Laboratories

Weeks 39-40

DM536 Introduction to Programming
DM562 Scientific Programming
DM857 Introduction to Programming
DS830 Introduction to Programming

1 Overview

In this laboratory you will implement a program that computes arithmetic expressions.

1.1 Expressions

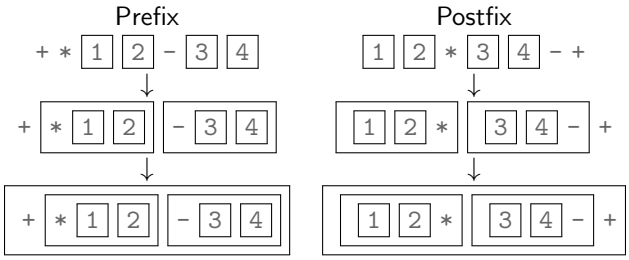
Expressions are formed by the following elements (called *tokens*) separated by spaces:

- integer numbers (`1`, `-1`, etc.) and
- binary arithmetic operators of Python (`+`, `-`, `*`, `/`, `//`, `%`, `**`).

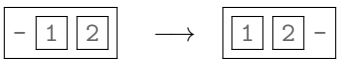
For example, `1 + 2` is a valid expression whereas `1+2` is not. We all are familiar with the infix notation for arithmetic expressions where additions, sums, etc. are written with the binary operator between its operands (`1 + 2 * 3`) hence the name "infix notation". This notation requires parenthesis and precedence levels and this adds complexity to the algorithms for evaluating them. The prefix and postfix notations were introduced to avoid this complexity:

- In prefix notation, operands always follow operators e.g., the (infix) expression `1 * 2 + (3 - 4)` is written in prefix notation as `+ * 1 2 - 3 4`.
- In postfix notation, operations always operands e.g., the (infix) expression `1 * 2 + (3 - 4)` is written in postfix notation as `1 2 * 3 4 - +`.

In both cases we can identify the subexpressions an operation is applied to without parenthesis or priorities: start by putting boxes around numbers and then around a binary operations and the first two boxes to its right (if prefixed) or to its left (if postfixed) until we reach the sides of the expression (or find that something is amiss):



This suggests how to translate one notation into the other: move every operator to the opposite side of the two boxes representing its operands.



Below are some examples of expressions in the three notations.

Infix	Prefix	Postfix	Result
`-1`	`-1`	`-1`	-1
`-1 + 2`	`+ -1 2`	`-1 2 +`	1
`-1 + 2 * 2`	`+ -1 * 2 2`	`-1 2 2 * +`	3
`1 * (2 // 3)`	`* 1 // 2 3`	`1 2 3 // *`	0

1.2 Structure

The program must be organised in the following files (the next Sections discuss how to proceed to implement them):

- `calculator.py` This module contains the program for evaluating expressions in prefix or postfix notation, depending on the user choice.
- `prefix.py` This module offers functions for evaluating and validating arithmetic expressions in prefix notation.
- `postfix.py` This module offers functions for evaluating and validating arithmetic expressions in postfix notation.
- `utils.py` This module offers definitions that are helpful for implementing the functionalities of `prefix.py` and `postfix.py`.

You can find implementations of these modules on the course material. These are intentionally unreadable since in this lab you will have to write your own version of some of them.

1.2.1 Prefix

This module provides the following functions.

`is_valid(expression)` This function returns `True` if `expression` is a valid arithmetic expression in prefix notation, `False` otherwise.

`eval(expression)` This function evaluates a valid arithmetic expression and returns the result.

`to_postfix(expression)` Translates a valid prefix expression into postfix notation. For instance, `to_postfix('+ * 1 2 3')` return '1 2 * 3 +'.

The module can also be executed directly as a program. In this case it must ask the user to enter an expression and if it is valid it prints its value.

1.2.2 Postfix

This module is similar to `prefix`; it provides the following functions.

`is_valid(expression)` This function returns `True` if `expression` is a valid arithmetic expression in prefix notation, `False` otherwise.

`eval(expression)` This function evaluates a valid arithmetic expression and returns the result.

`to_postfix(expression)` Translates a valid prefix expression into postfix notation. For instance, `to_postfix('+ * 1 2 3')` return '1 2 * 3 +'.

The module can also be executed directly as a program. In this case it must ask the user to enter an expression and if it is valid it prints its value.

1.2.3 Utils

This module provides the following functions.

`is_binary_op(op)` This function returns `True` if `op` is `'+'`, `'-'`, `'*'`, `'/'`, `'//'`, `'%'`, or `'**'`, and returns `False` otherwise.

`eval_binary_op(op,l,r)` If `op` is a binary arithmetic operator (`is_binary_op(op)` returns true) then, the function the result of applying the corresponding operations to `l` and `r` (both numbers) in that order. For instance, `eval_binary_op('+',5,6)` returns `11`.

This module provides the following classes.

`Tokenizer(expression)` This class allows you to incrementally consume the tokens of an expression given as a string. The expression is read left-to-right and tokens are separated by white spaces (e.g., the tokens in `'a + 6'` are, `'a'`, `'+'`, and lastly `'6'`). Initialisation takes the expression (e.g., `Tokenizer('a + b c * 15')`). The class provides the following methods:

- `has_next(self)` returns `True` if the expression has more tokens `False` otherwise.
- `next(self)` removes and returns the next token in the expression.
- `remainder(self)` returns a string containing the token left in the expression.
  - Preconditions: there is at least a token (`has_next(self)` returns `True`).
  - Errors: Raises `IndexError` if there are no tokens.

`Stack()` This class allows you to store items in a Last-In/First-Out manner. Instances are initially empty (they have no items) The class provides the following methods:

- `is_empty(self)` returns `True` if the stack has no elements and `False` otherwise.
- `push(self,item)` adds `item` to the top of the stack.
- `pop(self)` removes and returns the item at the top of the stack.
  - Preconditions: the stack is not empty (`is_empty(self)` returns `False`).
  - Errors: Raises `IndexError` if the stack is empty.

2 Calculator

In this part of the lab you will write the program `calculator.py` using the modules `prefix` and `postfix`. Create a file `calculator.py`, obtain `utils.py`, `prefix.py`, and `postfix.py` from the course material on itsLearning and place them in the same folder of your `calculator.py`.

When run, `calculator.py` must behave as follows.

1. Ask the user to select between prefix and postfix notation.
2. Ask the user to input an arithmetic expression.
3. Check if the expression is valid in the selected notation using the functionality of the corresponding module.
   - If the expression is valid, it prints the result of its evaluation.
   - If the expression is not valid, it informs the user.
4. Terminate.

3 Prefix

In this part of the lab you will write the module `prefix` using `utils`. When you replace the implementation obtained from the course material with yours `calculator.py` should run and behave as expected.

3.1 Warming up

These are some exercises to familiarise yourself with the elements of module `utils`.

1. Write a function `print_tokens(expression)` that for each token prints a line with the token. (Hint: use Tokenizer and an inner function)

   ```
   >>> print_tokens('a b c')
   a
   b
   c
   ```

2. Write a function `print_tokens_remainder(expression)` that for each token prints a line with the token, a colon, and the remainder at that point.

   ```
   >>> print_tokens_remainder('a b c')
   a : b c
   b : c
   c :
   ```

3. Write a function `print_tokens_reverse(expression)` that prints each token of `expression` on separate lines in the opposite order.

   ```
   >>> print_tokens_reverse('a b c')
   c
   b
   a
   ```

4. Consider the following function.

   ```
   def f(expression):
     tokens = Tokenizer(expression)
     def process_tokens(e):
       if tokens.has_next()
         token = tokens.next()
         if e != token:
           print(e)
         process_tokens(token)
     process_tokens('+')
   ```

   Without executing them, write the output of the following statements.

   (a) `>>> f('* / +')`

   (b) `>>> f('+ * /')`

   (c) `>>> f('+ * *')`

   (d) `>>> f('* + *')`

5. Write a function `print_op_tokens(expression)` that prints only tokens that represent binary operations or integers. (Hint: use method `isnumeric` of class `str` and function `is_binary_op` of module `util`.)

```
>>> print_valid_tokens('+ 1 b 2 * 5.2')
+
1
2
*
```

6. Write a function `print_first_int(expression)` that prints the first integer from the left, if any, and the remainder separated by a colon.

```
>>> print_first_int('+ 1 * 2 3')
1 : * 2 3
```

7. Write a function `sum_all_ints(expression)` that returns the sum of all integers the expression (`0` if there is none)

```
>>> sum_all_ints('+ 1 b 2 * 5.2')
3
>>> sum_all_ints('+ b * 5.2')
0
```

8. Consider the following function.

```
def f(expression):
  tokens = Tokenizer(expression)
  def process_tokens():
    if tokens.has_next():
      token = tokens.next()
      if token == '+':
        l = process_tokens()
        r = process_tokens()
        print(l, token, r)
        return '( ' + l + ' + ' + r + ' )'
      else:
        return token
    else:
      return '?'
  print(process_tokens())
```

Without executing them, write the output of the following statements.

1. `>>> f('+')`
2. `>>> f('+ a b')`
3. `>>> f('+ + a b c')`
4. `>>> f('+ a + b c')`
5. `>>> f('+ +')`
6. `>>> f('+ a b c')`

3.2 Evaluating and validating expressions

Create a file `prefix.py` and copy to the same folder `utils.py` (from the course material). Implement the following functions to your `prefix.py`.

1. `eval(expression)` that returns the result of evaluating expression under the assumption that it is a valid arithmetic expression. (Hint: solve Exercise 8 from the previous section before doing this.)

2. `is_valid(expression)` that checks if the string expression is a valid arithmetic expression in prefix notation. (Hint: interpret integers as `True` and operations as `and`.)

3. (Optional) `to_postfix(expression)` that given a valid expression in prefix notation returns the equivalent expression in postfix notation.

Test your implementation using the shell (you can use `doctest`).

Copy `postfix.py` and `calculator.py` (course material)to the same folder of your `prefix.py` (and `utils.py`). Test your implementation by running the `calculator.py`.