INFO1112 - Assignment 1 – mysh

Your task is to implement your own, Unix shell!

Ultimately, your shell should support the following features:

- a custom initialisation file, `.myshrc`, which is formatted as JSON.
- run and execute any program on the system's `PATH`
- define your own custom commands using JSON files
- support additional quality-of-life features to mimic other Unix shells

Without further ado, let's break down the main components on how the shell works!

Starting and Running the Shell

Starting the Shell

Starting the shell is simply done by running:

```
python3 mysh.py
```

The shell expects no additional arguments, and additional arguments are silently ignored.

Running the Shell

When the shell is running, you will see the prompt that has been set from the `PROMPT` environment variable, and the user will be able to type in commands to be executed:

```
>>
```

Effectively, your shell will *run infinitely* (in an infinite loop), until it is terminated, either by EOF (e.g. Ctrl + D), or if the `exit` built-in command is run (see here for more details on this). You can think of the shell as an "event-driven" program, where it is waiting for a command input (which will be the event), which it then reacts to.

While the prompt is being displayed, there are a couple of extra things to keep in mind:

- If the user enters Ctrl + C, but no command is running (i.e. the prompt is being displayed), Ctrl + C should be silently ignored, and a new prompt should be displayed.
- If the user inputs a blank command (an empty line, with or without whitespace), the shell should simply display a new prompt.

> **Note**
>
> If your terminal is terminated by EOF, make sure to print an extra newline character! (Thank you to the students who helped raise this to be compatible with Ed test cases, as this was initially overlooked!)

Syntax and Execution

Splitting a Line Into Arguments

Like other Unix shells, mysh is a line-based command parser. A user inputs one line at a time, and the line is executed as a command.

When we input one line of input, the line is split by whitespace, spaces, tabs, etc. (with an exception, more on this below), with each "word" forming an "argument". For example, the line:

```
hello there      how  are you
```

can be split into a list of arguments (using Python syntax) as:

```
["hello", "there", "how", "are", "you"]
```

There is one exception to this: quoted strings (both single and double quotes)! The outer level of quotes are stripped from the quoted string. If a string is inside some quotes, and it includes whitespace, the whitespace is preserved.

For example, the below line:

```
hello  there "how are you" 'going   today'
```

will be split as:

```
["hello", "there", "how are you", "going   today"]
```

while the below line:

```
hello  there  'my   name is  "Sarah"'
```

will be split as:

```
["hello", "there", 'my   name is  "Sarah"']
```

Quotes can also be *escaped* by typing a backslash (\) before the quote character. This means that if you have quoted argument of single or double quotes, and type in the same type of quote character prepended with a backslash (e.g. `\"` or `\'`), it won't be interpreted as a closing quote, but instead, as a quote character – similar to the behaviour of Python.

For example, the below line:

```
my "\"string\"" which 'is \'quoted\''
```

will be split as:

```
["my", '"string"', "which", "is 'quoted'"]
```

> **Tip**
>
> The `shlex` module will be incredibly useful to help parse these strings, and will meet the edge cases below! Please use it!

Edge cases

- It is valid to have double quotes inside single quotes (as in the above example), and single quotes inside double quotes.
- It is valid for quotes to be spliced inside of arguments, and can even be next to each other, as long as the outer layer of quotes is stripped out! For example, the line below:

  ```
  hello there "quotes are""next to 'each'"-other
  ```

  should be split as:

  ```
  ["hello", "there", "quotes arenext to 'each'-other"]
  ```

Error cases

- An unterminated quoted argument should be considered a **syntax error** by your program and should output an error message to `stderr`, formatted below. For example, for the line:

  ```
  line  with "unterminated string
  ```

  the program should output:

  ```
  mysh: syntax error: unterminated quote
  ```

  to `stderr`.
  - Note that unterminated quote characters *are valid* if they are inside quotes of a different type. For example, the line:

    ```
    this  is   "Tom's PC"
    ```

can be validly split as:

```
["this", "is", "Tom's PC"]
```

Executing Commands

When a line is split, the first "word" is always defined as the name of the command to execute. For example, for the line:

```
  sort     input.txt  -o output.txt
```

will be parsed as:

```
["sort", "input.txt", "-o", "output.txt"]
```

with `"sort"` being the name of the command (or program) we wish to execute, and `["input.txt", "-o", "output.txt"]` being the arguments passed to the `"sort"` program.

The command name will be attempted to be matched to a built-in command (see Built-in Commands for all built-in commands), otherwise, it will execute an executable on the system with this name, either by checking in the system's `PATH` for a matching executable name, or by executing the executable given by an absolute or relative path. For instance:

```
./my_compiled_prog
```

would execute an executable `my_compiled_prog` in the current directory of the user.

If a valid command is successfully entered, the shell will then execute the command, and wait for it to complete, before displaying the prompt again, and asking the user to input another command.

Tip

The info you learn about `fork` and `exec` in Week 3 will be very useful for helping to implement the ability to run programs in your shell! You can also find implementations of these system calls as functions in the Python `os` module.

Important

If a command is run in a separate process, it is important to set the **process group** of the child process to a brand new one, and set the new **foreground terminal process group** to the new group the child process is in. This is so functionality, such as pressing Ctrl + C, only affects the child, rather than the parent process.

In short, in the child process, the only other thing you'll need to do after forking is to create a new process group by calling `os.setpgid(0, 0)` (before doing whatever you need to do to `exec`).

In the parent process, there are a couple of steps that need to be followed after forking:

1. First, also try to create a new process group for the child process, by adding:

   ```
   try:
       os.setpgid(child_pid, child_pid)
   except PermissionError:
        # Child has already set new process group!
        pass
   ```

   Why do we need both? It's a little tricky at first, and there's no need to fully understand it, but essentially: since each process effectively runs immediately after forking, if the parent would set the process group (call `os.setpgid`) *after* the child has called an `exec` function, it would fail and cause a `PermissionError` to be thrown. Setting the process group in both the parent and child processes prevents this *race condition* from impacting our program's functionality (Thank you to the student who identified this in Ed thread #159!).
2. Get the child's new process group ID with `os.getpgid(child_pid)`.
3. Open the **current terminal device**, located in `/dev/tty` to get its file descriptor.
4. Set the terminal foreground process group with `os.tcsetpgrp(</dev/tty open file descriptor>, child_pgid)`.
5. Wait for the child process to complete.
6. Restore the terminal foreground process group back to the parent's process group with `os.tcsetpgrp(</dev/tty open file descriptor>, parent_pgid)`.
7. Make sure the `/dev/tty` file descriptor is closed as well!

Useful functions:

- `os.setpgid`
- `os.getpgid`
- `os.tcsetpgrp`
- `os.getpgrp`

Error cases

- If the first argument does **not** contain a slash and the argument does not refer to a built-in command or executable on the `PATH`, the shell should output:

  ```
  mysh: command not found: <command name>
  ```

  to `stderr`.
- If the first argument **contains** a slash anywhere within it (indicating it is a relative or absolute path), and it does not refer to a valid file or directory, the shell should output:

  ```
  mysh: no such file or directory: <argument>
  ```

  to `stderr`.

- If the first argument **contains** a slash and *is* a valid path, but cannot be executed due to it being a directory, the shell should output:

  ```
  mysh: is a directory: <documents>
  ```

  to `stderr`.
- If the first argument **contains** a slash and *is* a valid path, but the user cannot execute the file due to lacking appropriate executable permissions, the shell should output:

  ```
  mysh: permission denied: <argument>
  ```

  to `stderr`.

Pipes

Like other Unix shells, one of the features that mysh will support is piping the `stdout` result of one command as `stdin` input for another command!

In POSIX shells, such as Bash, commands are piped from one to another using the pipe operator (`|`). For instance, in the line below:

```
echo "Hello!" | cat
```

both `echo` and `cat` are immediately started by the shell. `echo` prints `Hello!` to what normally would be `stdout`, however, instead, the shell *captures* this output. Meanwhile, `cat` is also waiting for input to come from what normally would be `stdin`, however, the shell will instead feed it the output of `echo`. The result is that after `echo` prints `Hello!`, the shell captures this output, and passes it along to `cat` as if it was from `stdin`, which it then reads, and thus prints `Hello!` to the terminal screen.

In mysh, pipes operate in the same way. The syntax for pipes is below:

```
<command 1 with arguments> | <command 2 with arguments> | ... | <command n with
arguments>
```

Each pipe operator (`|`) is placed between 2 commands with arguments, with the `stdout` output of the previous command being captured and being fed as `stdin` to the next command. All commands as part of a sequence of pipes (also known as a **pipeline**) start simultaneously, and each command waits for input to read (as if it was from `stdin`) from the previous command.

For example, in the line:

```
a b c | d e|f
```

the command `a` is run with arguments `b c`, where its `stdout` output is passed to `stdin` to command `d` (run with arguments `e`), and in turn, `d`'s `stdout` output is passed as `stdin` to command `f`.

However, if the pipe operator is in single or double quotes, it is not interpreted as a shell pipe, but rather as a literal '|' character instead.

For instance, in the line:

```
a | b 'c | d'
```

we interpret this as passing the `stdout` of command `a` to the `stdin` of command `b`, which was invoked with the single argument `c | d` (just a string).

> **Tip**
>
> The scaffold for the assignment contains a module, `parsing`, which contains a function, `split_by_pipe_op` which you can use to split strings by an unquoted pipe operator!

> **Tip**
>
> The `os.pipe`, as well as the `os.dup2` / `os.dup` functions will be incredibly useful here!

> **Important**
>
> Each process which is executed as part of a **pipeline** must be part of the same **process group** while executing, with this process group being set as the **foreground terminal process group**, so that if the user wants to terminate execution of the pipeline early (e.g. by pressing Ctrl + C), all processes in the pipeline will terminate. See the Executing commands section for more details about setting this.

Error cases

- If a line contains a trailing pipe operator without a command after it, for example:

  ```
  a | b |
  ```

  the shell should output:

  ```
  mysh: syntax error: expected command after pipe
  ```

  to `stderr`.
- If a line contains multiple pipes, but there is no command between 2 pipes, for example:

  ```
  a | b | | c
  ```

  this should output the same `syntax error` to `stderr` as above.

Using Shell Variables

Just like other Unix shells, mysh supports defining your own variables, which are saved into the process's environment variables, and can be used as part of the shell prompt.

Defining your own variables can be done via the `var` command, and usage of this command can be found its corresponding section, so let's talk about how we can use defined shell variables first.

Shell variables may be used as part of some command line input with the `${<variable name>}` syntax. If an argument contains this syntax, it should be replaced with the value of the environment variable matching `${<variable name>}`.

For example, if we have an environment variable `greeting` with the value `Hello, world!`, and the user enters the line:

```
echo ${greeting}
```

`${greeting}` should be substituted with its value, with the line being interpreted as:

```
echo Hello, world!
```

However, the user may choose to "escape" this syntax with a backslash ('\'), as they may not want to have this interpreted as a string. To "escape" the variable usage syntax, the user writes a backslash before the `$` symbol, i.e. `\${greeting}`. This means that for the line:

```
echo \${greeting}
```

the line should instead be interpreted as:

```
echo ${greeting}
```

i.e. calling the `echo` command, with the literal string `${greeting}` as a single argument.

If the variable is used inside a quoted argument (either single or double quotes), substitution should occur as normal. For example, if the `name` environment variable had the value `Rosanna`, the line:

```
echo "My name is ${name}"
```

would be interpreted as:

```
echo "My name is Rosanna"
```

The same goes for if an argument is inside single quotes single quotes - for instance, the line:

```
echo 'My name is ${name}'
```

would be interpreted as:

```
echo 'My name is Rosanna'
```

Error cases

- If `<variable name>` contains characters which are invalid for a variable name (see `var`), a syntax error should be printed to `stderr` in the form:

  ```
  mysh: syntax error: invalid characters for variable <variable name>
  ```

  The command which contained the invalid variable name should also not be executed.

  If there are multiple variables with invalid characters, mysh will only raise an error on the first invalid variable.
- If there is no environment variable named `<variable name>`, the string `${<variable name>}` will be substituted (if applicable) by an empty string. For instance, if no variable `name` exists, the following line:

  ```
  echo "My name is ${name}"
  ```

  will be parsed as:

  ```
  echo "My name is "
  ```

Built-in Commands

As part of mysh, some commands are defined as "built-in" commands, that is to say, that the commands do not necessarily run a separate executable, but instead, is executed as part of some built-in functionality for the shell.

`var`

The `var` built-in command is used to set the value of environment variables within mysh. The syntax for defining this command is:

```
var [-s] <variable name> <argument>
```

where `<variable name>` is the name of the shell variable, which can be:

- any alphabetical letters A-Z, capital or lowercase
- any digits 0-9
- underscores (_)

and `<argument>` represent the value of the shell variable.

The `-s` flag is optional – see the Setting the result of a command subsection for more details. In this case, more than 1 argument may be provided.

> **Important**
>
> All shell variables are interpreted as strings, as these **must** be stored in the process's environment, which only supports string environment variables. See `os.environ` for more info.

In basic usage, if the user were to input:

```
var count 0
```

the shell would create a new environment variable `count`, with its value set to `0`.

If the user were to input:

```
var greeting "Hello, world!"
```

the shell would create a new environment variable `greeting`, with its value set to `Hello, world!`.

If `var` is successfully called, there should be no additional output created.

Setting the result of a command

The `var` command also accepts one possible optional flag – `-s`. This flag allows you to set a variable to a value that is the `stdout` result of a command, rather than a string value.

If the `-s` flag is specified, the syntax for `var` will look like the below:

```
var -s <variable name> <command argument>
```

`<command argument>` is a single argument which represents the execution of a command as if it was to be executed by the shell. This may be just a command name (e.g. `ls`), a command with arguments (e.g. `"ls -l ~"`), or a pipeline (e.g. `'ls -1 ~ | wc -l'`) – the latter 2 options being entered as a quoted argument to be passed as a single argument.

For example, if the user was to input:

```
var -s add_res "expr 2 - 5"
```

the shell should store the result of executing the command `expr 2 - 5` into the environment variable `add_res`.

Similarly, if the user was to input:

```
var -s usyd_info_units 'sort usyd-scs-units.csv | grep "INFO"'
```

the shell should store the result of executing `sort usyd-scs-units | grep "INFO"` into the environment variable `usyd_info_units`.

Edge cases

- Flag arguments (`-s`, as well as any invalid flags) are only interpreted as flags if they are the first argument passed to `var`. If they are in any other position, they are interpreted as a regular argument to `var` (following the syntax requirements above).

Error cases

- If the user enters any flag arguments which are not `-s`, the shell should output the following to `stderr`:

  ```
  var: invalid option: -<option>
  ```

  For instance, if the user enters `var -t test_var test_value`, the shell should output:

  ```
  var: invalid option: -t
  ```

  If multiple options are specified, such as `var -tu`, the first invalid option found should be outputted.
- If the user does **not** enter 2 arguments for `var` (which does not include optional arguments, like `-s`), the shell should output the following to `stderr`:

  ```
  var: expected 2 arguments, got <count>
  ```

  This error takes priority over invalid options being entered (so if the user enters an invalid option **and** more/less than 2 arguments, this is the error message which should be outputted).
- If the `<variable name>` the user enters does NOT contain valid characters (ones other than those specified), the shell should print an error to `stderr`, saying:

  ```
  var: invalid characters for variable <variable name>
  ```

`pwd`

Like the `pwd` command found in most Unix shells, the `pwd` built-in command should print the current working directory as an absolute path that the shell is located in to `stdout`.

The user may optionally pass one flag: `-P`, which, if passed, will resolve additionally resolve all symbolic links in the shell's current working directory to their real path before printing this as an absolute path to `stdout`.

Example usage

If the user was in their home directory (e.g. `/home/bob`), the following:

```
pwd
```

should output

```
/home/bob
```

If the user was in the directory `/home/bob/current-year`, where `/home/bob/current-year` was a symbolic link to `/home/bob/2024`, the following:

```
pwd
```

should output

```
/home/bob/current-year
```

and the following

```
pwd -P
```

should output

```
/home/bob/2024
```

> **Tip**
>
> To help ensure your program outputs the correct absolute path if there are symbolic links within the path (i.e. if `pwd` with no arguments is called), you should keep track of your program's current working directory (as an absolute path) independently somewhere, and update this accordingly with commands such as `cd`.

Error cases

- If the user enters any additional non-flag arguments (e.g. arguments which do not start with a `-`), the shell should output the following to `stderr`:

  ```
  pwd: not expecting any arguments
  ```

- If the user enters any flag arguments which are not `-P`, the shell should output the following to `stderr`:

  ```
  pwd: invalid option: -<option>
  ```

  For instance, if the user enters `pwd -f`, the shell should output:

```
pwd: invalid option: -f
```

If multiple options are specified, such as with `pwd -f -p` or `pwd -fp`, the first invalid option found should be outputted.

`cd`

The `cd` command will change the current working directory to the path specified by a single argument, **and** modify the contents of the environment variable `PWD` to reflect the current absolute path after the directory has been changed (i.e. what the output of running `pwd` by itself would be).

The syntax to execute it is:

```
cd <path>
```

`<path>` may either be an absolute or relative path.

> **Important**
>
> - **If the path is an relative path**: it is important to first update the user's working directory by ***normalising*** the updated path `${PWD}/<path>` to factor in any special constructs such as `.` and `..` in the path, whilst keeping other elements of the path, including symbolic links, intact.
>   - This is especially important to help ensure you output the correct path for `pwd` (without the `-P` flag) by keeping track of the current working directory somewhere in your program – independently of `os.getcwd`, as this does not take into account symbolic links!
>   - You will likely find something helpful in the `os.path` module to assist you with normalisation!
> - **If the path is an absolute path**: there shouldn't be any need for normalisation, so you should make sure your `cd` command can differentiate if `<path>` is an absolute or relative path.

There is a special case where the used can execute `cd` with **no** arguments. In this case, the shell should change the current working directory (and update `PWD` accordingly) to the **home** directory of the user.

Error cases

- If the user inputs more than one argument to `cd`, the shell should output the following error to `stderr`:

  ```
  cd: too many arguments
  ```

- If the the `<path>` provided as an argument is invalid (does not exist), the shell should output the following error to `stderr`:

  ```
  cd: no such file or directory: <path>
  ```

- If the `<path>` provided is not a directory, the shell should output the following error to `stderr`:

```
cd: not a directory: <path>
```

- If the user does not have the correct permissions to change directory to `<path>`, the shell should output the following error to `stderr`:

```
cd: permission denied: <path>
```

`which`

The `which` command identifies whether a command refers to a built-in command in the shell, or otherwise provides the absolute path of the first executable matching the command's name in the system's `PATH` variable.

> **Important**
>
> If the `PATH` environment variable is unset for any reason, `which` falls back to the path set in `os.defpath`.

The syntax defining it is:

```
which <commands...>
```

where `<commands...>` is a list of one or more arguments.

If an argument is a built-in command, `which` should output (to `stdout`):

```
<command name>: shell built-in command
```

Otherwise, if the command is on the system `PATH`, **and** it **is a file** which has **executable permissions**, `which` will output the first matching absolute path to this command. For example, `which echo` might output (to `stdout`):

```
/bin/echo
```

If no matching built-in command or executable is found for the argument, `which` should output the following to `stdout`:

```
<command name> not found
```

If `<commands...>` consists of multiple arguments, each will be evaluated in order of which they were provided using the above rules, and have its result printed to `stdout` on separate lines. For instance, `which cd ls foo-bar` might output:

```
cd: shell built-in command
/bin/ls
foo-bar not found
```

> ⚠ **Warning**
>
> Since `which` is a built-in command in our shell, your program should **not** invoke the system-installed `which` command (e.g. `/usr/bin/which`) when it is executed! This will be double-checked with automatic test cases and manual marking.

Error cases

- If no arguments are entered, the shell should output the following to `stderr`:

  ```
  usage: which command ...
  ```

`exit`

A nice, simple built-in command. When executed, the shell exits.

Its syntax is:

```
exit [<exit code>]
```

Exit optionally takes one additional additional argument, `<exit code>`, which is the integer exit code the shell should exit with.

Error cases

- If there is more than 1 argument provided to `exit`, the shell should output the following to `stderr`:

  ```
  exit: too many arguments
  ```

  and **not** terminate the shell.
- If a non-integer argument is passed to `exit`, the shell should output the following to `stderr`:

  ```
  exit: non-integer exit code provided: <argument>
  ```

  and **not** terminate the shell.

Configuration and Initialisation

Default Environment Variables

Many Unix shells will use environment variables to modify how they operate. For instance, one common environment variable used by shells is `PS1`, which dictates the shell *prompt*.

> **Tip**
>
> You can test the above in your shell! Try running `PS1='My Prompt >> '`, and watch your prompt change in real time!

These environment variables are often inherited by a child process from its parent when its starts, and are set per process, and all of the process's children.

Just like other Unix shells, mysh also has some environment variables which it uses to affect how its process runs. If these are undefined when mysh is launched, they are set to the default values defined below:

> **Important**
>
> These should be set in the actual process environment when your program is launched. Refer to `os.environ` for more details.

- `PROMPT`: `>>`
- `MYSH_VERSION`: `1.0`

Initialisation File

Unix shells often use an *initialisation file* which can be used to customise some environment variables that change how the shell, or other programs, run. The name of this initialisation file, and the syntax it uses depends on the shell being used – for example, Bash will use `.bashrc`, and Zsh `.zshrc`.

Just like other Unix shells, our shell, mysh, also has an initialisation file – called `.myshrc`, structured as a JSON object. By default, this file lives in the home directory of the user – `~/.myshrc`, unless the `MYSHDOTDIR` environment variable is set prior to mysh being started, which will cause mysh to look for the file in `${MYSHDOTDIR}/.myshrc`. The user can use this file to set or override any previously-set **environment variables** for the current process, with the key of the JSON object being the environment variable's name, and the value being the environment variable's value.

Each environment variable's corresponding value may also reference other environment variables, using the same syntax defined in the Using Shell Variables section. Variables are initialised from top to bottom in the file, meaning that if a variable in `.myshrc` references a variable which is defined later in the file, it should be interpreted as if the variable did not exist.

> **⚠ Warning**
>
> Remember that strings are the only valid data type for environment variable values.

Below is a sample of a `.myshrc` file:

```
{
  "PROMPT": "${USER}'s cool shell! > ",
  "UNI_FILES": "~/Documents/Uni",
  "CURRENT_SEM_FILES": "${UNI_FILES}/2024/S2"
}
```

> **Tip**
>
> The `json` Python standard library module will be incredibly helpful to help parse these files!

> **ⓘ Important**
>
> If `.myshrc` cannot be found, no error should be reported – the shell should continue as normal.

Error cases

- An invalidly-formatted `.myshrc` file (that cannot be parsed by the `json` module), should result in the following error message being printed to `stderr` upon startup:

  ```
  mysh: invalid JSON format for .myshrc
  ```

  and all environment variable values in `.myshrc` ignored.
- For *every* environment variable value in `.myshrc` which contains invalid characters (refer to `var`), the following message should be printed to `stderr`:

  ```
  mysh: .myshrc: <variable name>: invalid characters for variable name
  ```

  and its value not set in the process's environment variables. Other valid environment variables, however, should still be set.
- For *every* environment variable value in `.myshrc` which is not a string, the following message should be printed to `stderr`:

  ```
  mysh: .myshrc: <variable name>: not a string
  ```

  and its value not set in the process's environment variables. Other valid environment variables, however, should still be set.

Testing

As part of your submission, you are expected to create your own test cases which you should use to test your program with continually while you are in development. All files related to testing should reside in the `tests/` subfolder of your submission. **Files/folders used for testing which are outside of `tests/` may not be considered for testing marks**.

The type of tests that should be created are **input/output (I/O) tests**, or **end-to-end tests**, which are designed to test the entire execution of your program given a set of inputs passed as `stdin` to your program, and comparing your program's `stdout` / `stderr` to some expected file. Generally, this is done by having a corresponding `.in` file containing only the *exact* lines of input you would enter into your program, and a `.expected` or `.out` file containing the expected output of your program. You don't strictly have to follow this exact structure, however - the most important thing is that end-to-end testing is performed.

You should also submit a testing script, `tests/run_tests.sh`, which will run all tests developed for your program automatically, **without user intervention**. The testing script should output whether your program passes/fails one of your corresponding test cases. As part of marking your testing submission, staff will run `tests/run_tests.sh`.

README/Report

As part of your submission, you are required to create a `README.md` file that acts as a report for your program that is 750 words or less in length. Your README file is required to answer the following questions, containing references source code lines or functions as appropriate to support your answers:

- How does your shell translate a line of input that a user enters into a command which is executed in the command line?
- What is the logic that your shell performs to find and substitute environment variables in user input? How does it handle the user escaping shell variables with a backslash (\) so that they are interpreted as literal strings?
- How does your shell handle pipelines as part of its execution? What logic in your program allows one command to read another command's `stdout` output as `stdin`?

Your README file should also contain a brief description on the categories/types of tests you've created, and how your test files are structured in the `tests/` directory.

Allowed Modules

The following Python standard library modules are permitted to be used in your submission:

- `abc`
- `collections.abc`
- `dataclasses`
- `enum`
- `json`
- `os`

- `os.path`
- `pathlib`
- `re`
- `readline` (**note**: not assessable, but useful for optional quality-of-life features!)
- `shlex`
- `signal`
- `sys`
- `typing`

Any other modules which are *not* on this list **may not be used in your assignment**, and **will result in a mark deduction (20% of A1 max grade, or 4 marks) if they are used**.

This list, however, may be expanded in future. If you would like to use a certain Python module, please ask on Ed, and staff will be happy to review your request, and if appropriate, update the list of allowed modules. A pinned post will be created which will track any updates to the list of allowed modules.

Restricted Functions

In addition, some functions are **restricted**, as they break the spirit of the assignment of making your own shell. Using these functions **will also result in a mark deduction of 40% of A1 max grade (8 marks)**.

- `os.system` – as this just passes a command string to the shell - we want to implement the shell ourselves!
- `os.popen` – similar to above.
- Any other function which spawns a sub-shell to execute a command. If you are unsure, **please search/ask** on Ed.
- `__import__` function to import a module – please use the standard `import` to import modules and/or functions.

Whilst falling under restricted modules, if the following modules are used, these will be **subject to the 40% penalty mentioned above**, as they also **break the spirit of the assignment**:

- `subprocess`
- `multiprocess`
- `cmd`
- any other module which interacts with processes/sub-shells to execute a program/command. If you are unsure, **please search/ask** on Ed.

Marking Breakdown

The mark breakdown for this assessment is as follows:

- Automated test cases (13 marks) - Marks from automated test cases from Ed, which include public, hidden, and private test cases.
  - Different test cases will have different weightings (points) attached to them, visible on Ed.
  - Private test cases may be run after the due date.
- Own test cases (2 marks)

- Code style (2 marks) - broken down:
  - Automated code style checking with Pylint (1 marks) - feel free to run `pylint` on your code to check your code style yourself
    - The marks for this will follow the following scheme:
      - $\text{score} \geq 7.5/10$ = 1/1 marks
      - $5.5/10 \leq \text{score} < 7.5/10$ = 0.75/1 marks
      - $2.5/10 \leq \text{score} < 5.5/10$ = 0.5/1 marks
      - $1/10 \leq \text{score} < 2.5/10$ = 0.25/1 marks
      - $\text{score} < 1$ = 0/1 marks
  - Manual code style checking (1 marks)
- README/Report (3 marks)

Friendly Note and Important Dates

Sometimes we find typos or other errors in specifications. Sometimes the specification could be clearer. Students and tutors often make great suggestions for improving the specification. Therefore, this assignment specification may be clarified up to **Week 4, Tuesday, 20th August**. No major changes will be made. Revised versions will be clearly marked, and the changelog updated both on the assignment specs, and made visible by a pinned post on Ed.

You are encouraged to apply for special consideration and arrangements if any difficulty comes up which may impact your submission for the assignment on time.

Please make sure to also reach out to staff for any questions on the specs!

A friendly reminder that per the unit outline, late submissions are **not accepted** and **result in a mark of 0** being awarded for the assignment.

Warning

Any attempts to deceive the automatic marking system will result in an immediate zero for the entire assignment.

Negative marks can be assigned if your code is unnecessarily or deliberately obfuscated.

All documentation in your submission, including comments, README files, and so on, must be written in English. If not, penalties will apply. **A deduction of 1 mark per line of non-English code will apply, up to a maximum deduction of 5/20 marks for the assignment**.

Academic Declaration

By submitting this assignment, you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.

Changelog

Changes/clarifications to the specs along with their corresponding dates will be made here.

- v1.3 – 19/08/24
  - Issue clarification that if `.myshrc` cannot be found, no error should be thrown (instead, silently ignore this)
  - Add clarification for `var` error message that should be displayed if < 2 arguments are passed to it (**note**: options (such as `-s`) do not count as arguments in this case)
  - Added `enum` as an allowed module to import
- v1.2 – 13/08/24
  - Added clarification about normalising path for the `cd` command (based on Ed thread #137)
  - Added important tip about ensuring your program keeps track of the valid current working directory independently in your program (in the `pwd` section), to ensure that your program knows its "true" path on the system, taking into account factors such as symbolic links (based on Ed thread #137)
  - Added extra information and tips for the Executing Commands section to ensure that your shell doesn't crash in an extreme edge case where the child `exec`s before the parent calls `os.setpgid` (race condition) (based on Ed thread #159)
- v1.1.1 – 09/08/24
  - Fixed typo in `pwd` error condition for invalid flags: the condition should have been "arguments which do not start with a `-`" rather than "arguments which do not start with a `-P`"
- v1.1 – 08/08/24
  - Clarify that after EOF is encountered when reading input, the shell should print an extra newline (Thank you to the students which found this!)
  - Clarified that an empty command in-between 2 pipes should also output a `syntax error` to `stderr`
  - Add `readline` as a (non-assessable) allowed module to import
  - Clarify Pylint marking scheme