Homework 3 (100 points) **Due date:** July 2, 2023, 11:59 PM ET

Before you start…

Keep in mind these tips from your instructor:

- Watch the lectures in Module 5 before you start. Those lectures describe the main algorithms required to complete this assignment.
- Read carefully the information presented in the instructions. You will be using custom classes that need to interact with each other and with Python's data types
- All methods that output a string must **return** the string, not **print** it. Code will not receive credit if you use print to display the output
- Ask questions using our Homework 3 channel in Microsoft Teams
- **One more time:** read the instructions carefully and outline the purpose of each method. You are expected to use methods defined in other classes to complete methods in other sections instead of repeating code. Proper design of the code is 25% of the grade for this assignment.

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.

60% > Passing all test cases 40% > Clarity and design of your code

Section 1: The Stack class (20 points)

This class represents the stack data structure discussed in this module. Use the implementation of the Node class to implement a stack that supports the following operations.

Make sure to update the top pointer accordingly as the stack grows and shrinks. You are not allowed to use any other data structures for the purposes of manipulating elements, nor may you use the built-in stack tool from Python. Your stack must be implemented as a Linked List, not a Python list.

Attributes

Type	Name	Description
Node	top	A pointer to the top of the stack

Methods

Type	Name	Description
None	push(self, item)	Adds a new node with value=item to the top of the stack
(any)	pop(self)	Removes the top node and returns its value
(any)	peek(self)	Returns the value of the top node without modifying the stack
bool	isEmpty(self)	Returns True if the stack is empty, False otherwise

Special methods

Type	Name	Description
str	__str__(self)	Returns the string representation of this stack
str	__repr__(self)	Returns the same string representation as __str__
int	__len__(self)	Returns the length of this stack (number of nodes)

The Node class has already been implemented for you in the starter code and is described below. Do not modify the Node class.

Attributes

Type	Name	Description
(any)	value	The value that this node holds
Node	next	A pointer to the next node in the data structure

Type	Name	Description
str	__str__(self)	Returns the string representation of this node
str	__repr__(self)	Returns the same string representation as __str__

push(self, item)
Adds a new node with value=item to the top of the stack. Nothing is returned by this method.

Input (excluding self)		
(any)	item	The value to store in a node

pop(self)
Removes the top node from the stack and returns that node's value (not the Node object).

Output	
(any)	Value held in the topmost node of the stack
None	Nothing is returned if the stack is empty

peek(self)
Returns the value (not the Node object) of the topmost node of the stack, but it is not removed.

Output	
(any)	Value held in the topmost node of the stack
None	Nothing is returned if the stack is empty

isEmpty(self)
Tests to see whether this stack is empty or not.

Output	
bool	True if this stack is empty, False otherwise

__len__(self)
Returns the number of elements in this stack.

Output	
int	The number of elements in this stack

__str__(self), __repr__(self)
Two special methods that return the string representation of the stack. This has already been implemented for you, so do not modify it. If the class is implemented correctly, __str__ and __repr__ display the contents of the stack in the format shown in the doctest.

Output	
str	The string representation of this stack

Section 2: The Calculator class (50 points)

Implement a class that calculates mathematic expressions. The input passed to this Calculator is in infix notation, which gets converted to postfix internally, then the postfix expression is evaluated (we evaluate in postfix instead of infix because of how much simpler it is). More details about infix to postfix conversion can be found in the video lectures.

This calculator should support numeric values, five arithmetic operators (+, –, *, /, ^), and parenthesis. Follow the PEMDAS order of operations (you can define precedence of operators with a dictionary or a helper method). Note that exponentiation is ** in Python.

The infix expressions will have tokens (operators, operands) that might be separated by spaces. For the case of **negative numbers**, you can assume the negative sign is prefixed to the number (no spaces) and that we don't have cascade of negatives such as '3 + - -5', but '3 + -5', '3 - -5', etc. are possible. You must define your own version of split() to isolate the tokens, since spaces between them is not guaranteed.

Expressions are considered invalid if they meet any of the following criteria:

- Contains unsupported operators `4 $ 5`
- Contains consecutive operators (excludes cases for negative numbers such as 3 + -5) `4 * + 5`
- Has missing operands `4 +`
- Has missing operators `4 5`
- Has unbalanced parenthesis `) 4 + 5 (` or `( 4 + 5 ) )`
- Tries to do implied multiplication `3(5)` instead of `3*( 5 )`

Make sure to have proper encapsulation of your code by using proper variable scopes and writing other helper methods to generalize your code for processes such as string processing and input validation. Do not forget to document your code.
As a suggestion, start by implementing your methods assuming the expressions are always valid, that way you have the logic implemented and you only need to work on validating the expressions.

Attributes

Type	Name	Description
str	__expr	The expression this calculator will evaluate

Methods

Type	Name	Description
None	setExpr(self, new_expr)	Sets the expression for the calculator to evaluate
str	getExpr(self)	Getter method for the private expression attribute
bool	_isNumber(self, aSring)	Returns True if aSring can be converted to a float
str	_getPostfix(self, expr)	Converts an expression from infix to postfix
float	calculate(self)	Calculates the expression stored in this calculator

setExpr(self, new_expr)

Sets the expression for the calculator to evaluate. This method is already implemented for you.

Input (excluding self)		
str	new_expr	The new expression (in infix) for the calculator to evaluate

getExpr(self)

Property method for getting the expression in the calculator. This method is already implemented for you.

Output	
str	The value stored in __expr

_isNumber(self, txt) (5 points)

Returns True if txt is a string that can be converted to a float, False otherwise. Note that the type conversion `float('4.56')` returns 4.56 but `float('4  56')` raises an exception. A try/except block could be useful here.

Input (excluding self)		
str	txt	The string to check if it represents a number

Output	
bool	True if txt can be successfully casted to a float, False otherwise

_getPostfix(self, expr) (35 points)

Converts an expression from infix to postfix. All numbers in the output must be represented as a float. You must use the Stack defined in section 1 in this method, otherwise your code will not receive credit. (Stack applications video lecture can be helpful here).

Input (excluding self)		
str	expr	The expression in infix form

Output	
str	The expression in postfix form
None	None is returned if the input is an invalid expression

calculate(self) (10 points)

A property method that evaluates the infix expression saved in self.__expr. First convert the expression to postfix, then use a stack to evaluate the output postfix expression. You must use the Stack defined in section 1 in this method, otherwise your code will not receive credit.

Input (excluding self)		
str	txt	The string to check if it represents a number

Output	
float	The result of the expression
None	None is returned if the input is an invalid expression or if expression can’t be computed

Section 3: The AdvancedCalculator class (30 points)

The AdvancedCalculator class represents a calculator that supports multiple expressions over many lines, as well as the use of variables. Lines will be split by semicolons (;), and every token will be separated by a single space. Each line will start with a variable name, then an '=' character, then a mathematical expression. The last line will ask to return a mathematical expression too.

You can assume that an expression will not reference variables that have not been defined yet.

You can assume that variable names will be consistent and case sensitive. A valid variable name is a non-empty string of alphanumeric characters, the first of which must be a letter.

You must use a Calculator to evaluate each expression in this class, otherwise, no credit will be given.

Attributes

Type	Name	Description
str	expressions	The expressions this calculator will evaluate
dict	states	A dictionary mapping variable names to their float values

Methods

Type	Name	Description
bool	_isVariable(self, word)	Returns True if word is a valid variable name
str	_replaceVariables(self, expr)	Replaces all variables in an expression with values
dict	calculateExpressions(self)	Calculates all expressions and shows state at each step

```
>>> C = AdvancedCalculator()
>>> C.setExpression('A = 1;B = A + 9;C = A + B;A = 20;D = A + B + C;return D + 2 * B')
>>> C.states
{}
>>> C.calculateExpressions()  # spacing added for readability
{'A = 1':         {'A': 1.0},
 'B = A + 9':     {'A': 1.0,  'B': 10.0},
 'C = A + B':     {'A': 1.0,  'B': 10.0, 'C': 11.0},
 'A = 20':        {'A': 20.0, 'B': 10.0, 'C': 11.0},
 'D = A + B + C': {'A': 20.0, 'B': 10.0, 'C': 11.0, 'D': 41.0},
 '_return_':      61.0}
>>> C.states
{'A': 20.0, 'B': 10.0, 'C': 11.0, 'D': 41.0}
```

_isVariable(self, word) (5 points)

Determines if the input is a valid variable name (see above for rules for names). The string methods str.isalpha() and str.isalnum() could be helpful here.

Input (excluding self)		
str	word	The string to check if it is a valid variable name

Output	
bool	True if word is a variable name, False otherwise

_replaceVariables(self, expr) (10 points)

Replaces all variables in the input expression with the current value of those variables saved in self.states.

Input (excluding self)		
str	expr	The input expression that may contain variables

Output	
str	The expression with all variables replaced with their values
None	Nothing is returned if expression has invalid variables or uses a variable that has not been defined

calculateExpressions(self) (15 points)

Evaluates each expression saved in self.expressions. For each line, replace all variables in the expression with their values, then use a Calculator object to evaluate the expression and update self.states. This method returns a dictionary that shows the progression of the calculations, with the key being the line evaluated and the value being the current state of self.states after that line is evaluated. The dictionary must include a key named '_return_' with the return value as its value.

Hint: the str.split(*sep*) method can be helpful for separating lines, as well as separating the variable from the expression (for the lines that are formatted as var = expr). The str.strip() method removes the white space before and after a string. Don't forget dictionaries are mutable objects!

```
>>> 'hi;there'.split(';')
['hi', 'there']
>>> 'hi=there'.split('=')
['hi', 'there']
>>> ' hi     there       '.strip()
' hi    there'
```

Output	
dict	The different states of self.states as the calculation progresses.
None	Nothing is returned if there was an error replacing variables