CM10227/50258 Principles of Programming
Coursework 1: SRPN

1 Introduction

This document provides the specification for the first Principles of Programming coursework.

System requirements You can use any environment for the development of your script, but we must be able to compile and run your code using **Python 3.8** (this is the version on **repl.it**) without requiring the installation of additional libraries, modules or other programs.

Learning Objectives At the end of this coursework you will be able to design and write a medium-sized program using the appropriate procedural software techniques of data encapsulation and decomposition. You will also be able to understand, construct and use key data structures in a program.

Plagiarism This is an individual assignment. Using someone else's solutions or sharing your solutions with others both constitute an academic offence.

Questions Questions regarding the coursework can be posted on the Moodle forums, sent to the programming1@lists.bath.ac.uk mailing list, or asked in labs/lectures.

2 Saturated Reverse Polish Notation (SRPN) Calculator

Whilst performing some maintenance on a legacy system you find that it makes use of a program called **SRPN**. **SRPN** is not documented and no one seems to know who wrote it. It is your task to reverse-engineer the program and rewrite it in Python.

SRPN is a reverse Polish notation calculator with the extra feature that all arithmetic is saturated, i.e. when it reaches the maximum value that can be stored in a variable, it stays at the maximum rather than wrapping around.

You can find the SRPN calculator, as well as any other resources for this coursework, on Moodle.

Your task is to write a program which matches the functionality of SRPN as closely as possible. Note that this includes not adding or enhancing existing features.

Read the below description of a RPN calculator. You will be provided a basic specification for the SRPN calculator to get you started. There is then information to help you investigate and find more obscure details of the calculator.

3 Reverse Polish notation calculator

Reverse Polish notation (RPN) is a machine-oriented way of writing arithmetic expressions. Instead of operators being **infix**, as in `3 + 4`, they are written **postfix**, `3 4 +`. Here are a few more examples:

Regular notation	Reverse Polish notation
`(1+3)*7`	`1 3 + 7 *`
`4*(2-5)`	`4 2 5 - *`
`((1+2)+3)+4`	`1 2 + 3 + 4 +`
`1+(2+(3+4))`	`1 2 3 4 + + +`
`8^(2*3)`	`8 2 3 * ^`

Reverse Polish notation has the following two features that make it excellent for implementation on a machine:

- In postfix notation, no parentheses are needed, as you can see from the examples above.
- An expression can be interpreted directly as a sequence of machine instructions that work with a stack of numbers.

We will use this stack implementation in our **SRPN** application. It works as follows. A **stack** data structure is a sequence of values that comes with two operations:

- `push(v)`: adds the item `v` to the top of the stack
- `pop()`: removes the top item off the stack and returns it

Note that `push` takes an argument and has no return value, while `pop` takes no argument but does have a return value. Depending on the implementation, a stack may have further operations, but these are the two characteristic ones. We will use a stack of numbers, and write it as a sequence with the top to the right. For example, pushing `6` to the stack below has the following result.

`1 2 3 4 5` $\xrightarrow{\texttt{push(6)}}$ `1 2 3 4 5 6`

Popping from the stack below returns `5`, and modifies the stack as follows.

`1 1 2 3 5` $\xrightarrow{\texttt{pop() == 5}}$ `1 1 2 3`

An RPN expression can be viewed directly as a sequence of instructions to work with a stack of numbers.

- A number `n` is pushed to the stack.

- An operator, e.g. +, pops the necessary arguments from the stack and pushes the result.

Below is an example computation for the arithmetic expression `(1+3)*(5-7)`, which is `1 3 + 5 7 - *` in reverse Polish notation. Each line is a single step, which processes the first token of the remaining RPN expression and updates the stack. The stack starts out empty, and the expression ends up empty. Note how the step for + adds the top two items on the stack, that for * multiplies them, and that for - subtracts the top element from penultimate one. The whole computation correctly gives the result `-8`.

Stack	Expression
	`1 3 + 5 7 - *`
`1`	`3 + 5 7 - *`
`1 3`	`+ 5 7 - *`
`4`	`5 7 - *`
`4 5`	`7 - *`
`4 5 7`	`- *`
`4 -2`	`*`
`-8`	

Starting specification

Your starting specification for **SRPN** calculator is as follows. Note that this will get you a starting implementation which will pass many of the marking tests, but which you should then build upon. Implementing this **starting specification only** is not enough to get full marks for functionality!

- Your calculator must have an internal stack that stores integers and starts out empty.
- Your calculator must accept as input a single line of the prompt.
- If the input is an integer it must be pushed to the stack.
- If the input is one of the following operator symbols, and if there are at least two items on the stack, your calculator should pop the top two items on the stack, apply the operator, and push the result.

  `+ - * / % ^`

  The operators `/`, `%`, and `^` are integer division, modulus, and "power" respectively. For the operators `-`, `/`, `%`, and `^` make sure the arguments are in the correct order: e.g. "`4 3 -`" should give `1` not `-1`.
- If the input is the symbol = your calculator should display the top element of the stack on the prompt. It should not alter the stack.

4 Running the SRPN Legacy program and testing for functionality

You have been provided with a repl.it link which allows you to run and interact with the SRPN program. You should spend time inputting data and observing the output to understand what the program does.

On the following pages we have provided four categories of tests:

- Single operations
- Multiple operations
- Saturation
- Obscure functionality

In each category we have provided **example** test data that you should input and observe the output of. This test data will help you to make more sense of the starting specification.

1. Single operations. The program must be able to input at least two numbers and perform one operation correctly and output

```
Input:

10
2
+
=

Input:

11
3
-
=

Input:

9
4
*
=

Input:

11
3
/
=

Input:

11
```

3
%
=

2. Multiple operations. The program must be able to handle multiple numbers and multiple operations

```
Input:

3
3
*
4
4
*
+
=

Input:

1234
2345
3456
d
+
d
+
d
=
```

3. Saturation. The program must be able to correctly handle saturation

```
Input:

2147483647
1
+
=

Input:

-2147483647
1
-
=
20
-
=

Input:

100000
0
-
d
*
=
```

4. Obscure functionality. The program includes the less obvious features of SRPN. These include but are not limited to...

```
Input:

1
+

Input:

10
5
-5
+
/

Input:

11+1+1+d

Input:

# This is a comment #
1 2 + # And so is this #
d

Input:

3 3 ^ 3 ^ 3 ^=

Input:

r r r r r r r r r r r r r r r r r r r r d r r r d
```

4.1 Investigating and implementing more of SRPN

This is an investigative assignment. Once you have tried the inputs described above, you should spend some time trying other inputs into the calculator. The legacy **SRPN** calculator has a lot of functionality, some of it obscure. We are looking for you to recreate the same calculator, *not to fix its flaws or to add new functionality*.

It is important that your outputs are the same as the SRPN calculator. For example, SRPN's "Stack overflow." is not the same as "stack overflown"—note the different case of the verb, lack of capitalisation and missing full stop.

We will run the above tests, as well as others that are similar or more obscure. You are not expected to find and implement every feature. Start by implementing functionality to meet the basic specification and above tests. Then add bits of functionality one at a time, continually testing that your solution works before adding more.

It is possible to spend many hours working on this assignment to attempt to replicate it perfectly—do not do this. You should manage your time to implement the core functionality with some additional features if possible, then dedicate time to commenting your code and improving your code quality.

4.1.1 Automated tests

You have been provided with some starting code `srpn.py` and a test script `mark-code.py` that will run the above example inputs. These are available on Moodle. As you develop your code, you should run it against the test script to check that it meets expected outputs.

You can run the test script in repl.it by going to the *Shell* window and entering `python mark-code.py`. A guide on this is available on Moodle.

This assignment will be marked against a test script similar to the one provided. It will run additional tests covering the same set of functionality, as well as some more advanced features.

5 Submission Instructions

Your source (`.py`) files must be submitted in a single zip, with all files in the root of the zip. You must not include subfolders within the zip file. Your zip file name must be of the form:

`username-srpn.zip`, e.g. `abc12-srpn.zip`.

Your .zip file must be submitted to the Moodle assignment page by the submission deadline shown. Submissions received after this deadline will be capped at 40% if received within 5 working days. Any submissions received after 5 working days will marked at 0%. If you have a valid reason for an extension, you must submit an extension request through your Director of Studies – unit leaders cannot grant extensions.

You should leave yourself time to download your file from Moodle, extract it, and check that you have attached the correct file, with the content that you want to be marked. **You** are responsible for checking that you are submitting the correct material to the correct assignment.

The submission deadline is 8pm on Friday 4th November 2022.

6 Assessment

6.1 Conditions

The coursework will be conducted individually. Attention is drawn to the University rules on plagiarism. While software reuse (with correct referencing of the source) is permitted, we will only be able to assess your contribution.

6.2 Marking

This coursework is worth 50% of the unit.

Key considerations for marking the code will be: compiling, running with expected input, robustness (handling incorrect user input), module design, proper use of data encapsulation and decomposition. You stand to gain marks for well-structured and documented code. Marks are also awarded for well-named variables or functions and consistent indentation.

6.3 Marks Table

Below is a breakdown of marks, with descriptions on how each criteria is met.

Criteria	Max Score	Description
Running with expected input	max 50	Code replicates the functionality of the SRPN program to read input and produce expected outputs. This includes handling incorrect user inputs.
Code quality and design	max 40	Code makes proper use of coding techniques, such as data encapsulation and generalisation, use of functions and return statements, and robustness.
Code formatting and commenting	max 10	Appropriate variable and function names. Code is well-structured and indented. Appropriate comments are used to document functions.

In cases where it is not clear from the assignment how it should be marked, you may be called on to explain or demonstrate your program. Such cases include but are not limited to suspected plagiarism.

6.4 Grades

Marks and feedback will be made available within 3 working weeks of the submission.

The following is a guideline for marks:

- **Above 40%** – Pass.
- **50-69%** – Very good. (2:2-2:1 / Pass/Merit range)
- **70%+** – Outstanding. (1st / Distinction range)