**Lab 4: Dictionaries and Classes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name: | Shaheer Ziya |  | University Number: | 3035946760 |

**Exercise 1: Canadian Postal Codes**

AIM:

In Canada, the first, third, and fifth characters in the postal code are letters while the second, fourth, and sixth characters are digits. We can determine the province or territory in which an address resides from the first character of its postal code as shown in the following table. No valid postal codes currently begin with D, F, I, O, Q, U, W, or Z.

|  |  |
| --- | --- |
| Province/Territory | First Character(s) |
| Newfoundland and Labrador | A |
| Nova Scotia | B |
| Prince Edward Island | C |
| New Brunswick | E |
| Quebec | G, H, and J |
| Ontario | K, L, M, N, and P |
| Manitoba | R |
| Saskatchewan | S |
| Alberta | T |
| British Columbia | V |
| Nunavut or Northwest Territories | X |
| Yukon | Y |

The second character in a postal code identifies whether the address is rural or urban. If that character is a 0, then the address is rural; otherwise, it is urban. Write a Python program that prompts a postal code from the user and displays the province or territory associated with it, along with whether the address is urban or rural. Your program should use a dictionary to map from the first character of the postal code to the province or territory name. Here are the sample input and output of the program:

Enter a Canadian postal code: X0D 0A0

The postal code is for a rural address in Nunavut or Northwest Territories.

Enter a Canadian postal code: H8Z 7R6

The postal code is for an urban address in Quebec.

Enter a Canadian postal code: ABB 2L5

The second character in the postal code must be a digit!

Enter a Canadian postal code: Q5N 8P4

The first character in the postal code is invalid!

It cannot be D, F, I, O, Q, U, W, or Z!

ALGORITHM:

|  |
| --- |
| * Initialize a python dictionary which maps characters to territories * Obtain the postal code from the user * Extract the 1st and 2nd letters from the postal code * Find the image of the 1st letter through the dictionary mapping * Determine if the rurality is urban or not depending on if the 2nd letter is non-zero or not. * Handle cases where the first two letters can be invalid and output the result in the desired way using f-strings |

PROGRAM:

|  |
| --- |
| # Canada's Postal Codes  # Created by Shaheer Ziya  # Postal Code Letters & Their Corresponding Territories  postalCodes = {  'A': "Newfoundland and Labrador",  'B': "Nova Scotia",  'C': "Prince Edward Island",  'D': "Invalid",  'E': "New Brunswick",  'F': "Invalid",  'G': "Quebec",  'H': "Quebec",  'I': "Invalid",  'J': "Quebec",  'K': "Ontario",  'L': "Ontario",  'M': "Ontario",  'N': "Ontario",  'O': "Invalid",  'P': "Ontario",  'Q': "Invalid",  'R': "Manitoba",  'S': "Saskatchewan",  'T': "Alberta",  'U': "Invalid",  'V': "British Columbia",  'W': "Invalid",  'X': "Nunavut or Northwest Territories",  'Y': "Yukon",  'Z': "Invalid"  }  def main():  # Obtain postal code from user  code = input("Enter a Canadian postal code: ")  # Obtain first letter of postal code  firstLetter = code[0]  # Obtain corresponding territory  territory = postalCodes[firstLetter]  # Determine if the area is rural or urban  secondLetter = code[1]  # 0 -> Rural, 1-9 -> Urban  if secondLetter == '0':  rurality = "a rural"  elif (not secondLetter.isdigit()):  rurality = "invalid"  else:  rurality = "an urban"  # Error Handling & I/O  # If the postal code's letter is invalid  if territory == "Invalid":  print("The first character in the postal code is invalid!")  print("It cannot be D, F, I, O, Q, U, W, or Z!")  # If the postal code's digit is invalid  elif rurality == "invalid":  print("The second character in the postal code must be a digit!")  else:  print(f"The postal code is for {rurality} address in {territory}.")  main() |

OUTPUT:

|  |
| --- |
| Text  Description automatically generated |

**Exercise 2: Converting Floating-Point Values to Monetary Amounts**

AIM:

Write a Python class MoneyFmt for converting floating-point values to monetary amounts. The class has one data attribute of type float for the floating-point value called value. It also provides the following methods:

1. \_ \_init\_ \_(self, value=0.) for creating a new instance of the class with the floating-point value initialized,
2. update(self, value=None) for updating the floating-point value,
3. \_ \_str\_ \_(self) for displaying the floating-point value as a monetary amount to two decimal places with the correct sign,
4. \_ \_repr\_ \_(self) for returning a string such that eval applied to the string recreates the instance,
5. isnonzero(self) for checking if the floating-point value is non-zero where a value with absolute value less than 0.005 is regarded as zero.

Here are the sample input and output for using the class MoneyFmt:

>>> cash = MoneyFmt(135.79)

>>> cash

MoneyFmt(135.79)

>>> print(cash)

$135.79

>>> cash.update(5000000.2468)

>>> cash

MoneyFmt(5000000.2468)

>>> print(cash)

$5,000,000.25

>>> cash.update(-1386.42)

>>> print(cash)

-$1,386.42

>>> cash.isnonzero()

True

>>> cash.update(-0.0049)

>>> cash.isnonzero()

False

ALGORITHM:

|  |
| --- |
| * Define the class * Initialize the object with an instance variable value set to the passed value * Write the required methods for the class |

PROGRAM:

|  |
| --- |
| # Convert Floating Point to Monetary Values  # Created by Shaheer Ziya  # A class to convert floating point values to monetary values  class MoneyFmt:  def \_\_init\_\_(self, value: "float" = 0) -> None:  '''Initialize the MoneyFmt object with the passed value argument'''  self.value = value    def \_\_str\_\_(self) -> str:  '''Return a human readable string representation of the MoneyFmt object'''  # Add commas & round to 2 decimal places  return f"${self.value:,.2f}"    def \_\_repr\_\_(self) -> str:  '''Return a string representation of the MoneyFmt object that instantiates the object under eval()'''  return f"MoneyFmt({self.value})"    def update(self, value: "float" = None) -> None:  '''Update the existing value of the MoneyFmt object'''  self.value = value    def isnonzero(self):  '''Return True if the magnitude of value of the MoneyFmt object is not less than 0.005'''  if abs(self.value) >= 0.005:  return True  else:  return False  def main():  cash = MoneyFmt(135.79)    print(repr(cash))  print(str(cash))    cash.update(5000000.2468)    print(repr(cash))  print(str(cash))  cash.update(-1386.42)  print(str(cash))  print(cash.isnonzero())  cash.update(-0.0049)  print(cash.isnonzero())  main() |

OUTPUT:

|  |
| --- |
| Text  Description automatically generated |

**Exercise 3: Queue**

AIM:

A queue behaves like a real-world queue such as a checkout line in a supermarket. The customer at the front of the line is serviced first and other customers can only enter at the back of the line. So the first customer in line is always the first one to be serviced (i. e. it follows the First In First Out principle). Write a Python class Queue for representing queues of integers. The class has two data attributes qlst and maxsize where qlst is the list holding the elements of the queue and maxsize is the size limit for the queue. It also provides the following methods:

1. \_ \_init\_ \_(self, maxsize=10) for initializing the queue with an empty list and a size limit,
2. enqueue(self, item) for adding an element to the back of a non-full queue,
3. dequeue(self) for removing an element from the front of a non-empty queue,

(Hint: use the pop function)

1. isempty(self) for checking whether the queue is empty,
2. isfull(self) for checking whether the queue is full (i.e. whether the number of elements in the queue is equal to maxsize),
3. count(self) for returning the number of elements in the queue,
4. \_ \_str\_ \_(self) for displaying all the elements in the queue.

Note that in the method enqueue, the provided element would be added to the queue only if it is an integer. Here are the sample input and output for using the class Queue:

>>> a = Queue(3)

>>> a.isempty()

True

>>> a.enqueue(1.2)

You can only add integers to the queue!

>>> a.enqueue(1)

>>> a.enqueue(2)

>>> a.enqueue(3)

>>> a.enqueue(4)

The queue is full and no new element can be added!

>>> a.isfull()

True

>>> a.count()

Number of elements in the queue = 3

>>> a.dequeue()

1

>>> print(a)

Element 1 in the queue = 2

Element 2 in the queue = 3

>>> a.dequeue()

2

>>> a.dequeue()

3

>>> a.dequeue()

The queue is empty and there is no front element!

ALGORITHM:

|  |
| --- |
| * Define the Queue class with its required methods * The docstrings of the methods should be self-explanatory |

PROGRAM:

|  |
| --- |
| # The Definition For the Queue Class  # Created by Shaheer Ziya  # A class to represent a queue using FIFO (First In First Out)  class Queue:  def \_\_init\_\_(self, max\_size: int = 10) -> None:  self.queue = []  self.max\_size = max\_size  def \_\_str\_\_(self) -> str:  elementsString = ""  for idx, element in enumerate(self.queue):  elementsString += f"Element {idx+1} in the queue = {element}\n"    # Don't return last newline  return elementsString[:-1]    def enqueue(self, item) -> None:  ''' Enter an integer to a queue '''  # Ensure only integer values are added to the queue  if type(item) != int:  print("You can only add integers to the queue!")  # See if the queue is full, if it is not then add to the queue  elif len(self.queue) < self.max\_size:  self.queue.append(item)  else:  print("The queue is full and no new element can be added!")    def dequeue(self):  ''' Remove an integer from the queue and return the removed integr '''  # See if the queue is empty, if it is not then remove from the queue  if len(self.queue) > 0:  # Return the removed element  return self.queue.pop(0)  else:  print("The queue is empty and there is no front element!")    def isempty(self) -> bool:  ''' Check if the queue is empty '''  if len(self.queue) == 0:  return True  else:  return False    def isfull(self) -> bool:  ''' Check if the queue is full '''  if len(self.queue) == self.max\_size:  return True  else:  return False    def count(self) -> int:  ''' Return the number of items in the queue '''    print(f"Number of elements in the queue = {len(self.queue)}")  return len(self.queue)      def main():  a = Queue(3)    print(a.isempty())    a.enqueue(1.2)  a.enqueue(1)  a.enqueue(2)  a.enqueue(3)  a.enqueue(4)  print(a.isfull())  a.count()  print(a.dequeue())  print(a)  print(a.dequeue())  print(a.dequeue())  print(a.dequeue())    main() |

OUTPUT:

|  |
| --- |
| Text  Description automatically generated |

**Exercise 4: Quadrilaterals, Trapezoids, Parallelograms, and Squares**

AIM:

Suppose a Python class Point is defined in the program point.py as follows:

class Point:

""" A class representing geometric points """

def \_ \_init\_ \_(self, x, y):

""" Initialize the point with its x, y coordinates """

self.x = x

self.y = y

def getx(self):

""" Return the x coordinate of the point """

return self.x

def gety(self):

""" Return the y coordinate of the point """

return self.y

def \_ \_str\_ \_(self):

""" Return a string representation of the point """

return "({:.1f}, {:.1f})".format(self.x, self.y)

Write a Python class Quadrilateral for representing quadrilaterals. This class has four data attributes pt1, pt2, pt3, pt4 of class Points for the four endpoints of the quadrilateral. In addition to the constructor for initializing the instance, it provides the methods getpt1(self), getpt2(self), getpt3(self), getpt4(self) for returning the individual endpoints of the quadrilateral, getcoorstr(self) for returning a string containing the coordinates of the endpoints of the quadrilateral, and \_ \_str\_ \_(self) for returning a string representation of the quadrilateral with the coordinates of its endpoints obtained by calling the method getcoorstr. Next, write a Python class Trapezoid for representing trapezoids by inheriting from the class Quadrilateral. It overrides the method \_ \_str\_ \_(self)for returning a string representation of the trapezoid with the coordinates of its endpoints, its height, and its area. It also adds the methods getheight(self) for returning the height of the trapezoid, getsumoftwosides(self) for returning the sum of the length of the two parallel sides of the trapezoid, and getarea(self) for returning the area of the trapezoid, assuming the points on each parallel side to have the same *y* coordinates. Next, write a Python class Parallelogram for representing parallelograms by inheriting from the class Trapezoid. It overrides the method \_ \_str\_ \_(self)for returning a string representation of the parallelogram with the coordinates of its endpoints, its width, its height, and its area. It also adds the methods getwidth(self) for returning the width of the parallelogram. Finally, write a Python class Square for representing squares by inheriting from the class Parallelogram. It overrides the method \_ \_str\_ \_(self)for returning a string representation of the square with the coordinates of its endpoints, its side, and its area. Assume the constructors of these classes are called with valid input. Here are the sample input and output for using these classes:

>>> a = Quadrilateral(1.1, 1.2, 6.6, 2.8, 6.2, 9.9, 2.2, 7.4)

>>> print(a)

Coordinates of the Quadrilateral are:

(1.1, 1.2), (6.6, 2.8), (6.2, 9.9), (2.2, 7.4)

>>> b = Trapezoid(0.0, 0.0, 10.0, 0.0, 8.0, 5.0, 3.3, 5.0)

>>> b.getsumoftwosides()

14.7

>>> print(b)

Coordinates of the Trapezoid are:

(0.0, 0.0), (10.0, 0.0), (8.0, 5.0), (3.3, 5.0)

Height = 5.0

Area = 36.75

>>> c = Parallelogram(5.0, 5.0, 11.0, 5.0, 12.0, 20.0, 6.0, 20.0)

>>> print(c)

Coordinates of the Parallelogram are:

(5.0, 5.0), (11.0, 5.0), (12.0, 20.0), (6.0, 20.0)

Width = 6.0

Height = 15.0

Area = 90.00

>>> d = Square(4.0, 0.0, 8.0, 0.0, 8.0, 4.0, 4.0, 4.0)

>>> print(d)

Coordinates of the Square are:

(4.0, 0.0), (8.0, 0.0), (8.0, 4.0), (4.0, 4.0)

Side = 4.0

Area = 16.00

ALGORITHM:

|  |
| --- |
| * Import the point class from point.py * Define 4 Classes that build on each other using inheritance * Add in the required methods for each class * Using super() and formatting methods to print required strings |

PROGRAM:

|  |
| --- |
| # Quadrilaterals, Trapezoids, Parallelograms, and Squares  # Created by Shaheer Ziya  # Import the Point class from the file point.py  from point import Point  # A class representing a geometric quadrilateral  class Quadrilateral:    def \_\_init\_\_(self, \*args) -> None:  # Define the four points of the quadrilateral  self.pt1 = Point(args[0], args[1])  self.pt2 = Point(args[2], args[3])  self.pt3 = Point(args[4], args[5])  self.pt4 = Point(args[6], args[7])    ### Methods to retrieve the coordinates of the quadrilateral ###  def getpt1(self) -> Point:  """ Return the first point of the quadrilateral """  return self.pt1    def getpt2(self) -> Point:  """ Return the second point of the quadrilateral """  return self.pt2    def getpt3(self) -> Point:  """ Return the third point of the quadrilateral """  return self.pt3    def getpt4(self) -> Point:  """ Return the fourth point of the quadrilateral """  return self.pt4    def getcoorstr(self) -> str:  """ Return a string representation of the coordinates of the quadrilateral """  return f"{self.pt1}, {self.pt2}, {self.pt3}, {self.pt4}"    def \_\_str\_\_(self) -> str:  """ Return a string representation of the quadrilateral """  return f"Coordinates of the Quadrilateral are:\n{self.getcoorstr()}"  # A class representing a geometric trapezoid  class Trapezoid(Quadrilateral):  def \_\_init\_\_(self, \*args) -> None:  # Call the constructor of the Quadrilateral class  super().\_\_init\_\_(\*args)    def getheight(self) -> float:  """ Return the height of the trapezoid """  # It has been assumed that the points will be entered in a cyclic order  # And that the parallel sides of the trapezoiod will be parallel to the x-axis  return self.pt3.gety() - self.pt1.gety()  def getsumoftwosides(self) -> float:  """ Return the sum of the two parallel sides of the trapezoid """  # It has been assumed that the points will be entered in a cyclic order  # And that the parallel sides of the trapezoiod will be parallel to the x-axis  return (self.pt2.getx() - self.pt1.getx()) + (self.pt3.getx() - self.pt4.getx())    def getarea(self) -> float:  """ Return the area of the trapezoid """  return (self.getheight() \* self.getsumoftwosides()) / 2    def \_\_str\_\_(self) -> str:  # Adjust for the first line of the string representation  return f"Coordinates of the Trapezoid are:\n{super().\_\_str\_\_()[38:]}" + "\n" + \  f"Height = {self.getheight()}" + "\n" + \  f"Area = {self.getarea()}"  # A class representing a geometric parallelogram  class Parallelogram(Trapezoid):  def \_\_init\_\_(self, \*args) -> None:  super().\_\_init\_\_(\*args)    def getwidth(self) -> float:  """ Return the width of the parallelogram """  # It has been assumed that the points will be entered in a cyclic order  # And that the parallel sides of the parallelogram will be parallel to the x-axis  return self.pt2.getx() - self.pt1.getx()    def \_\_str\_\_(self) -> str:  # Insert the width part into the original string representation  superiorLine = super().\_\_str\_\_()[34:]  newline = superiorLine.find("\n")  Cord\_line = superiorLine[:newline]  Area\_line = superiorLine[superiorLine.find("\nHeight"):]  # Area\_Line =  return "Coordinates of the Parallelogram are:\n" + Cord\_line + f"\nWidth = {self.getwidth()}" + \  Area\_line  # A class representing a geometric square  class Square(Parallelogram):    def \_\_init\_\_(self, \*args) -> None:  super().\_\_init\_\_(\*args)      def getside(self) -> float:  """ Return the side of the square """  # It has been assumed that the points will be entered in a cyclic order  # And that the parallel sides of the square will be parallel to the x-axis  return self.pt2.getx() - self.pt1.getx()      def \_\_str\_\_(self) -> str:  superiorLine = super().\_\_str\_\_()[38:]  Cord\_line = superiorLine[:superiorLine.find("\n")]  Area\_line = superiorLine[superiorLine.find("\nArea"):]  return f"Coordinates of the Square are:\n" + Cord\_line + f"\nSide = {self.getside()}" + Area\_line  def main():    a = Quadrilateral(1.1, 1.2, 6.6, 2.8, 6.2, 9.9, 2.2, 7.4)  print(a)  print()  b = Trapezoid(0.0, 0.0, 10.0, 0.0, 8.0, 5.0, 3.3, 5.0)  print(b.getsumoftwosides())  print(b)  print()  c = Parallelogram(5.0, 5.0, 11.0, 5.0, 12.0, 20.0, 6.0, 20.0)  print(c)  print()  d = Square(4.0, 0.0, 8.0, 0.0, 8.0, 4.0, 4.0, 4.0)  print(d)  main() |

OUTPUT:

|  |
| --- |
| Text  Description automatically generated |