CMPSC-132: Programming and Computation II

Lab 2 (10 points) **Due date:** February 4th, 2022, 11:59 PM

Goal: The goal of this assignment is to familiarize with object-oriented programming by providing functionality to custom classes and interacting with custom objects

General instructions:

- The work in this assignment must be your own original work and be completed alone.
- The instructor and course assistants are available on Teams and with office hours to answer any questions you may have. You may also share testing code on Teams.
- A doctest is provided to ensure basic functionality and may not be representative of the full range of test cases we will be checking. Further testing is your responsibility.
- Debugging code is also your responsibility.
- You may submit more than once before the deadline; only the latest submission will be graded.

Assignment-specific instructions:

- Download the starter code file from Canvas. Do not change the function names or given starter code in your script.
- Each question has different requirements, read them carefully and ask questions if you need clarification. No credit is given for code that does not follow directions.
- Watch the Module 3 lectures first. Do not overthink this assignment, you do not have to come up with a complex algorithm to solve it. Just follow the directions and formulas, the purpose of this assignment is to familiarize you with object-oriented programming syntax.
- If you are unable to complete a function, use the pass statement to avoid syntax errors

Submission format:

- Submit your LAB2.py file to the Lab 2 Gradescope assignment before the due date.
- As a reminder, code submitted with syntax errors does not receive credit, please run your file before submitting.

Section 1: The Fibonacci class **(2 pts)**

The Fibonacci sequence is the series of numbers where each number is the sum of the two preceding numbers. For example:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, …

Mathematically we can describe this as $x_n = x_{n-1} + x_{n-2}$

The Fibonacci class has a *value* attribute that represents a number in the Fibonacci sequence. Write the code to implement the *next* method that returns an instance of the Fibonacci class whose value is the next Fibonacci number. You are not allowed to modify the constructor and the __repr__ methods.

Methods

Return Type	Name	Description
Fibonnaci	next(self)	Next number in the Fibonacci sequence
str	__repr__(self)	Legible representation for Fibonacci objects. Already implemented for you in the starter code

next(self)

Returns a Fibonnaci instance whose value is the next Fibonacci number. Method should not compute all previous Fibonnaci numbers each time is called (in other words, no loops or recursion required). Instead, keep track of the previous number by setting a new instance attribute inside this method. Remember from this week's lectures that we can create instance attributes for objects at any point, even outside the constructor.

Output

Fibonnaci	Next Fibonacci number. Note that Fibonacci objects are displayed based on the return value of the __repr__ method in the Fibonacci class.

Examples:

```
>>> fib_seq = Fibonacci()
>>> fib_seq
<Fibonacci object>, value = 0
>>> fib_seq.next()
<Fibonacci object>, value = 1
>>> fib_seq.next().next()
<Fibonacci object>, value = 1
>>> fib_seq.next().next().next()
<Fibonacci object>, value = 2
>>> fib_seq.next().next().next()
<Fibonacci object>, value = 2
```

Section 2: The Vendor and VendingMachine classes **(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.

Tip: Python's string formatting syntax could be useful

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

Methods

Type	Name	Description
str	purchase(self, item, qty=1)	Attempts to buy something from the machine.
str	deposit(self, amount)	Deposits money into the vending machine.
str	_restock(self, item, stock)	Adds stock to the vending machine
bool	isStocked(self)	A property method that checks for the stock status.
dict	getStock(self)	A property method that gets the current stock status of the machine.
str	cancelTransaction(self)	Gives the money back to the user

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.

Input (excluding self)

int	*item*	An integer that might represent item ID of item to purchase
int	*qty*	The desired quantity of said item. Defaults to 1

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.

Input (excluding self)

int or float	amount	Amount of money to add to the 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)

Input (excluding self)

int	item	Item ID of item to restock.
int	stock	The amount of stock to add to the vending machine.

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'
>>> x.deposit(10)
'Balance: $10'
>>> 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 3: 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.

Methods

Type	Name	Description
float	getDistance	Returns the distance between two Point2D objects
float	getSlope	Returns the slope of the line that passes through the two points

Special methods

Type	Name	Description
str	__str__ __repr__	Gets a legible representation of the Line in the form $y = mx + b$
bool	__eq__	Determines if two Line objects are equal
Line	__mul__	Returns a new Line object where each coordinate is multiplied by a positive integer

getDistance(self)

A property method (behaves like an attribute) 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)`

Output

float	Returns the distance between the two Point2D objects that pass through the Line

getSlope(self)

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

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

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

Output

float	Returns the slope of the line,
float	inf float for undefined slope (denominator is zero)

Examples:

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

__str__ and __repr__

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)`

Output

str	Slope-intercept equation of the line, representation must be adjusted if *m* or *b* are 0, or if *b* is positive/negative
str	'Undefined' for undefined slope

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. 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

Output

bool	True if lines have the same points, False otherwise

__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 operations allowed are integer*Line and Line*integer, any other non-integer values return None. You will need to override both the "normal" version and the "right side" version of mul to support such operations, since integer*Line calls __mul__ in the int class.

Output

Line	A new Line object where point1 and point2 are multiplied by an integer

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 = 4*line1
>>> line3
y = 1.825x + 15.1
>>> line1==line2
False
>>> line3==line2
True
>>> line5==9
False
```