Programming Test

Implement a Reverse Polish Notation calculator in Python

PART I (please observe, there are TWO parts in this exercise)

Background

RPN is an alternative method for expressing calculations where operators (such as +, -, * etc) are placed after their arguments, rather than in between them. We typically use in-fix to express arithmetic, like:

```
10 / 5
```

But in RPN, we use post-fix:

```
10 5 /
```

Note that this means that there are no operator precedence rules and subsequent brackets to manipulate them:

```
( 1 + 2 ) * 3
```

becomes

```
1 2 + 3 *
```

Implementation

Typically, this would be implemented by way of a stack - numbers are pushed onto the stack, operators pop the arguments they need, evaluate and push the result back.

The solution requires a module which has a minimum of two or three public methods

```
push( string )
push( float )
pop( )
```

The push string should expect either individual numbers or operators and should route valid operators to functions which do the work required. Assuming a method based system and c++, / would be implemented as:

Code:

```
div( )
{
    float b = pop( );
    float a = pop( );
    push( a / b );
}
```

Note:

1. the pop removes and returns the top of stack
2. for non-commutative operators (like / and -), we need to ensure that we map to the in-fix and as a general rule push(pop( ) / pop( )) will be treated as undefined behaviour because languages typically don't specify the order of evaluation - it is safer to implement operators as shown.

Testing:
Input: `1 2 + 3 *`
output: 9.
Input 1 2 `* 3 +`
output: 5

For node js the input full be provided from console and will evaluate the result when press return..

For other cases take the input in text field and display on a simple page.

PART II : Infix

As a further challenge that builds on everything here, consider developing an infix to RPN converter with all the ( ) and operator precedence rules honoured. If you do this you will pass the exercise with 3 gold stars.

Goal

When you solve this task, think about the following aspects:

- Make it easy to add more operators in the future. Less code is preferred.
- Try to combine **functional programming** best practices like separation of concerns, with the demand of easy-to-read and easy to maintain code.
- Make sure your implementation scale. If we add thousands of operations to the system, what will happen?
- Take care about about error handling and error reporting - divide by 0 for instance.
- The project should have code coverage through Unit test. The more the merrier.
- The solution **should** be easy to build and to run. The project **must** contain a readme.md file explaining the steps.

Bonus:

Feel free to implement additional operators like sin, cos, tan, square, sqrt etc. This will be counted as bonus.