Lab 2 (10 points) **Due date:** June 4, 2023, 11:59 PM ET

Before you start…

Keep in mind these tips from your instructor:

- Watch the lectures in Module 3.1 before you start this assignment, including the hands-on videos (especially if you are new to object-oriented programming)
- Outline a strategy before you start typing code and share it with the course staff for additional support.
- ALL string must match the given documentation, typos or additional character will result on failed cases and point deductions.
- Ask questions using our Lab 2 channel in Microsoft Teams

Tip: Python's string formatting syntax could be useful to construct your output strings

```
>>> item, price, stock = 'Potatoes', 3.5, [20]
>>> f'{item} cost {price} and we have {stock[0]}'
'Potatoes cost 3.5 and we have 20'
```

Practice: Using the definition of the Car class shown below, fill in the blanks without using the Python interpreter (solutions are available at the end of this document)

```
class Car:

    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.color = None

    def get_color(self):
        if self.color is None:
            print(f"{self.make} {self.model} has no color")
        return self.color

    def paint(self, color):
        self.color = color
        return f"{self.make} {self.model} has been painted {color}"

>>> my_car = _________________     # Create an instance of the class
>>> my_car.make
'Honda'
>>> my_car.model
'Civic'
>>> my_car.get_color()
_____________________________       # What would Python display
>>> ______________________________        # What call produces the output
'Honda Civic has been painted Navy'
>>> ______________________________        # What call produces the output
'Navy'
```

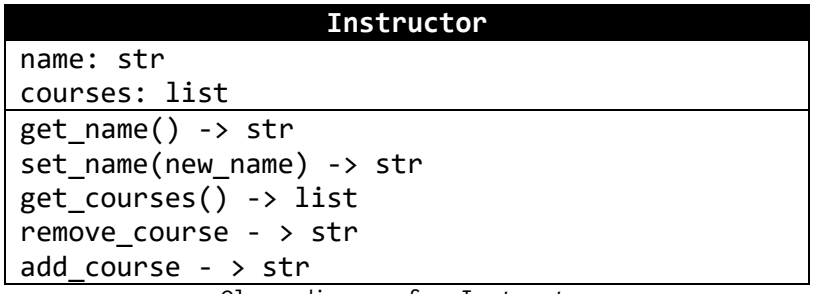
REMINDER: As you work on your coding assignments, it is important to remember that passing the examples provided does not guarantee full credit. While these examples can serve as a helpful starting point, it is ultimately your responsibility to thoroughly test your code and ensure that it is functioning correctly and meets all the requirements and specifications outlined in the assignment instructions. Failure to thoroughly test your code can result in incomplete or incorrect outputs, which will lead to deduction in points for each failed case.

Create additional test cases and share them with you classmates on Teams, we are in this together!

90% > Passing all test cases 10% > Clarity and design of your code

Section 1: The Instructor class **(1.5 pts)**

Implement the Instructor class to initialize an instance as follows: The instructor's name should be stored in the *name* attribute, and the list of courses the instructor teaches should be stored in the *courses* attribute. The list of courses is initially empty when a new Instructor object is created.



Class diagram for Instructor

get_name(self)

Returns the name of the instructor.

set_name(self, new_name):

Sets the name of the instructor to the specified *new_name*. Method performs the update only when *new_name* is a non-empty string.

get_courses(self):

Returns the list of courses taught by the instructor.

remove_course(self, course):

Removes the specified course from the list of courses taught by the instructor, if it exists in the list.

add_course(self, course):

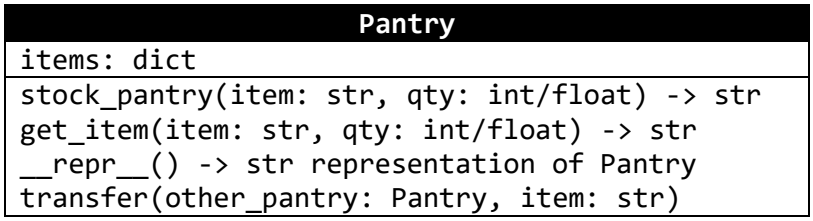
Adds the specified course to the list of courses taught by the instructor, if it is not already in the list.

Examples:

```
>>> t1= Instructor('John Doe')
>>> t1.get_name()
'John Doe'
>>> t1.get_courses()
[]
>>> t1.add_course('MATH140')
>>> t1.get_courses()
['MATH140']
>>> t1.add_course('STAT100')
>>> t1.get_courses()
['MATH140', 'STAT100']
>>> t1.add_course('STAT100')
>>> t1.get_courses()
['MATH140', 'STAT100']
>>> t1.remove_course('MATH141')
>>> t1.get_courses()
['MATH140', 'STAT100']
>>> t1.remove_course('MATH140')
>>> t1.get_courses()
['STAT100']
```

Section 2: The Pantry class **(2 pts)**

The Pantry class is used to keep track of items in a kitchen pantry, letting users know when an item has run out. This class uses a dictionary to store items, where the key is a string representing the name of the item, and the value is a numerical value of the current quantity for that item. The dictionary has been initialized in the constructor for you, **do not modify it**. The strings returned by the class methods must match the provided examples in the docstring. All values in the dictionary must be stored as float values.



Class diagram for Pantry

__repr__(self)
Special methods that provide a legible string representation for instances of the Pantry class. Objects will be represented using the format return "I am a Pantry object, my current stock is <all items>"

Examples:

```
>>> sara_pantry = Pantry()
>>> sara_pantry
I am a Pantry object, my current stock is {}
>>> sara_pantry.stock_pantry('Bread', 2)   # returned string not included
>>> sara_pantry.stock_pantry('Lettuce', 1.5) # returned string not included
>>> sara_pantry
I am a Pantry object, my current stock is {'Bread': 2.0, 'Lettuce': 1.5}
```

stock_pantry(self, item, qty)
Adds the given quantity of item to the dictionary, returning a string with the current stock using the format "Pantry Stock for <item>: <total>"

Examples:

```
>>> sara_pantry = Pantry()
>>> sara_pantry.stock_pantry('Lettuce', 1.5)
'Pantry Stock for Lettuce: 1.5'
>>> sara_pantry.stock_pantry('Lettuce', 2)
'Pantry Stock for Lettuce: 3.5'
>>> sara_pantry.stock_pantry('Cookies', 3)
'Pantry Stock for Cookies: 3.0'
```

get_item(self, item, qty)
Subtracts up to the given quantity of the item from the dictionary. If the quantity is greater than the current stock, it should only use the remaining quantity, alerting the user to buy more of that item. If the item is not in the pantry, it should also let the user know.

Examples:

```
>>> sara_pantry = Pantry()
>>> sara_pantry.stock_pantry('Lettuce', 1.5)
'Pantry Stock for Lettuce: 1.5'
>>> sara_pantry.get_item('Lettuce', 0.5)
'You have 1.0 of Lettuce left'
>>> sara_pantry.get_item('Cereal', 5)
"You don't have Cereal"
>>> sara_pantry.get_item('Lettuce', 1.5)
'Add Lettuce to your shopping list!'
>>> sara_pantry.items
{'Lettuce': 0.0}
```

transfer(self, other_pantry, item)
Moves then entire item stock from other_pantry to the original pantry. Items that have zero stock are not moved to the original Pantry.

Examples:

```
>>> sara_pantry = Pantry()
>>> sara_pantry.stock_pantry('Bread', 2)
'Pantry Stock for Bread: 2.0'
>>> sara_pantry.stock_pantry('Cereal', 1)
'Pantry Stock for Cereal: 1.0'
>>> sara_pantry
I am a Pantry object, my current stock is {'Bread': 2.0, 'Cereal': 1.0}
>>> ben_pantry = Pantry()
>>> ben_pantry.stock_pantry('Cereal', 2)
'Pantry Stock for Cereal: 2.0'
>>> ben_pantry.stock_pantry('Noodles', 5)
'Pantry Stock for Noodles: 5.0'
>>> ben_pantry
I am a Pantry object, my current stock is {'Cereal': 2.0, 'Noodles': 5.0}
>>> sara_pantry.transfer(ben_pantry, 'Noodles')
>>> sara_pantry
I am a Pantry object, my current stock is {'Bread': 2.0, 'Cereal': 1.0, 'Noodles':
5.0}
>>> ben_pantry
I am a Pantry object, my current stock is {'Cereal': 2.0, 'Noodles': 0.0}
```

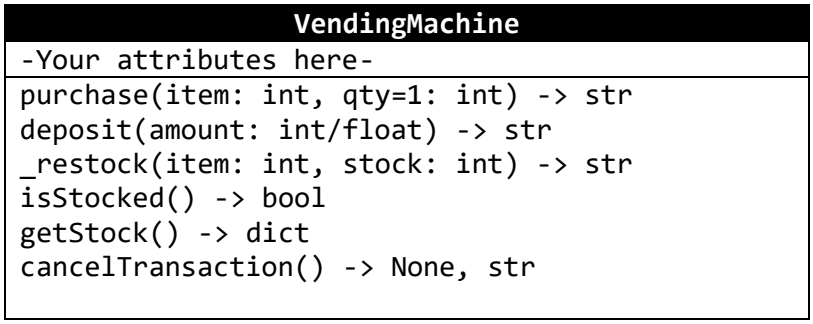
Section 3: The Vendor and VendingMachine classes **(3.5 pts)**

These classes will represent a vendor (someone that makes goods and services available to companies or consumers) and a vending machine that gives the user money back only when a transaction is cancelled, after making a purchase (change back), or if the machine is out of all stock.

Instances of VendingMachine are created through an instance of the Vendor class (already implemented for you in the starter code) using the *install* method. This vending machine will sell four different products:

Product ID	Price
156	1.5
254	2.0
384	2.5
879	3.0

When creating an instance of VendingMachine, the machine starts out with 3 items of each product. Note that a dictionary could be useful here to keep track of the item_id, the unit cost and the stock for that item. A VendingMachine object returns strings describing its interactions. All numbers representing money must be displayed as floats.



Class diagram for VendingMachine

Note: Read the description of all method first, then outline your strategy. In this section, you are expected to reduce the amount of repeated code by calling methods in the class to reuse code. Both functionality and design are part of the grade for this exercise.

purchase(self, item, qty=1)
Attempts to buy something from the vending machine. Before completing the purchase, check to make sure the item is valid, there is enough stock of said item, and there is enough balance.

Output (in order of priority)

str	"Invalid item" if the item id is invalid (Highest priority)
	"Machine out of stock" is returned if the machine is out of stock for all items
	"Item out of stock" is returned if there is no stock left of requested item.
	"Current *item_id* stock: *stock*, try again" if there is not enough stock
	"Please deposit $*remaining*" if there is not enough balance
	"Item dispensed" if there is no money to give back to the user
	"Item dispensed, take your $*change* back" if there is change to give back

deposit(self, amount)
Deposits money into the vending machine, adding it to the current balance.

Output

str	"Balance: $*balance*" when machine is stocked
	"Machine out of stock. Take your $*amount* back" if the machine is out of stock.

_restock(self, item, stock)
A protected method that adds stock to the vending machine. Note that when the VendingMachine runs out of a product, a Vendor object will trigger the operation using its restock method (already implemented for you)

Output

str	"Current item stock: *stock*" for existing id
	"Invalid item" is returned if the item id is invalid.

isStocked(self)
A property method (behaves like an attribute) that returns True if any item has any nonzero stock, and False if all items have no stock.

Output

bool	Status of the vending machine.

getStock(self)
A property method (behaves like an attribute) that gets the current stock status of the machine. Returns a dictionary where the key is the item and the value is the list [*price*, *stock*].

Output

dict	Current stock status represented as a dictionary.

cancelTransaction(self)

Returns the current balance to the user

Output

None	Nothing is returned if balance is 0
str	“Take your $*amount* back” if there is money to give back

Examples:

```
>>> vendor1 = Vendor('John Doe')
>>> x = vendor1.install()
>>> x.getStock
{156: [1.5, 3], 254: [2.0, 3], 384: [2.5, 3], 879: [3.0, 3]}
>>> x.purchase(56)
'Invalid item'
>>> x.purchase(879)
'Please deposit $3.0'
>>> x.deposit(10)
'Balance: $10.0'
>>> x.purchase(156, 3)
'Item dispensed, take your $5.5 back'
>>> x.isStocked
True
```

More examples of class behavior provided in the starter code

Section 4: The Line class **(3 pts)**

The Line class represents a 2D line that stores two Point2D objects and provides the distance between the two points and the slope of the line using the **property methods** *getDistance* and *getSlope*. The constructor of the Point2D class has been provided in the starter code. You might use the math module

Point2D
x : int/float y : int/float
- Your methods here -

Line
- Your attributes here -
get_distance -> float get_slope -> sloat __str__() -> str representation of Line __repr__ -> str representation of Line __eq__(other: any) -> bool __mul__(other: any) -> Line __contains__(point: any) -> bool

Note: Read the description of all method first, then outline your strategy. In this section, you are expected to reduce the amount of repeated code by calling methods in the class to reuse code. Both functionality and design are part of the grade for this exercise.

getDistance(self)
A **property method** that gets the distance between the two Point2D objects that created the Line. The formula to calculate the distance between two points in a two-dimensional space is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Returns a float rounded to 3 decimals. To round you can use the round method as `round(value, #ofDigits)`

getSlope(self)
A **property method** that gets the slope (gradient) of the Line object. The formula to calculate the slope using two points in a two-dimensional space is:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

If the slope exists, returns a float rounded to 3 decimals, otherwise, returns infinity as a float (inf float). To round you can use the round method as `round(value, #ofDigits)`

Examples:

```
>>> p1 = Point2D(-7, -9)
>>> p2 = Point2D(1, 5.6)
>>> line1 = Line(p1, p2)
>>> line1.getDistance
16.648
>>> line1.getSlope
1.825
```

__str__(self) and __repr__(self)

Special methods that provide a legible representation for instances of the Line class. Objects will be represented using the "slope-intercept" equation of the line:

$$y = mx + b$$

To find *b*, substitute *m* with the slope and *y* and *x* for any of the points and solve for *b*. *b* must be rounded to 3 decimals.. To round you can use the round method as `round(value, #ofDigits)`. The representation will be the string 'Undefined' if the slope of the line is undefined. You are allowed to define a property method to compute the interception *b*.

Examples:

```
>>> p1 = Point2D(-7, -9)
>>> p2 = Point2D(1, 5.6)
>>> line1 = Line(p1, p2)
>>> line1
y = 1.825x + 3.775
>>> line5=Line(Point2D(6,48),Point2D(9,21))
>>> line5
y = -9.0x + 102.0
>>> line6=Line(Point2D(2,6), Point2D(2,3))
>>> line6.getDistance
3.0
>>> line6.getSlope
inf
>>> line6
Undefined
>>> line7=Line(Point2D(6,5), Point2D(9,5))
>>> line7
y = 5.0
```

__eq__

A special method that determines if two Line objects are equal (supports the == operator). For instances of this class, we will define equality as two lines having the same points. To simplify this method, you could try defining equality in the Point2D class. You cannot make any assumptions about the value that will be compared with the Line object.

__mul__

A special method to support the * operator. Returns a new Line object where the x,y attributes of every Point2D object is multiplied by an integer. The only operation allowed is Line*integer, any other non-integer values return None.

__contains__

A special method to support the in operator. Returns True if the Point object lies on the Line object, False otherwise. You cannot make any assumptions about the value that will be on the right side of the operator. If the slope is undefined, return False.

Recall that floating-point numbers are stored in binary, as a result, the binary number may not accurately represent the original base 10 number. For example:

```
>>> 0.7 + 0.2 == 0.9
False
```

This error, known as floating-point representation error, happens way more often than you might realize. To implement this method, you will need to compare floating-point numbers. Avoid checking for equality using == with floats. Instead use the isclose() method available in the math module:

```
>>> import math
>>> math.isclose(0.7 + 0.2, 0.9)
True
```

math.isclose() checks if the first argument is acceptably close to the second argument by examining the distance between the first argument and the second argument, which is equivalent to the absolute value of the difference of the values:

```
>>> x = 0.7 + 0.2
>>> y = 0.9
>>> abs(x-y)
1.1102230246251565e-16
```

If abs(x - y) is smaller than some percentage of the larger of x or y, then x is considered sufficiently close to y to be "equal" to y. This percentage is called the relative tolerance.

Examples:

```
>>> p1 = Point2D(-7, -9)
>>> p2 = Point2D(1, 5.6)
>>> line1 = Line(p1, p2)
>>> line2 = line1*4
>>> isinstance(line2, Line)
True
>>> line2
y = 1.825x + 15.1
>>> line3 = line1*4
>>> line3
y = 1.825x + 15.1
>>> line1==line2
False
>>> line3==line2
True
>>> line3==9
False
>>> line5=Line(Point2D(6,48),Point2D(9,21))
>>> line5
y = -9.0x + 102.0
>>> Point2D(45,3) in line5
False
>>> Point2D(34,-204) in line5
True
>>> (9,5) in line5
False
```

Solutions to Practice problem

```
class Car:

    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.color = None

    def get_color(self):
        if self.color is None:
            print(f"{self.make} {self.model} has no color")
        return self.color

    def paint(self, color):
        self.color = color
        return f"{self.make} {self.model} has been painted {color}"
```

```
>>> my_car = Car('Honda', 'Civic')
>>> my_car.make
'Honda'
>>> my_car.model
'Civic'
>>> my_car.get_color()
Honda Civic has no color
>>> my_car.paint('Navy')
'Honda Civic has been painted Navy'
>>> my_car.get_color()      # OR  my_car.color
'Navy'
```