TDT4113 (PLAB2) Project 4: Calculator

— Assignment Sheet —

This document describes the fourth project in PLAB, "Calculator".

- The project must be solved individually
- The project should be solved with object-oriented code written in Python
- You should achieve a Pylint grade of at least 8.0
- The deadline for the assignment is 2 weeks, i.e., your implementation will be uploaded on Blackboard no later than February 24, 2021 at 08:00am and must be demonstrated and approved by 08:00pm on the same day. Early submissions and demonstrations are possible.

1 Background for the task

In the project you will create a calculator that takes in a calculating task written as a text string, translates the text into a suitable representation ("Reverse Polish Notation" a.k.a. RPN, we will return to this), and evaluates the RPN representation to find the answer. We assume that the input text is formally correct, so we do not get text of type "`3 add 3 multiply`". The "language" that calculator will support is flexible: It will handle the four calculators (addition, subtraction, multiplication and division), parentheses (including nested parentheses), and common mathematical functions such as `exp` and `log`.

2 Help classes and general layout

We start the work of creating some help classes. If you do not understand the point of these or how to use them, you may need to read the entire document before beginning the implementation.

2.1 Queue and Stack

Most of the calculations are done using a regular queue (a list where the item that was inserted first is what we take out first, ie "first-in first-out") and a stack (a list with "last-in, first-out" principle). You must implement these two yourself. Since the two classes are quite similar, it is wise to create a superclass `Container`, and then create the subclasses `Stack` and `Queue` afterwards. It is not difficult, for example it can be a Python list with suitable methods around. It is useful to look at what a Python list can do and what you can use in your code. For example, if you check the documentation, you will see that `list.append(element)` adds an element to the back of the list, which is all we need to implement `push`. You can check which item is at a given position in a list with "slicing", for example, `list[-1]` provides the **last** item (which corresponds to `peek` in a `Stack`) while `list[0]` provides the **first** item, as is `peek` for a `Queue`. Finally, list in Python already has a `list.pop()` that pops off and returns the last item in the list (and thus relevant to your `Stack` class). If you call `list.pop()` with an argument index, it will pop the item in that space in the list, so `list.pop(0)` is what you need for the `Queue.pop()`. If we start with `Container`, the outline of the code can look like this:

```
class Container:
    def __init__(self):
        self._items = []

    def size(self):
        # Return number of elements in self._items

    def is_empty(self):
        # Check if self._items is empty

    def push(self, item):
        # Add item to end of self._items

    def pop(self):
        # Pop off the correct element of self._items, and return it
        # This method differs between subclasses, hence is not
        # implemented in the superclass
        raise NotImplementedError

    def peek(self):
        # Return the top element without removing it
        # This method differs between subclasses, hence is not
        # implemented in the superclass
        raise NotImplementedError
```

Sub-classification is then done by overwriting `pop` and `peek`. Thus, for example, **Queue** might look like this (and much the same for **Stack**, but with "last-in, first-out"):

```python
class Queue(Container):
    def __init__(self):
        # Initialization is done at superclass
        super(Queue, self).__init__()

    def peek(self):
        # Return the first element of the list without deleting it
        assert not self.is_empty()
        return self._items[0]

    def pop(self):
        # Pop off the first element
        assert not self.is_empty()
        return self._items.pop(0)
```

Implementation – Part 1:
Finish implementation of **Container**, **Queue** and **Stack** so that all the methods given above have content. Create a "unit test" where you check that the behavior is as you expect. For example, you can put some elements in a stack, and then create a loop that, as long as the stack is not empty, takes items one by one, print them out, and tells how many items are left on the stack. Do the same test for the **Queue** class.

2.2 Function

The next thing we need is to define auxiliary classes for the functions the calculator will support. Let's start with the features where you can use this class:

```
class Function:
    def __init__(self, func):
        self.func = func

    def execute(self, element, debug=True):
        # Check type
        if not isinstance(element, numbers.Number):
            raise TypeError("The element must be a number")
        result = self.func(element)

        # Report
        if debug is True:
            print("Function: " + self.func.__name__
                  + "({:f}) = {:f}".format(element, result))

        return result
```

The class does the following: When you run `__init__`, the instance is bound to the function given as input in `func`. Later we can run the instance `execute` method and get the function evaluated in `self.func` for a given argument. In a way, this seems like a tricky way to evaluate functions, but it will prove advantageous later that all functions that the calculator knows are available through a well-defined interface. It is also necessary for us to be able to recognize objects as `Function`s. We can now find out whether the element is a function by using the `isinstance.(element, Function)`. Notice that `execute` checks that it is receiving a number. Here, the definition of numbers from the package `numbers` is used, so you will need `import numbers` in your code for this to work.

We will use Numpy to define the actual numeric calculation. The package is installed by default in most Python systems. In case you do not have this package installed, download it now with the command `pip install numpy`. `numpy` must then be imported by the program. Numpy contains many mathematical functions, such as `numpy.exp` and `numpy.log`. You can give your calculator access to them by wrapping them in the `Function` class like this:

```
exponential_func = Function(numpy.exp)
sin_func = Function(numpy.sin)
print(exponential_func.execute(sin_func.execute(0)))
```

Implementation – Part 2:
Implement `Function` - or use the implementation above. Try the self-test and check for the right result.

2.3 Operator

In the same way that we wrap functions, we will also wrap the calculation operators. You must create a class **Operator** that executes one of **numpy.add**, **numpy.multiply**, **numpy.divide**, and **numpy.subtract**. An **Operator** is *almost* like a **Function**, but there are two important differences: First, an operator must have two inputs to its **execute** method. Secondly, the class must know how strong the operator is. For example, the expression "`1 add 2 multiply 3`" should give `7` instead of `9`. We evaluate from left to right, but since multiplication is stronger than addition, we have to perform `2 multiply 3 = 6` first, then `1 add 6 = 7`. Below is an example usage of the **Operator** class:

```
add_op = Operator(operation=numpy.add, strength=0)
multiply_op = Operator(operation=numpy.multiply, strength=1)
print(add_op.execute(1, multiply_op.execute(2, 3)))
```

Implementation – Part 3:
Implement **Operator**, and verify that everything works using the test.

2.4 Calculator

The main class in this assignment is the **Calculator** class. The first thing to do is to create **Calculator**'s **__init__** method, where you give the calculator access to the operators and functions. It is advisable to have these in Python dictionaries, where you use the name of the operation as the key. In addition, the calculator needs storage space for its calculation task. When your system is going through a calculation task, it must iterate through the expression from left to right, so to make the implementation simple we use a "first-in first-out" queue for this. You have already implemented this, so now the **Queue** class is going to work.

For example, **__init__** might look like this:

```
def __init__(self):
    # Define the functions supported by linking them to Python
    # functions. These can be made elsewhere in the program,
    # or imported (e.g., from numpy)
    self.functions = {'EXP': Function(numpy.exp),
                      'LOG': Function(numpy.log),
                      'SIN': Function(numpy.sin),
                      'COS': Function(numpy.cos),
                      'SQRT': Function(numpy.sqrt)}

    # Define the operators supported.
    # Link them to Python functions (here: from numpy)
    self.operators = {'ADD': Operator(numpy.add, 0),
                      'MULTIPLY': Operator(numpy.multiply, 1),
                      'DIVIDE': Operator(numpy.divide, 1),
                      'SUBTRACT': Operator(numpy.subtract, 0)}

    # Define the output-queue.
    # The parse_text method fills this with RPN.
    # The evaluate_output_queue method evaluates it
    self.output_queue = Queue()
```

You may also add synonyms for the operators. For example, both 'PLUS' and 'ADD' are bound to `numpy.add`, 'MINUS' and 'SUBTRACT' to `numpy.subtract`, and 'TIMES' and 'MULTIPLY' to `numpy.multiply`.

You can check that it works by running this:

```
calc = Calculator()
print(calc.functions['EXP'].execute(
      calc.operators['ADD'].execute(1,
      calc.operators['MULTIPLY'].execute(2, 3))))
```

Implementation – Part 4:
Implements the class and check that everything works by using the test.

3 Evaluate "Reverse Polish Notation"

"Reverse Polish Notation" (RPN) is a way of writing calculations that do not require parentheses. The rule is that an operator or function comes after its operand(s). So, instead of typing "`1 add 2`", in RPN, you would type "`[1, 2, add]`", and "`exp(7)`" becomes "`[7, exp]`". As long as the operations are sorted correctly, RPN can describe all kinds of expressions without using parentheses; for example, "`(1 add 2) multiply 3`" is given as "`[1, 2, add, 3, multiply]`" in RPN.

Now you need to create the part of the calculator that gets into a task in RPN and solves it. When implementing RPN, the algorithm is quite simple. The calculation task comes in as a queue, and we use the **`self.output_queue`** that was defined in **`__init__`** for this. Each item in the queue is a number, a function, or an operator. To do the calculation, you need a **`Stack`** for intermediate storage.

The pseudo-code for evaluating RPN is as follows:

1. Go through each item in the queue by popping them one by one:

   (a) If the item is a number, push it on the stack. You can check the type of the item with **`isinstance`**.

   (b) If the item is a function, pop one element of the stack and evaluate the function with that element (which is a number if the RPN syntax is correct). You push the result on the stack.

   (c) If the item is an operator, pop two items from the stack. Do the operation with these two elements and push the result back onto the stack. Watch the order of the items: For example, if your stack has the items `[2, 1]` and you are going to subtract, the answer is given as `2 subtract 1=1` and not `1 subtract 2=-1`. Using **`value=element.execute(stack.pop())`** is thus wrong; you must first pop the two elements into local variables, then execute with them in the correct order).

2. When the queue is empty, there is one item on the stack. This is the answer you need to return.

This example below shows how we calculate that `exp(1 add 2 multiply 3) = 1096.63` by handling the queue `[1, 2, 3, multiply, add, exp]`.

Element	Handling	Stack
1	`stack.push( 1 )`	$\boxed{1}$
2	`stack.push( 2 )`	1 , $\boxed{2}$
3	`stack.push( 3 )`	1 , 2 , $\boxed{3}$
`multiply`	`multiply .execute( 2 , 3 )` ⇒ `6`	1 , $\cancel{2}$, $\cancel{3}$
	`stack.push( 6 )`	1 , $\boxed{6}$
`add`	`add .execute( 1 , 6 )` ⇒ `7`	$\cancel{1}$, $\cancel{6}$
	`stack.push( 7 )`	$\boxed{7}$
`exp`	`exp .execute( 7 )` ⇒ `1096.63`	$\cancel{7}$
	`stack.push( 1096.63 )`	$\boxed{1096.63}$

Implementation – Part 5:
Implement the RPN calculation and create a test for your system as calculate the example and verify that the answer is `exp(7)`≈`1096.63`. To do the test, you must build the **`calc.output_queue`**. For this, you can use operations such as **`calc.output_queue.push(7.)`** and **`calc.output_queue.push(calc.functions['EXP'])`**

4 From "normal" notation to RPN

The calculator does not make much use if it can only handle RPN input. In the last part of the assignment we will create a parser that gets a string as input and translate it into RPN. Once in place we can then evaluate the text and we are able to do calculations automatically. The parser should recognize the four operator types, all the supported functions, and be able to trigger arbitrarily nested parentheses. In this section we create the part that gets the elements in the math (numbers, operators, functions, parentheses) in the "usual way" (called normal notation queue) and convert it into RPN. In the next section we will discuss how to convert input string the normal notation queue.

To build the RPN queue, we will use the *shunting-yard* algorithm. This algorithm assumes that you have a queue where each element is either a number, an operation, a function, a left parenthesis, or an right parenthesis. The algorithm looks at the elements one by one ("first-in, first-out") and builds an output queue. Some items need to be temporarily stored on an operator stack to make sure the order of the elements is correct.

The pseudo-code is as follows:

1. Review each item in the input queue:
   (a) If the element `elem` is a number, push it on the output queue.
   (b) If `elem` is a function, push it on the operator stack
   (c) If `elem` is a left parenthesis, push it on the operator stack.
   (d) If `elem` is an right parenthesis:
      - Pop items from the operator stack one by one and push them to the output queue until the top element on the operator stack is a left parenthesis.
      - If there is a left parenthesis at the top of the operator stack, pop the left parenthesis from the operator stack and discard it.
      - If there is a function at the top of the operator stack, pop the function from the operator stack onto the output queue.
   (e) If `elem` is an operator, you need to sort it into the correct location, for example, to compare the strengths of the operations. To achieve so, we move items one by one from the operator stack to the output queue until one of the following cases:

- the operator stack is empty
- the top element on the operator stack is an operator weaker than `elem` (i.e., has lower precedence)
- the top element on the operator stack is a left parenthesis

After moving, push `elem` on the operator stack.

2. Pop each item on the operator stack and push it on the output queue.

The following example shows how we translate `exp(1 add 2 multiply 3)` to RPN:

elem	**output queue**	**operator stack**
exp		$\boxed{\texttt{exp}}$
(		exp, $\boxed{\texttt{(}}$
1	$\boxed{\texttt{1}}$	exp, (
add	1	exp, (, $\boxed{\texttt{add}}$
2	1, $\boxed{\texttt{2}}$	exp, (, add
multiply	1, 2	exp, (, add, $\boxed{\texttt{multiply}}$
3	1, 2, $\boxed{\texttt{3}}$	exp, (, add, multiply
)	1, 2, 3, $\boxed{\texttt{multiply}}$	exp, (, add, ~~multiply~~
	1, 2, 3, multiply $\boxed{\texttt{add}}$	exp, (, ~~add~~
	1, 2, 3, multiply add	exp, ~~(~~
	1, 2, 3, multiply, add, $\boxed{\texttt{exp}}$	~~exp~~

Note that there are never right parentheses on the operator stack, and that there are never parentheses at all in the output queue.

The next example translates `2 multiply 3 add 1`. Multiplication has higher precedence than addition, so when we put `add` on the operator stack, and `multiply` must first be moved to the output queue.

`elem`	output queue	operator stack
2	$\boxed{2}$	
`multiply`	2	$\boxed{\texttt{multiply}}$
3	2, $\boxed{3}$	`multiply`
`add`	2, 3, $\boxed{\texttt{multiply}}$	~~multiply~~
	2, 3, `multiply`	$\boxed{\texttt{add}}$
1	2, 3, `multiply`, $\boxed{1}$	`add`
	2, 3, `multiply`, 1, $\boxed{\texttt{add}}$	~~add~~

Implementation – Part 6:
Implement the shunting yard, and create a test to check that the implementation works.

5 Text-parser

Now create a text parsing method in the `Calculator` that recognizes the different parts of the input text and produces the element list that the "shunting-yard" algorithm can handle. The method receives a text string, such as "`2 multiply 3 add 1`", and produces `[2, multiply, 3, add, 1]`. The result is a list of Python objects, where 2 is a `float`, `multiply` is an `Operator`, and so on.

It is recommended to use regular expressions (regex) here, which are implemented in the Python package `re`; the package is part of the standard distribution of Python so you can import it directly. If you do not remember how the regular expressions are defined, it may be worth checking a tutorial online[1]. For example, `match = re.search("^[-0123456789.]+", txt)` will look at the text string given in `txt` and look for at least one of the elements of the symbols "-", "." or the numbers from "0" to "9". That means, for example, we will recognize "-322.7623" as a match. When the search string starts with ^ it means we are looking from *the start* of `txt`, so we will not get a match if `txt="exp(-322.7623)"`. When there is at least a match, `match.end(0)` is the position of the first element after the match in the text. For example, `match.end(0)` is 5 if `txt="-37.4 add 91"`. So `txt[match.end(0):]` is the part of `txt` that still needs to be parsed. `match.group(0)` is the text that matches, so `float(check.group(0))` returns the number value "translated" from the string. If there is no match, `match` is `None`.

A useful shortcut in `re.search` is that we can look for several sub strings simultaneously if we enter them as a composite string with "|" between the parts. If `self.functions` is a dictionary with all the functions your program supports as keys, then

```
targets = "|".join(["^" + func for func in self.functions.keys()])
match = re.search(targets, txt)
```

will look for all the functions you have defined. Because we are watching match only from the beginning of the text (notice "^"), this works as long as we assume that there are no multiple functions where a function name is the beginning of another. This means that if you have defined "`SQRT`" to calculate square root, you cannot use "`SQR`" to square. Instead, you can use "`SQUARE`" for example.

Before you start parsing, remove all spaces from the text and make it uppercase. You do this with `text = text.replace (" ", "").upper()`.

As you may have noticed in the document, we did not use "+", "−", "∗", and

[1]See e.g. `https://regexone.com/`

"/" in the input string. Instead we have used `add`, `subtract`, `multiply` and `divide`, respectively. This is to avoid the ambiguity between the negative sign and the subtract operator. It is possible to setup rules to differentiate the two, but this rather complicates the project. For simplicity we just require that operators in the input string must use their names instead of symbols. The operator names can be found in **`Calculator.__init__`** method. That means we will get a task text such as "`1 SUBTRACT -2`", but not "`1 - -2`".

The method that implements your parser should take a text string as input, search through the text, and build a list of all the elements contained in the input text. It should return a list of objects of the correct type: numbers are coded as **`float`**, functions as **`Function`**, the spreadsheets as **`Operator`**, and the parentheses as **`str`**. The list produced should be such that your implementation of the shunting-yard algorithm can use it as input.

Implementation – Part 7:
Implement the parser as described. Create a test where you submit texts that you know should be able to be parsed and check that the returned items are of the correct type.

6 Put everything together

Now you have all the parts you need and all that's left is to put them together.

Implementation – Part 8:
Create a master routine **`Calculator.calculate_expression(txt)`** that takes a text string as input and evaluates it. Test your system with various input strings, such as

- "`EXP (1 add 2 multiply 3)`"
- "`((15 DIVIDE (7 SUBTRACT (1 ADD 1))) MULTIPLY 3) SUBTRACT (2 ADD (1 ADD 1))`"

7 What is required to pass the project

To pass this project you must:

- Solve all the required part assignments.
- You must do the work alone and have it approved by the deadline (see page 1)
- The system must be implemented with object-oriented Python, with pylint grade at least 8.0

— o —