Assignment 2: Slippy

version: 0.2 last updated: 2022-07-20 12:00

Aims

This assignment aims to give you:

- practice in `Python` programming generally.
- a clear and concrete understanding of `sed`'s core semantics.

Introduction

Your task in this assignment is to implement **Slippy**.

Slippy stands for **[S]ed [L]anguage [I]nterpreter in [P]ure [PY]thon**.

A subset of the important Unix/Linux tool Sed.

You will do this in `Python`.

`Sed` is a *very* complex program that has many commands.
You will implement only a few of the most important commands.
You will also be given a number of simplifying assumptions, which make your task easier.

Slippy is a POSIX-compatible subset of `sed` with extended regular expressions (EREs).
On CSE systems you would run **`sed -E`**

You must implement **Slippy** in Python only.
See the **Permitted Languages** section below for more information.

the material in the lecture notes will not be sufficient by itself to allow you to complete this assignment.
You may need to search the command-line and online documentation for Python, Sed, and Regex
Being able to search documentation efficiently for the information you need is a *very* useful skill for any kind of computing work.

Reference implementation

Many aspects of this assignment are not fully specified in this document;
instead, you must match the behaviour of the reference implementation: **`2041 slippy`**
Provision of a reference implementation is a common method to provide or define an operational specification,
and it's something you will likely need to do after you leave UNSW.

Discovering and matching the reference implementation's behaviour is deliberately part of the assignment,
and will take some thought.

If you discover what you believe to be a bug in the reference implementation, report it in the class forum.
Andrew and Dylan may fix the bug, or indicate that you do not need to match the reference implementation's behaviour in this case.

Slippy Commands

Subset 0

In subset 0 `slippy` will always be given a single Slippy command as a command-line argument.
The Slippy command will be one of 'q', 'p', 'd', or 's' (see below).

The only other command-line argument possible in subset 0 is the **`-n`** option.

Input files will not be specified in subset 0.

For subset 0 `slippy` need only read from standard input.

Subset 0: q - quit command

The Slippy **q** command causes `slippy.py` to exit, for example:

```
seq 1 5 | 2041 slippy '3q'
1
2
3
seq 9 20 | 2041 slippy '3q'
9
10
11
seq 10 15 | 2041 slippy '/.1/q'
10
11
seq 500 600 | 2041 slippy '/^.+5$/q'
500
501
502
503
504
505
seq 100 1000 | 2041 slippy '/1{3}/q'
100
101
```

`slippy` commands are applied to input lines as they are read.

The **q** command means `slippy` may not read all input.

For example, the command prints an "infinite" number of lines containing (by default) "yes".

```
yes | 2041 slippy '3q'
y
y
y
```

This means `slippy` can not read all input first, e.g. into a list, before applying commands.

Subset 0: p - print command

The Slippy **p** commands prints the input line, for example:

```
seq 1 5 | 2041 slippy '2p'
1
2
2
3
4
5
seq 7 11 | 2041 slippy '4p'
7
8
9
10
10
11
seq 65 85 | 2041 slippy '/^7/p'
65
66
67
68
69
70
```

Subset 0: d - delete command

The Slippy **d** command deletes the input line, for example:

```
seq 1 5 | 2041 slippy '4d'
1
2
3
5
seq 1 100 | 2041 slippy '/.{2}/d'
1
2
3
4
5
6
7
8
9
seq 11 20 | 2041 slippy '/[2468]/d'
11
13
15
17
19
```

Subset 0: s - substitute command

The Slippy `s` command replaces the specified regex on the input line.

```
seq 1 5 | 2041 slippy 's/[15]/zzz/'
zzz
2
3
4
zzz
seq 10 20 | 2041 slippy 's/[15]/zzz/'
zzz0
zzz1
zzz2
zzz3
zzz4
zzz5
zzz6
zzz7
zzz8
zzz9
20
seq 100 111 | 2041 slippy 's/11/zzz/'
100
101
```

The substitute command can be followed optionally by the modifier character `g`, for example:

```
echo Hello Andrew | 2041 slippy 's/e//'
Hllo Andrew
echo Hello Andrew | 2041 slippy 's/e//g'
Hllo Andrw
```

`g` is the only permitted modifier character.

Like the other commands, the substitute command can be given addresses to be applied to:

```
seq 11 19 | 2041 slippy '5s/1/2/'
11
12
13
14
25
16
17
18
19
seq 51 60 | 2041 slippy '5s/5/9/g'
51
52
53
54
99
56
57
58
59
60
```

Subset 0: -n command line option

The Slippy `-n` command line option stops input lines being printed by default.

```
seq 1 5 | 2041 slippy -n '3p'
3
seq 2 3 20 | 2041 slippy -n '/^1/p'
11
14
17
```

`-n` command line option is the only useful in conjunction with the `p` command,
but can still be used with the other commands.

Subset 0: Addresses

All Slippy commands in subset0 can optionally be preceded by an address specifying the line(s) they apply to.

In subset 0, this address can either be a line number or a regex.

The line number must be a positive integer.

The regex must be delimited with slash `/` characters.

Subset 0: Regexes

In subset 0, you can assume backslashes `\` do **not** appear in address or substitution regexes.

In subset 0, you can assume semicolons `;` do **not** appear in address or substitution regexes.

In subset 0, you can assume commas `,` do **not** appear in address or substitution regexes.

In subset 0, regexes are delimited with slash `/` characters, so you can assume slashes do not appear in regexes.

In subset 0 and all other subsets, you can assume the regex is correct. You do not have to check for errors in the regex.

In subset 0 and all other subsets, you can assume the regex is a POSIX-compatible extended regular expression.

In subset 0 and all other subsets, you can assume the regex is compatible with Python.
In other words, the regex can be used directly as a Python regular expression, for example passed to `re.search`, and will have the same meaning.

Subset 1

Subset 1 is more difficult. You will need to spend some time understanding the semantics (meaning) of these operations, by running the reference implementation and researching the equivalent `sed` operations.

Note the assessment scheme recognises this difficulty.

Subset 1: s - substitute command

In subset 1, any non-whitespace character may be used to delimit a substitute command, for example:

```
seq 1 5 | 2041 slippy 'sX[15]XzzzX'
zzz
2
3
4
zzz
seq 1 5 | 2041 slippy 's?[15]?zzz?'
zzz
2
3
4
zzz
seq 1 5 | 2041 slippy 's_[15]_zzz_'
zzz
2
3
4
zzz
seq 1 5 | 2041 slippy 'sX[15]Xz/z/zX'
z/z/z
2
```

Subset 1: Multiple Commands

In subset 1, multiple Slippy commands can be supplied separated by semicolons `;` or newlines. For example:

```
seq 1 5 | 2041 slippy '4q;/2/d'
1
3
4
seq 1 5 | 2041 slippy '/2/d;4q'
1
3
4
seq 1 20 | 2041 slippy '/2$/,/8$/d;4,6p'
1
9
10
11
19
20
```

```
seq 1 5 | 2041 slippy '4q
/2/d'
1
3
4
seq 1 5 | 2041 slippy '/2/d
4q'
1
3
4
```

Semicolons can not appear elsewhere in subset 1 commands.

Subset 1: -f command line option

The Slippy `-f` reads Slippy commands from the specified file, for example:

```
echo 4q   >  commands.slippy
echo /2/d >> commands.slippy
seq 1 5 | 2041 slippy -f commands.slippy
```
```
1
3
4
```

```
echo /2/d >  commands.slippy
echo 4q   >> commands.slippy
seq 1 5 | 2041 slippy -f commands.slippy
```
```
1
3
4
```

commands can be supplied separated by semicolons `;` or newlines.

Subset 1: Input Files

In subset 1, input files can be specified on the command line:

```
seq 1 2 > two.txt
seq 1 5 > five.txt
2041 slippy '4q;/2/d' two.txt five.txt
```
```
1
1
2
```

```
seq 1 2 > two.txt
seq 1 5 > five.txt
2041 slippy '4q;/2/d' five.txt two.txt
```
```
1
3
4
```

```
echo 4q   >  commands.slippy
echo /2/d >> commands.slippy
seq 1 2 > two.txt
seq 1 5 > five.txt
2041 slippy -f commands.slippy two.txt five.txt
```
```
1
1
2
```

Subset 1: Comments & White Space

In subset 1, whitespace can appear before and/or after commands and addresses.

In subset 1, '#' can be used as a comment character, for example:

```
seq 24 43 | 2041 slippy ' 3, 17  d  # comment'
```
```
24
25
41
42
43
```

On both the command line and in a command file, a newline ends a comment

```
seq 24 43 | 2041 slippy '/2/d # delete  ;  4  q # quit'
30
31
33
34
35
36
37
38
39
40
41
43
```

Subset 1: Addresses

In subset 1, `$` can be used as an address.
It matches the last line, for example:

```
seq 1 5 | 2041 slippy '$d'
1
2
3
4
seq 1 10000 | 2041 slippy -n '$p'
10000
```

Slippy can read one line of input ahead to handle `$` addresses.

In subset 1, Slippy commands can optionally be preceded by a comma-separated pair of addresses specifying the start and finish of the range of lines the command applies to, for example:

```
seq 10 21 | 2041 slippy '3,5d'
10
11
15
16
17
18
19
20
21
seq 10 21 | 2041 slippy '3,/2/d'
10
11
21
seq 10 21 | 2041 slippy '/2/,4d'
10
11
14
15
16
17
```

Comma-separated pairs of addresses can not be used with the `q` command.

Subset 1: Regexes

All the rules from Subset 0 about regex still apply, except:

In subset 1, substitute regexes are **not always** delimited with slash `/` characters,
So you can **not** assume slashes do not appear in regexes.
You **can** assume that whatever the delimiter is, it will not appear in the substitute regex.
Only substitute regexes can be delimited with other characters, address regex are always delimited by slashes.

Subset 2

Subset 2 is even more difficult. You will need to spend considerable time understanding the semantics of these operations, by running the reference implementation, and/or researching the equivalent `sed` operations.

Note the assessment scheme recognises this difficulty.

Subset 2: s - substitute command

In subset 2, the character used to delimit the substitute command may appear in the regex or replacement string.
In subset 2, backslash may appear in the regex or replacement string.
In subset 2, you can not assume the regex is correct. You need to check for errors in the regex.

Subset 2: -i command line option

The Slippy `-i` command line option replaces file contents with the output of the Slippy commands. You should use a temporary file.

```
seq 1 5 > five.txt
cat five.txt
1
2
3
4
5
2041 slippy -i /[24]/d five.txt
cat five.txt
1
3
5
```

Subset 2: Multiple Commands

In subset 2, semicolons `;` and commas `,` can appear inside Slippy commands.

```
echo 'Punctuation characters include . , ; :' | 2041 slippy 's/;/semicolon/g;/;/q'
Punctuation characters include . , semicolon :
```

Subset 2: : - label command

The Slippy `:` command indicates where `b` and `t` commands should continue execution.
There can not be an address before a label command.

Subset 2: b - branch command

The Slippy `b` command branches to the specified label, if the label is omitted, it branches to the end of the script.

Subset 2: t - conditional branch command

The Slippy `t` command behaves the same as the `b` command except it branches only if there has been a successful substitute command since the last input line was read and since the last `t` command.

```
echo 1000001 | 2041 slippy ': start; s/00/0/; t start'
101
echo 0123456789 | 2041 slippy -n 'p; : begin;s/[^ ](.)/ \1/; t skip; q; : skip; p; b begin'
0123456789
 123456789
  23456789
   3456789
    456789
     56789
      6789
       789
        89
         9
```

Subset 2: a - append command

The Slippy `a` command appends the specified text.

```
seq 5 9 | 2041 slippy '3a hello'
5
6
7
hello
8
9
```

Subset 2: i - insert command

The Slippy `i` command inserts the specified text.

```
seq 5 9 | 2041 slippy '3i hello'
5
6
hello
7
8
9
```

Subset 2: c - change command

```
seq 5 9 | 2041 slippy '3c hello'
5
6
hello
8
9
```

The Slippy `c` command replaces the selected lines with the specified text.

Subset 2 Assmptions: Regexes

In subset 2, backslash `\` may appear in regexes.

In subset 2, the character used to delimit the regex may appear in the regex itself.

Other Sed Features

You do not have to implement in Slippy sed features and commands other than those described above.

For example, sed on CSE systems provides extra commands including `{} D h H g G l n p T w W x y` which are not part of Slippy.

For example, sed on CSE systems adds extra syntax to addresses including features involving the characters: `! + ~ 0 \`. These are not part of Slippy.

For example, sed on CSE systems has a number of command-line options other than `-i`, `-n` and `-f`. These are not part of Slippy

The reference implementation implements many of these extra sed features and commands.

The marking will not test your code on these extra features and commands.

You do not have to check for these extra features and commands.

You will not be penalized if you choose to implement any of these extra features and commands.

Assumptions/Clarifications - All Subsets

Like all good programmers, you should make as few assumptions as possible.

You can assume that only the arguments described above are supplied to `slippy` commands. You do not have to handle other arguments.

You must apply the Slippy commands to input lines as you read the input lines. You can not read all input lines first (e.g. into a list).

There may be an unlimited number of input lines.

You are permitted to read one line ahead to handle `$` addresses.

You are permitted to read one line ahead even if the commands do not use a `$` address.

You should match the output streams used by the reference implementations. It writes error messages to stderr: so should you.

You should match the exit status used by the reference implementation. It exits with status 1 after an error: so should you.

You can assume arguments will be in the position and order shown in the usage message from the reference implementation. Other orders and positions will not be tested. Here is the usage message:

```
./slippy --help
usage: slippy [-i] [-n] [-f <script-file> | <sed-command>] [<files>...]
```

You can assume, Slippy regular expressions are valid Python regular expressions and are compatible with Python. In other words, they can be used as Python regular expressions and will have the same effect.

You can assume command line arguments, STDIN and all files contain only ASCII bytes.

You can assume all input lines in `STDIN` and in all files are terminated by a `'\n'` byte.

Slippy error messages include the program name. It is recommended you use `sys.argv[0]` however it is also acceptable to hard-code the program name. The automarking and style marking will accept both.

Testing

Autotests

As usual, some autotests will be available:

```
2041 autotest slippy slippy
...
```

You can also run only tests for a particular subset or an individual test:

```
2041 autotest slippy subset1 slippy
...
2041 autotest slippy subset1_13 slippy
...
```

If you are using extra Python files, include them on the autotest command line.

You can download the files used by autotest as a zip file or a tar file.

You will need to do most of the testing yourself.

Test Scripts

You should submit ten Shell scripts, named `test00.sh` to `test09.sh`, which run slippy commands that test an aspect of Slippy.

Your test script should check whether the test is passed or failed and print a suitable message.

Your test script should exit with status 0 if the test was passed and exit with status 1 if it was failed.

The `test??.sh` scripts do not have to be examples that your program implements successfully.

You may share your test examples with your friends, but the ones you submit must be your own creation.

The test scripts should show how you've thought about testing carefully.

You are only expected to write test scripts testing parts of Slippy you have attempted to implement. For example, if you have not attempted subset 2 you are not expected to write test scripts testing the change command.

Permitted Languages

Your programs must be written entirely in Python.

Start `slippy` with:

```
#!/usr/bin/env python3
```

Your answer must be Python only.
You can not use other languages such as Shell, Perl or C.

You may **not** run external programs, e.g. via the `subprocess` module or otherwise.
For example, you can't run `cat`, `head`, `tail`.

You may **not** use the builtin functions `eval` or `exec` functions.
You may **not** use (import) the `importlib` and `subprocess` modules.

You are permitted to use any builtin function, except `eval` or `exec`.
You may use (import) any standard library module, except `importlib` and `subprocess`.

You are not permitted to install Python modules with `pip` or similar software.

Your Python code should work with python3.9 on CSE servers.

You may submit extra Python files.

Change Log

Version 0.1
(2022-07-15 12:00)
- Initial release

Version 0.2
(2022-07-20 12:00)
- Rearranged text to make it more readable

Assessment

Testing

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
2041 autotest slippy
```

`2041 autotest` will not test everything.
Always do your own testing.

Automarking will be run by the lecturer after the submission deadline, using a superset of tests to those `autotest` runs for you.

Submission

When you are finished working on the assignment, you must submit your work by running `give`:

```
give cs2041 ass2_slippy slippy test??.sh [any-other-files]
```

You must run `give` before **Week 11 Monday 10:00:00 AM 2022** to obtain the marks for this assignment. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

You can run `give` multiple times.
Only your last submission will be marked.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

You *cannot* obtain marks by emailing your code to tutors or lecturers.

You can check your latest submission on CSE servers with:

```
2041 classrun check ass2_slippy
```

You can check the files you have submitted here.

Manual marking will be done by your tutor, who will mark for style and readability, as described in the **Assessment** section below. After your tutor has assessed your work, you can view your results here; The resulting mark will also be available via give's web interface.

Due Date

This assignment is due **Week 11 Monday 10:00:00 AM 2022**.

The UNSW standard late penalty for assessment is 5% per day for 5 days - this is implemented hourly for this assignment.

Each hour your assignment is submitted late reduces its mark by 0.2%.

For example, if an assignment worth 60% was submitted 10 hours late, it would be awarded 58.8%.

Beware - submissions more 5 days late will receive zero marks. This again is the UNSW standard assessment policy.

Assessment Scheme

This assignment will contribute 15 marks to your final COMP(2041|9044) mark

15% of the marks for assignment 2 will come from hand-marking. These marks will be awarded on the basis of clarity, commenting, elegance and style: in other words, you will be assessed on how easy it is for a human to read and understand your program.

5% of the marks for assignment 2 will be based on the test suite you submit.

80% of the marks for assignment 2 will come from the performance of your code on a large series of tests.

An indicative assessment scheme follows. The lecturer may vary the assessment scheme after inspecting the assignment submissions, but it is likely to be broadly similar to the following:

HD (85+)	All subsets working; code is beautiful; great test suite
DN (75+)	Subset 1 working; good clear code; good test suite
CR (65+)	Subset 0 working; good clear code; good test suite
PS (55+)	Subset 0 passing some tests; code is reasonably readable; reasonable test suite
PS (50+)	Good progress on assignment, but not passing autotests
0%	knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 FL for COMP(2041|9044)	submitting any other person's work; this includes joint work.
academic misconduct	submitting another person's work without their consent; paying another person to do work for you.

Intermediate Versions of Work

You are required to submit intermediate versions of your assignment.

Every time you work on the assignment and make some progress you should copy your work to your CSE account and submit it using the `give` command below. It is fine if intermediate versions do not compile or otherwise fail submission tests. Only the final submitted version of your assignment will be marked.

Attribution of Work

This is an individual assignment.

The work you submit must be entirely your own work, apart from any exceptions explicitly included in the assignment specification above. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted.

You are only permitted to request help with the assignment in the course forum, help sessions, or from the teaching staff (the lecturer(s) and tutors) of COMP(2041|9044).

Do not provide or show your assignment work to any other person (including by posting it on the forum), apart from the teaching staff of COMP(2041|9044). If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, you may be penalized, even if that work was submitted without your knowledge or consent; this may apply even if your work is submitted by a third party unknown to you. You will not be penalized if your work is taken without your consent or knowledge.

Do not place your assignment work in online repositories such as github or anywhere else that is publicly accessible. You may use a private repository.

Submissions that violate these conditions will be penalised. Penalties may include negative marks, automatic failure of the course, and possibly other academic discipline. We are also required to report acts of plagiarism or other student misconduct: if students involved hold scholarships, this may result in a loss of the scholarship. This may also result in the loss of a student visa.

Assignment submissions will be examined, both automatically and manually, for such submissions.