief Calculates time taken to travel from current position to end position and returns
    arrival date.
#  @details Given an end point, starting date and speed, this function calculates the time it
    takes to travel from
#  start position (current object) to end position while traveling at a specified speed. It then
    calculates the
#  arrival date from the time taken and the starting date.
#  @param p GPosT object for the end position
#  @param d DateT object for starting date
#  @param s Travel speed in km/day
#  @throws ValueError Raises error if speed is negative
#  @return DateT object for arrival date
def arrival_date(self, p, d, s):
    if s < 0:
        raise ValueError(”Speed must be positive”)

    dist = self.distance(p)
    days = dist / s

    for i in range(int(days)):
        d = d.next()

    return d
```