Lab 2 (10 points) **Due date:** September 19, 2025, 11:59 PM

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 strings 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

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'
```

To represent infinity in Python, you can use the math.inf floating-point value from the math library.

```
>>> import math
>>> math.inf == float('inf')
True
```

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'
```

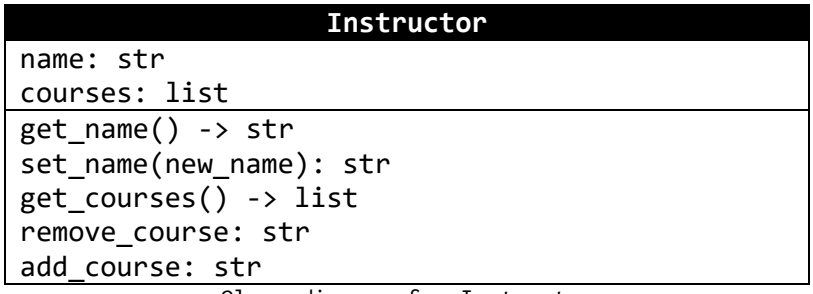
REMINDER: As you work on your coding assignments, it is important to remember that passing the examples provided does not guarantee full credit. While these examples can serve as a helpful starting point, it is ultimately your responsibility to thoroughly test your code and ensure that it is functioning 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.

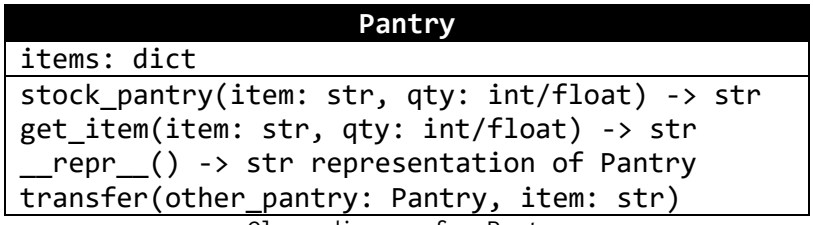
add_course(self, course):
Adds the specified course to the list of courses taught by the instructor, if it is not already in the list.

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.

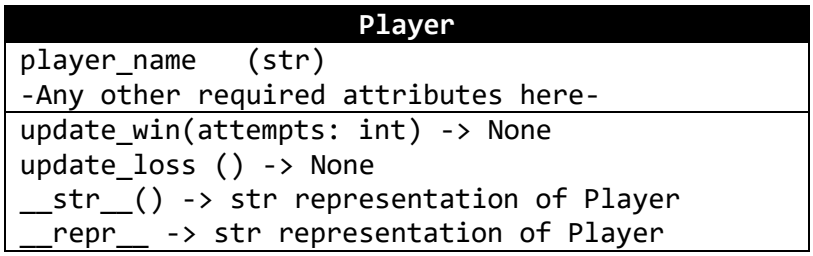
Examples:

```
>>> 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 Player and Wordle classes **(4 pts)**

These classes will represent a player (someone that plays a game) and one round of a Wordle game (guess a five-letter word in a given number of attempts). Player class is worth 1 pt, Wordle class 3 pts

When creating an instance of Player, the object starts out with no game records (games played, wins, losses and game won with the least number of attempts). These records will be updated via the Player methods `update_win` and `update_loss`, and the information will be displayed using the special methods to define the string representation of the object (Note that to get credit for the answer, the strings must match)



Class diagram for Player

update_win(self, attempts)
Increases by 1 the value of the instance attribute that keeps track of the number of games the player has won. It also updates the instance attribute that keeps track of the best game (game won with least number of attempts.

update_loss(self)
Increases by 1 the value of the instance attribute that keeps track of the number of games the player has lost.

__str__ and __repr__ methods
Special methods that provide a legible representation for instances of the Player class. When no games have been played, it returns the string `'No game records for player_name'`, otherwise, returns a multiline string with the player's stats for all games played (format must match the format from the examples)

Examples:

```
>>> p1 = Player('Susy')
>>> print(p1)
No game records for Susy
>>> p1.update_win(5)
>>> p1
*Game records for Susy*
Total games: 1
Games won: 1
Games lost: 0
Best game: 5 attempts
```

```
>>> p1.update_win(2)
>>> p1.update_loss()
>>> p1
*Game records for Susy*
Total games: 3
Games won: 2
Games lost: 1
Best game: 2 attempts
>>> p1.update_win(3)
>>> p1
*Game records for Susy*
Total games: 4
Games won: 3
Games lost: 1
Best game: 2 attempts
```

The game will be played with an instance of the Wordle class. To create an instance of this class, a Player object and a 5-lowercase letter word are required. A Player will have 6 attempts to guess the word, however, then game can change the number of attempts at any time, and this change must affect all instances of the Wordle class (do not hard-code the number of guesses into the `play` method)

The Worle object will accept a guess using the `play` method that invokes the `process_guess` method to provide feedback to the player for each letter:

- If the guess does not contain exactly 5 characters, return `"Guess must be 5 letters long"`
- If the guess contains non-alphabet letters, return `"Guess must be all letters"`
- If the letter in the guess is in the right place, provide that letter as uppercase
- If the letter in the guess is not in the right place, provide that letter as lowercase
- If the letter in the guess isn't in the word at all, provide `"_"`

The game ends once the player uses all attempts, or the word has been guessed (the format of the returned value is shown in the examples). At this point, the game states should be updated for the Player object. Further attempts to play the game should result in the string `"Game over"`

Wordle
user (Player) word (str) -Any other required attributes here-
play(guess: str) -> str process_guess(guess: str) -> str

Class diagram for Wordle

Note: Defining the proper attributes to control the state of each object, correct use of methods, and correct syntax to update the internal state of the Player object from the Wordle game are part of the grade.

play(self, guess)

Handles each attempt of the game and updates the stats for the Player. Calls `process_guess` to send feedback back to the player

Output

str	"You won the game" when player has guessed the word
	"The word was *solution*" (where solution is the word to guess) when plyer ran out of attempts
	"Game over" when player tris to play another guess after game has ended
	"Guess must be 5 letters long" when string provided does not have five characters
	"Guess must be all letters" when string of size 5 is not all letters
	"XXXXX" to provide feedback to player

process_guess(self, guess)

Revises each character in guess to provide the appropriate feedback

Output

str	"Guess must be 5 letters long" when string provided does not have five characters
	"Guess must be all letters" when string of size 5 is not all letters
	"XXXXX" to provide feedback to player

Examples:

```
>>> p1 = Player('Susy')
>>> p2 = Player('Taylor')
>>> w1 = Wordle(p1, 'water')
>>> w2 = Wordle(p2, 'cloud')
>>> w3 = Wordle(p1, 'jewel')
>>> w1.play('camel')
'_A_E_'
>>> w1.play('ranes')
'rA_E_'
>>> w1.play('baner')
'_A_ER'
>>> w1.play('water')
'You won the game'
>>> w1.play('rocks')
'Game over'
>>> w1.play('other')
'Game over'
>>> w3.play('beast')
'_E___'
>>> w3.play('peace')
'_E__e'
>>> w3.play('keeks')
'_Ee__'
>>> w3.play('jewel')
```

```
'You won the game'
>>> w2.play('classes')
'Guess must be 5 letters long'
>>> w2.play('cs132')
'Guess must be all letters'
>>> w2.play('audio')
'_ud_o'
>>> w2.play('kudos')
'_udo_'
>>> w2.play('would')
'_oulD'
>>> w2.play('bound')
'The word was cloud'
>>> w2.play('cloud')
'Game over'
>>> p1
*Game records for Susy*
Total games: 2
Games won: 2
Games lost: 0
Best game: 4 attempts
>>> p2
*Game records for Taylor*
Total games: 1
Games won: 0
Games lost: 1
Best game: None
```

Section 4: The Line class **(2.5 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 -
getDistance -> float getSlope -> float __str__() -> str representation of Line __repr__ -> str representation of Line __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
```

__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 **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
>>> 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'
```