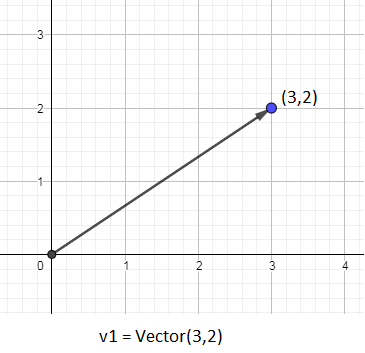
**Question 1:**

**Write a Python class to represent a 2D vector.**

**A vector is basically an arrow that has a magnitude (a length)**

**and a direction (an angle with respect to typically the x axis).**

**It is usually represented as an x,y pair of coordinates,**

**where the origin of the vector is 0,0**

**and the end of the vector is at the listed pair.**

**Write a Python class to represent a Vector. Implement the following behaviour in your Vector class:**

1. **vector addition: If V1 is (x1, y1) and V2 is (x2, y2), the V1+V2 is the vector (x1+x2, y1+y2)**
2. **vector multiplication by an int: if V is (x, y), the V\*n is the vector (x\*n, y\*n), where n is an integer number**
3. **vector subtraction: V1-V2 is the same as V1+(V2\*-1), a vector (x1-x2,y1-y2)**
4. **vector multiplication with another vector: implement the dot product.**

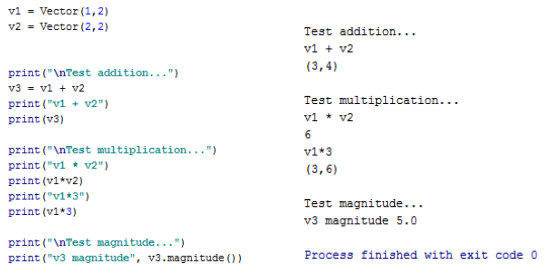
**If V1 is (x1,y1) and V2 is (x2,y2), then V1\*V2 = x1\*x2 + y1\*y2, a scalar.**

**Thus the dot product yields a scalar (number), not a vector**

1. **vector magnitude: The magnitude based on the Pythagorean theorem for a vector V=(x,y) is the square root of (x2 + y2) .**

**Include any other appropriate methods, such a constructor and \_\_str\_\_, and pay attention to naming standards, private/public, etc.**

**Sample output and testing of the class**

****

class Point(object):

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

self.x = x

self.y = y

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

return “{}, {}”.format(self.x, self.y)

class Vector(object):

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

self.point = Point(x, y)

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

return str(self.point)

def \_\_add\_(self, other):

new\_x = self.point.x + other.point.x

new\_y = self.point.y + other.point.y

return Vector(new\_x, new\_y)

def \_\_mul\_\_(self, other):

if type(other) == int:

new\_x = self.point.x \* other

new\_y + self.point\_y \* other

return Vector(new\_x, new\_y)

if type(other) == Vector:

return self.point.x \* other.point.x + self.point.y \* other.point.y

def \_\_sub\_\_(self, other):

return Vector(self.point.x – other.point.x, self.point.y – other.point.y)

# return self + (other \* -1)

def magnitude(self) -> float:

return self.point.x \*\* 2 + self.point.y \*\* 2

def \_\_rmul\_\_(self, other):

return self.\_\_mul\_\_(other)

v1 = Vector(1, 2)

v2 = Vector(2, 2)

v3 = v1 + v2

print(v1.magnitude())

**Question 2:**

**Design a class called TextDocument that reads the contents of a text file only. Create a**

**constructor that takes the path of a text file. Create the following overloaded operators for**

**this class:**

1. **An overloaded “addition” operator that adds (concatenates) two TextDocument objects (appends the second one at the end of the first one) and creates a new text file.**
2. **An overloaded “greater than” operator that compares two TextDocument objects and returns true if the length (number of words) of the first document is greater than the second document, or false otherwise.**
3. **An overloaded “multiplication” operator that multiplies the content of a TextDocument object by an integer number x. The result is the content of the TextDocument object file repeated in the same file x times.**

**Use introspection when implementing your overloaded operators. Create a command line**

**menu with the following functionality:**

1. **Ask for two file names, concatenate the two files and create a new text file.**
2. **Ask for two file names and check if the first file is greater than the second.**
3. **Ask for one file name and an integer number x, and repeat the content of the file x times. The result should be in the same file. Handle all possible error cases, such as for file existence and wrong inputs.**

import string

class TextDocument(object):

def \_\_init\_\_(self, path):

self.path = path

self.file = open(path, “r”)

def \_\_add\_\_(self, other):

file\_addition = open(“addition.txt”, “w+”)

file\_addition.write(“”.join(self.file.readlines() + other.file.readlines()))

return TextDocument(“addition.txt”)

def \_\_gt\_\_(self, other):

words1 = 0

for line in self.file:

words1 += len(line.split())

words2 = 0

for line in other.file:

words2 += len(line.split())

if words1 > words2:

return True

else:

return False

hare\_and\_tortoise = TextDocument(“C:\\Users\\453526\\PycharmProjects\\pythonProject\\hare\_and\_tortoise.txt”)

three\_little\_pigs = TextDocument(“C:\\Users\\453526\\PycharmProjects\\pythonProject\\three\_little\_pigs.txt”)

addition = hare\_and\_tortoise + three\_little\_pigs

if hare\_and\_tortoise > three\_little\_pigs:

print(True)

else:

print(False)

**Question 3:**

**Design a class for a book that an online retailer (such as Amazon.com) might use to keep track of the book. Fields might include the name of the book, publisher, price, author, and ISBN. Think about the methods appropriate for the class that an online retailer might use. Use a dictionary to implement at least one attribute. Include in your class at least two methods available in Python to overload operators. Demonstrate the use of such operators with different instances of your class.**

**Implement a second class puzzle book that inherits from the book class. Add information such as puzzle type, difficulty, sample puzzle, etc.**

Here's an implementation of the Book class that an online retailer could use to keep track of a book:

class Book:

def \_\_init\_\_(self, name, publisher, price, author, isbn):

self.name = name

self.publisher = publisher

self.price = price

self.author = author

self.isbn = isbn

self.attributes = {}

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

return f"{self.name} by {self.author}"

def \_\_repr\_\_(self):

return f"Book({self.name}, {self.publisher}, {self.price}, {self.author}, {self.isbn})"

def \_\_eq\_\_(self, other):

return self.isbn == other.isbn

def \_\_lt\_\_(self, other):

return self.price < other.price

def add\_attribute(self, key, value):

self.attributes[key] = value

def get\_attribute(self, key):

return self.attributes.get(key, None)

Here's an example of how to use the Book class:

book1 = Book("The Great Gatsby", "Scribner", 10.99, "F. Scott Fitzgerald", "9780743273565")

book2 = Book("To Kill a Mockingbird", "Harper Perennial", 8.99, "Harper Lee", "9780446310789")

book1.add\_attribute("language", "English")

book2.add\_attribute("language", "Spanish")

print(book1) # prints "The Great Gatsby by F. Scott Fitzgerald"

print(book2) # prints "To Kill a Mockingbird by Harper Lee"

print(book1 == book2) # prints "False"

print(book1 < book2) # prints "False"

print(book2 < book1) # prints "True"

print(book1.get\_attribute("language")) # prints "English"

print(book2.get\_attribute("publisher")) # prints "Harper Perennial"

Next, let's create a subclass of Book called PuzzleBook that adds additional information about the book:

class PuzzleBook(Book):

def \_\_init\_\_(self, name, publisher, price, author, isbn, puzzle\_type, difficulty, sample\_puzzle):

super().\_\_init\_\_(name, publisher, price, author, isbn)

self.puzzle\_type = puzzle\_type

self.difficulty = difficulty

self.sample\_puzzle = sample\_puzzle

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

return f"{self.name} by {self.author}, {self.puzzle\_type} puzzle book"

def \_\_repr\_\_(self):

return f"PuzzleBook({self.name}, {self.publisher}, {self.price}, {self.author}, {self.isbn}, {self.puzzle\_type}, {self.difficulty}, {self.sample\_puzzle})"

Here's an example of how to use the PuzzleBook class:

puzzle\_book = PuzzleBook("The New York Times Ultimate Crossword Omnibus", "St. Martin's Griffin", 19.99, "Will Shortz", "9780312311231", "Crossword", "Difficult", "Sample puzzle goes here")

print(puzzle\_book) #prints "The New York Times Ultimate Crossword Omnibus by Will Shortz, Crossword puzzle book"

print(puzzle\_book.get\_attribute("language")) # prints None

print(puzzle\_book == book1) # prints False

print(puzzle\_book < book1) # raises a TypeError, as the < operator is not defined between different types

**Question 4:**

**Design a Python class to represent one of the modules that a student (you!) is taking during a semester. Fields might include name, time (day of the week and hour), year, location, and lecturer. Use a dictionary to implement the set of assignments for this module with corresponding deadlines and percentages of overall marks. Include in your Python class methods to update/add assignments.**

**Implement a second class to represent a student (you!). Use composition/aggregation to add different instances of the module class representing the modules that you are taking.**

Here's an implementation of the `Module` class that represents a module a student is taking:

class Module:

def \_\_init\_\_(self, name, time, year, location, lecturer):

self.name = name

self.time = time

self.year = year

self.location = location

self.lecturer = lecturer

self.assignments = {}

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

return f"{self.name} ({self.year})"

def \_\_repr\_\_(self):

return f"Module({self.name}, {self.time}, {self.year}, {self.location}, {self.lecturer})"

def add\_assignment(self, name, deadline, percentage):

self.assignments[name] = {"deadline": deadline, "percentage": percentage}

def update\_assignment(self, name, deadline=None, percentage=None):

if name not in self.assignments:

raise ValueError(f"No assignment named {name}")

if deadline is not None:

self.assignments[name]["deadline"] = deadline

if percentage is not None:

self.assignments[name]["percentage"] = percentage

Here's an example of how to use the `Module` class:

module = Module("Introduction to Python Programming", "Tuesday 9:00am-11:00am", "2023", "Room 101", "Dr. John Smith")

module.add\_assignment("Assignment 1", "2023-09-30", 20)

module.add\_assignment("Assignment 2", "2023-10-31", 30)

module.add\_assignment("Final Project", "2023-11-30", 50)

print(module.assignments)

# prints {"Assignment 1": {"deadline": "2023-09-30", "percentage": 20},

# "Assignment 2": {"deadline": "2023-10-31", "percentage": 30},

# "Final Project": {"deadline": "2023-11-30", "percentage": 50}}

module.update\_assignment("Assignment 2", deadline="2023-11-15")

print(module.assignments["Assignment 2"])

# prints {"deadline": "2023-11-15", "percentage": 30}

Next, let's create a `Student` class that aggregates multiple instances of the `Module` class:

class Student:

def \_\_init\_\_(self, name, modules):

self.name = name

self.modules = modules

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

return f"Student: {self.name}"

def \_\_repr\_\_(self):

return f"Student({self.name}, {self.modules})"

def add\_module(self, module):

self.modules.append(module)

Here's an example of how to use the `Student` class:

module1 = Module("Introduction to Python Programming", "Tuesday 9:00am-11:00am", "2023", "Room 101", "Dr. John Smith")

module2 = Module("Data Structures and Algorithms", "Thursday 2:00pm-4:00pm", "2023", "Room 201", "Dr. Jane Doe")

student = Student("John Doe", [module1])

student.add\_module(module2)

print(student)

# prints "Student: John Doe"

print(student.modules)

# prints [Module(Introduction to Python Programming (2023)), Module(Data Structures and Algorithms (2023))]

**Question 5**

**Extend the class Currency in the file currency\_conversion.py in Week 11 folder. This class is to be instantiated with an amount and currency type (three character currency code), and is used to fetch information from the web. The class details are:**

**\_\_init\_\_: The constructor takes the following arguments (with defaults indicated):**

**a. An amount. Default: 1**

**b. A currency code. If the currency code provided is invalid, set the currency code to "" (an empty string) and the amount to 0. Default: EUR**

**convert\_to: This method takes a single argument, a currency code, with no default. It returns a new Currency object with the new currency code and the converted amount. This method is partially implemented. It uses the urllib.request library for opening URLs. You can see more about this library here** [**https://docs.python.org/3/library/urllib.request.html**](https://docs.python.org/3/library/urllib.request.html)

**The** [**https://exchangerate.host/**](https://exchangerate.host/) **website is used to perform conversions. The URL** [**https://api.exchangerate.host/convert?from=USD&to=EUR**](https://api.exchangerate.host/convert?from=USD&to=EUR) **returns a JSON string:**

**'{"motd":{"msg":"If you or your company use this project or like what we doing, please consider backing us so we can continue maintaining and evolving this project.","url":"https://exchangerate.host/#/donate"},"success":true,"query":{"from":"USD","to":"EUR","amount":1},"info":{"rate":0.965561},"historical":false,"date":"2022-11-16","result":0.965561}'**

**While the URL** [**https://api.exchangerate.host/convert?from=EUR&to=GBP**](https://api.exchangerate.host/convert?from=EUR&to=GBP) **returns the JSON string :**

**{"motd":{"msg":"If you or your company use this project or like what we doing, please consider backing us so we can continue maintaining and evolving this project.","url":"https://exchangerate.host/#/donate"},"success":true,"query":{"from":"EUR","to":"GBP","amount":1},"info":{"rate":0.873867},"historical":false,"date":"2022-11-16","result":0.873867}**

**Python offers a built-in JSON package which can be used to easily manipulate JSON data. You can see a nice introduction here** [**https://www.w3schools.com/python/python\_json.asp**](https://www.w3schools.com/python/python_json.asp)

**For example, it is possible to access the fields of the JSON string above in the following way:**

**import json**

**exchange\_info = json.loads('{"motd":{"msg":"If you or your company use this project or like what we are doing, please consider backing us so we can continue maintaining and evolving this project.", "url":"https://exchangerate.host/#/donate"}, "success":true, "query":{"from":"EUR","to":"GBP","amount":1}, "info":{"rate":0.873867}, "historical":false,"date":"2022-11-16","result":0.873867}')**

**print(exchange\_info["info"]["rate"]) *# 0.873867***

**print(exchange\_info["query"]["from"]) *# EUR***

**print(exchange\_info["date"]) *# 2022-11-16***

**You should extend this method to extract the exchange rate from a JSON string of this type and return a new Currency object with the corrected converted amount.**

**The following operations must be implemented. For each method it should be able to handle operands of type Currency, float, or int (for example, your \_\_add\_\_ method should be able to handle curr1 + curr2, or handle curr1 + 5, or handle curr1 + 2.71).**

**a. \_\_str\_\_: Return the amount and type as a string**

**b. \_\_add\_\_ and \_\_radd\_\_**

**c. \_\_sub\_\_ and \_\_rsub\_\_**

**d. \_\_gt\_\_**

**e. \_\_repr\_\_**

**Include a main program that tests all your methods.**

#!/usr/bin/python3

# Use the exchangeratesapi.io to perform currency conversions.

# https://api.exchangerate.host/convert?from=USD&to=EUR

import urllib.request

import json

class Currency:

"""INSERT A DOC STRING HERE"""

VALID\_CURRENCIES = ['USD', 'EUR', 'GBP']

def \_\_init\_\_(self, amount=1, currency\_type='USD'):

# a quick way of checking for valid currencies

# for a limited subset of valid currencies

if currency\_type in Currency.VALID\_CURRENCIES:

self.amount = amount

self.currency\_type = currency\_type

else:

print("Invalid currency type: %s\n", currency\_type)

self.amount = 0

self.currency\_type = ''

def convert\_to(self, new\_currency\_type):

if new\_currency\_type == self.currency\_type:

# nothing to do

return Currency(self.amount, self.currency\_type)

if new\_currency\_type not in Currency.VALID\_CURRENCIES \

or self.currency\_type not in Currency.VALID\_CURRENCIES:

print("Conversion from {} to {} not allowed".format(self.currency\_type, new\_currency\_type))

return

# prepare URL

url = "https://api.exchangerate.host/convert?from="

url += self.currency\_type

url += "&to=" + new\_currency\_type

conv = urllib.request.urlopen(url)

# read() returns an array of bytes, we want a string decoded in UTF-8

response = conv.read().decode('UTF-8')

#print (response)

# Extract the exchange rate from the variable 'result' and finish the implementation of the method.

# The return is given. Amount is the correct converted amount that needs to be found

exchange\_info = json.loads(response)

amount = self.amount \* float(exchange\_info["info"]["rate"])

print("{} {} => {} {}".format(self.amount, self.currency\_type, amount, new\_currency\_type))

return Currency(amount, new\_currency\_type)

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

output = str(self.amount) + " " + self.currency\_type

return output

def \_\_repr\_\_(self):

pass

def \_\_add\_\_(self, other\_curr):

if type(other\_curr) == Currency:

if self.currency\_type == other\_curr.currency\_type:

amount = self.amount + other\_curr.amount

return Currency(amount, self.currency\_type)

else:

new\_other = other\_curr.convert\_to(self.currency\_type)

amount = self.amount + new\_other.amount

return Currency(amount, self.currency\_type)

elif type(other\_curr) == int or type(other\_curr) == float:

amount = self.amount + other\_curr

return Currency(amount, self.currency\_type)

else:

raise TypeError

def \_\_sub\_\_(self, other\_curr):

pass

def \_\_radd\_\_(self, other\_curr):

return self.\_\_add\_\_(other\_curr)

def \_\_rsub\_\_(self, other\_curr):

pass

def \_\_gt\_\_(self, other\_curr):

pass

# This main is incomplete because not all methods are tested

# Some outputs are given by the comments next to the commands (this will depend on the exchange rate of the day)

# Your code should be able to output similar values when

# you remove the '#' in the beginning of the lines

curr = Currency(7.50, 'USD')

print(curr) # 7.50 USD

curr2 = Currency(2, 'EUR')

print(curr2) # 2.00 EUR

new\_curr = curr2.convert\_to(curr.currency\_type) # 2.000000 EUR => 2.211600 USD

print(new\_curr) # 2.21 USD

sum\_curr = 5 + curr # 5.\_\_add\_\_(curr) -> curr.\_\_radd\_\_(5) -> curr.\_\_add\_\_(5)

print(sum\_curr) # 9.71 USD

#sum\_curr2 = curr + 5.5

#print(sum\_curr2) # 13.00 USD