Assignment 2 - Simple Nautilus*

This assignment is worth **20%** of your overall grade for the course.

Due: Week 8, Thursday, September 22nd at 23:59:00, Sydney local time.

You are going to be writing an interactive application in Python that allows the user to send commands to a file management application and receive outputs. The application largely mocks common Unix commands such as `ls` , `cd` , `pwd` , and so on.

You will need to implement and thoroughly test your solution.

On the high level, the application is capable of:

1. interpret commands received on standard input.
2. produce messages on standard output.
3. maintain some data structure that keeps track of a ***virtual*** name space.
4. create, delete and move ***virtual*** files and folders during program run time.
5. support user and permission management in addition to file management.

In Linux system, you can interact with the Terminal prompt to navigate in the file system using `cd` , to perform actions to create, move, delete and copy files using `touch` `mv` `rm` `cp` , to alter the ownership and permission of files by `chown` `chmod` and to add or remove users by `adduser` `deluser` .

In this assignment, you need to write a Python program which provides an interactive prompt as if a Linux Terminal. Human users can interact with the Python program via standard input and standard output. One can input commands, and as a result, the program may print out feedback messages due to certain input.

Note that the system is incapable of storing content but purely the file system structure. This is an intensional design and therefore the size of a file is not defined and is omitted in the system. Similarly, the concept of time and `group` is omitted intentional. Please see the whole specification for more details.

The assignment has three tasks that you will need to implement and thoroughly test. You will be provided a test suite to assist in developing your solutions.

The three tasks are:

1. Implement a minimal Simple Nautilus.
2. Implement a simplified Simple Nautilus.
3. Implement a full Simple Nautilus.

Please read the whole specification to better understand these tasks.

* Nautilus: a file manager for GNOME.

Formatting Syntax

It is very common that command strings are written in a simple template language: characters are usually interpreted literally into themselves, but format specifiers, which are enclosed by parentheses, indicate the location and method to translate a piece of placeholder to characters.

In Simple Nautilus, the following formatting is used:

- Formatted arguments enclosed in sharp brackets `<` and `>` can be replaced by any single syntactically valid string.
- Formatted arguments enclosed in square brackets `[` and `]` can be replaced by zero or one syntactically valid string.
- Parameters that are not further specified can be any string. In order to be considered as a single syntactically valid parameter, a string that contains whitespace has to be enclosed by double quotation marks.
- Syntactically valid does not suggest semantically valid. E.g. `"/a/b/c/d"` is a syntactically valid string, but it might be semantically invalid given an environment where `"/a/b/c/d"` does not exist and the user tries to `ls /a/b/c/d` .

Example

A formatted string

```
<a>@<b>:<c>,[d]
```

can be extended by

```
Never@" Gonna ":"Give You","up"
```

`<a>` , `<b>` , `<c>` and `[d]` are extended to `Never` , `" Gonna "` , `"Give You"` and `"up"` .

or

```
Among@Us:yo,
```

`<a>` , `<b>` and `<c>` are extended to `Among` , `Us` , `yo` and `[d]` is omitted.

Permission and Ownership

Like in Unix, this system supports user and permission management. However, the concept of `group` is omitted and therefore the mode string for permission is shorted than its form in Unix.

To view the permissions for all files in a directory, use the `ls` command with the `-l` option.

For example, if you enter:

```
ls -l
```

You should see output similar to the following:

```
-rw-r-- user myfile.txt
drwxr-x root Example
```

The first column indicates the file type and the permission information

The second column indicates the owner of the file or directory.

The third column indicates the name of the file or directory.

The first character in the first column indicates whether the listed object is a file or a directory. Directories are indicated by a (`d`); files are indicated by a dash (`-`), which is the absence of a `d` at the beginning of the first line. Then, there are two sets of three characters that represent different levels of ownership. Hence, `myfile.txt` is a file and `Example` is a directory.

The letters `rwx` represent different permission levels:

Permission	Files	Directories
`r`	can read the file	can `ls` the directory
`w`	can write the file	can modify the directory's contents
`x`	can execute the file	can `cd` to the directory

For example,

- `drwxr-x` : `d` being the first character indicates directory; the owner permissions are `rwx` , indicating that the owner can view, modify, and enter the directory; `other` (anyone other than the owner) can view as well as enter the directory. (Group is not defined, therefore there are no group permissions.)
- `-rw-r--` : `-` being the first character indicates file; the owner permissions are `rw-` , indicating that the owner can read and write to the file but can't execute it as a program; `other` (anyone other than the owner) can only read the file. (Again, group is not defined, therefore there are no group permissions.)

When the file or directory is being created, the current effective user will determine the owner of it.

Before the file or directory is being created, the current effective user can be updated by `su` . This indicates the owner of file or directory to be created can be different from time to time, depending on who the current effective user is.

Furthermore, the ownership of the file or directory can be directly updated by `chown` . This indicates the ownership does not remain unchanged indefinitely after creation.

Default setting and lifetime

The default user upon starting the program is `root` with the initial working directory being the root directory (`/`).

Whenever the program starts, the initial effective user `root` and an empty virtual name space are created; whenever the program exits, the whole virtual user space and virtual name space are lost.

`root` user is the most powerful user in this system. When the current effective user is `root` , they can ignore all permission requirements stated below. In other words, `root` will face no `Permission denied` .

Files are created with `-rw-r--` permission on default.

Directories are created with `drwxr-x` permission on default.

Prompt Formation

The input prompt should have the following formation, where the leading prompt is ended by a dollar symbol and a single white space:

```
<user>:<pwd>$ <command>
```

White spaces that come after a valid command should be ignored.

If there are only whitespace characters being inputted by the user as a commond string, nothing will happen when the user executes this command.

Example

If user `root` is at path `/` , the prompt printed before any input should be:

```
root:/$
```

If `bob` enter a command `pwd` , then the combination of the input and prompt forms the line:

```
bob:/$ pwd
```

Valid character set

User names, file names and directory names are valid when it only contains:

- All alphanumeric characters, `a` to `z` , `A` to `Z` , and `0` to `9` ;
- the Space character, ` `

  - An valid argument that contains Space must be surrounded by `"`
- the Hyphen character, `-` ;
- the Dot character, `.` ;
  - However itself alone `.` (dot) or `..` (dot-dot) has special meaning, see below.
- the Underscore character, `_` .

Special path

dot

A dot component in a path refer to the preceding component.

For a path `a/b/c/./d/.` , the preceding component of the first (left-most) dot is `a/b/c` . The preceding component of the second (right-most) dot is `a/b/c/./d` , which shall evaluate to `a/b/c/d` . The whole path shall evaluate to `a/b/c/d` .

dot-dot

A dot-dot component in a path refer to the parent of the preceding component. That is, it moved up one level toward the root `/` .

For a path `a/b/c/../d/..` , the preceding component of the first (left-most) dot-dot is `a/b/c` . The preceding component of the second (right-most) dot-dot is `a/b/c/../d` , which shall evaluate to `a/b/d` . The whole path shall evaluate to `a/b` if every prefix of the path is valid. Otherwise, the execution of the command is aborted and an error should be triggered and an error message should be printed as per command, as different command prints different error message upon non-exist error.

If the preceding component is empty, e.g. `../a/b/c` , assume the preceding component to the current working directory. If the preceding component already refers to the root `/` , e.g. `/..` , then dot-dot component will also refer to root `/` (not the parent of `/` nor an error).

Example

Assume there exists a tree of directories `/a/b/c/d/e` , where `a` to `e` are valid directories. There are no other files or directories on the system.

```
root:/$ cd a/b/c/..
root:/a/b$ cd ..
root:/a$ cd ..
root:/$ cd ..
root:/$ cd /a/b/c/d/e
root:/a/b/c/d/e$ cd ../e/../../d/../../..
```

```
root:/a$ cd ././././/b/././/c/./d
root:/a/b/c/d$ cd ./.././/.././/
root:/a/b$ cd .././/.//../.
root:/$ cd a/b/e/..
cd: No such file or directory
root:/$ cd a
root:/a$ cd b/c/e
cd: No such file or directory
root:/a$ cd b/c/e/..
cd: No such file or directory
```

Commands

Please note that all format string to be printed should be expended by the real parameter in-place.

`exit`

- To quit the program.
- Print: `bye, <current_user>`

`pwd`

- Print name of current working virtual directory.

`cd <dir>`

- Change the working directory to `<dir>` .
- Require current effective user's execute bit `x` on `<dir>` .
- If `<dir>` does not exist, print: `cd: No such file or directory`
- If `<dir>` refers to a file, print: `cd: Destination is a file`

`mkdir [-p] <dir>`

- Create the directory `<dir>` , if `<dir>` does not already refer to a file or directory.
- Require current effective user's execute bit `x` on `<dir>` 's ancestors.
- Require current effective user's write bit `w` on `<dir>` 's parent.
- If `-p` is not specified and any ancestor directory in `<dir>` does not exist, print: `mkdir: Ancestor directory does not exist`
- If `-p` is not specified and `<dir>` does exist, print: `mkdir: File exists`
- If `-p` is specified, do not print error and do nothing if `<dir>` cannot be created by just making ancestor directories as needed.

- If `-p` is specified, do not print error if any ancestor directory in `<dir>` exists, and make ancestor directories as needed.
- Directories are created with `drwxr-x` permission (with regards to the current effective user).

`touch <file>`

- Update the access and modification times of a `<file>` to the current time. However, because the concept of time is omitted in this assignment, this command is mainly used to create new files.
- Require current effective user's execute bit `x` on `<file>` 's ancestors.
- Require current effective user's write bit `w` on `<file>` 's parent.
- `<file>` that does not exist is created.
- Files are created with `-rw-r--` permission (with regards to the current effective user).
- If `<file>` refers to an existing file or directory, do nothing.
- If any ancestor directory in `<file>` does not exist, print: `touch: Ancestor directory does not exist`

`cp <src> <dst>`

- Copy a file `<src>` to a file `<dst>` .
- Require current effective user's read bit `r` on `<src>` .
- Require current effective user's execute bit `x` on `<src>` 's ancestors.
- Require current effective user's execute bit `x` on `<dst>` 's ancestors.
- Require current effective user's write bit `w` on `<dst>` 's parent.
- If `<dst>` already exists (e.g. `<src>` and `<dst>` share the same file name), and it refers to a file, print: `cp: File exists`
- If `<src>` does not exist, print: `cp: No such file`
- If `<dst>` refers to a directory, print: `cp: Destination is a directory`
- If `<src>` refers to a directory, print: `cp: Source is a directory`
- If `<dst>` does not exist, print: `cp: No such file or directory`

`mv <src> <dst>`

- Move a file `<src>` to a file `<dst>` .
- Require current effective user's execute bit `x` on `<src>` 's ancestors.
- Require current effective user's write bit `w` on `<src>` 's parent.
- Require current effective user's execute bit `x` on `<dst>` 's ancestors.
- Require current effective user's write bit `w` on `<dst>` 's parent.
- If `<dst>` already exists (e.g. `<src>` and `<dst>` share the same file name), and it refers to a file, print: `mv: File exists`
- If `<src>` does not exist, print: `mv: No such file`

- If `<dst>` refers to a directory, print: `mv: Destination is a direcotry`
- If `<src>` refers to a directory, print: `mv: Source is a directory`
- If `<dst>` does not exist, print: `mv: No such file or directory`

`rm <path>`

- Remove the file at `<path>` .
- Require current effective user's write bit `w` on `<path>` .
- Require current effective user's execute bit `x` on `<path>` 's ancestors.
- Require current effective user's write bit `w` on `<path>` 's parent.
- If `<path>` does not exist, print: `rm: No such file`
- If `<path>` refer to a directory, print: `rm: Is a directory`

`rmdir <dir>`

- Remove empty directory.
- Require current effective user's execute bit `x` on `<dir>` 's ancestors.
- Require current effective user's write bit `w` on `<dir>` 's parent.
- If `<dir>` cannot be found, print: `rmdir: No such file or directory`
- If `<dir>` is not a directory, print: `rmdir: Not a directory`
- If `<dir>` is a directory however not empty, print: `rmdir: Directory not empty`
- If `<dir>` is the current working directory, print: `rmdir: Cannot remove pwd`

`chmod [-r] <s> <path>`

- Change file mode bits.
- Require current effective user to be either the owner of `<path>` or `root` .
- Require current effective user's execute bit `x` of `<path>` 's ancestors.
- The format string (mode string) `<s>` is `[uoa...][-+=][perms...]` , where `perms` is either zero or more letters from the set `rwx` .
- If mode string `<s>` is invalid, print: `chmod: Invalid mode`
- If `<path>` cannot be found, print: `chmod: No such file or directory`
- If the current effective user is neither the owner of `<path>` nor `root` , print: `chmod: Operation not permitted`
- If `-r` is specified, change the mode of files and directories at `<path>` recursively.
  - If any error occurs when the recursion is ongoing, print the error message accordingly with the right order (dictionary order).

`chown [-r] <user> <path>`

- Change file owner. Only `root` user can perform this command.
- Require current effective user to be `root` .
- The user `<user>` is an existing user in the system.
- If `<user>` does not exist, print: `chown: Invalid user`
- If `<path>` cannot be found, print: `chown: No such file or directory`
- If the current user is not `root` , print: `chown: Operation not permitted`

`adduser <user>`

- Add a user to the system.
- Require current effective user to be `root` .
- If `<user>` already exists, print: `adduser: The user already exists`

`deluser <user>`

- Remove a user from the system.
- Require current effective user to be `root` .
- If `<user>` does not exist, print: `deluser: The user does not exist`
- If `<user>` is `root` , print:

```
WARNING: You are just about to delete the root account
Usually this is never required as it may render the whole system unusable
If you really want this, call deluser with parameter --force
(but this `deluser` does not allow `--force`, haha)
Stopping now without having performed any action
```

`su [user]`

- Switch the current effective user to `<user>` .
- No user requirement. (WARNING!)
  - Note, this is very dangerous in a real world situation because this allows anyone to switch to `root` .
  - Ideally, only `root` can perform this command, however this is omitted in the context of this assignment.
- If `<user>` is omitted, switch to `root` user.
- If `<user>` does not exist, print: `su: Invalid user`

`ls [-a] [-d] [-l] [path]`

- List information about `<path>` . Sort entries alphabetically.
- Require current effective user's read bit `r` on `<path>` if `<path>` is a valid directory.

- Require current effective user's read bit `r` on `<path>` 's parent if `<path>` is a valid file or `-d` is specified.
- Require current effective user's execute bit `x` on `<path>` 's ancestors.
- If `-a` is not specified, ignore entries starting with `.` .
- If `-a` is specified, do not ignore entries starting with `.` .
- If `-l` is not specified, use simple format, e.g. file or directory name.
- If `-l` is specified, use a long listing format.
  - See the example in "Permission and Ownership".
- If `<path>` is not specified, the current directory is listed.
- If `<path>` is a file, just print the file name.
- If `<path>` is a directory and `-d` is not specified, list information about its contents.
- If `<path>` is a directory and '-d is specified, list the directory itself, not its contents.
- If `<path>` does not exist, print: `ls: No such file or directory`

Other error handling

`Invalid syntax`

When given

- too many or too few arguments;
- a command line that contains invalid characters;
- a non-exist flag to with a valid command

the program should output:

```
<invalid_command>: Invalid syntax
```

`Command not found`

When given any invalid command, the program should output:

```
<invalid_command>: Command not found
```

Example

```
root:/$ pwp
pwp: Command not found
```

`Permission denied`

Whenever the current effective user does not have the correct permission to perform certain operation, print: `<cmd>: Permission denied`

Note: `Operation not permitted` error has higher priority than `Permission denied` .

Example

```
root:/$ mkdir -p folder1/folder2
root:/$ ls -l
drwxr-x root folder1
root:/$ ls -l folder1
drwxr-x root folder2
root:/$ chmod o-x folder1
root:/$ ls -l
drwxr-- root folder1
root:/$ adduser bob
root:/$ su bob
bob:/$ ls -d -l
drwxr-x root .
bob:/$ ls -d -l /
drwxr-x root /
bob:/$ ls -l folder1
ls: Permission denied
bob:/$ ls -l folder1/folder2
ls: Permission denied
bob:/$ su
root:/$ chmod o=x folder1
root:/$ su bob
bob:/$ ls -l folder1
ls: Permission denied
bob:/$ ls -l folder1/folder2
bob:/$ ls -a -l folder1/folder2
drwxr-x root .
drwx--x root ..
bob:/$ ls -d -l folder1
drwx--x root folder1
bob:/$ ls -d -l folder1/folder2
drwxr-x root folder1/folder2
bob:/$ exit
bye, bob
```

Testing

You are expected to write a number of test cases for your program. Example tests will be provided later. You are expected to test as many as possible execution paths of your code.

We will provide you with some test cases but these do not test all the function-

ality described in the assignment. It is important that you thoroughly test your code by writing your own end-to-end (input/output) tests.

Unit tests are not required and not assessable.

You must place all of your end-to-end test cases in the `e2e_tests/` directory. Ensure that each pair of test case must have an `<name>.in` input file and an `<name>.out` output file. Your own tests will not be marked if this naming convention is not followed. We recommend that the names of your test cases are descriptive so that the reader can easily know what each case is testing. E.g. `simple_ls_1.in` , `complex_cd_2.in` .

You should have a brief description of your tests, and how they can be run, in `README.md` . Please keep it concise.

`Coverage.py` will be used to generate the coverage of your own tests and the coverage rate will determine some of the testing marks.

Implementation

The assignment is to be implemented in Python. A set of scaffold files will be provided. You are expected to write legible code with good style, e.g. PEP 8.

You are **NOT** allowed to import **ANY** Python modules. You are free to use all builtin functions of Python. If you want to use an additional module which will not trivialize the assignment, please ask on Ed. The allowed library list may be extended. Related announcement should be posted accordingly.

You must **NOT** create, access or delete any actual files on disk. You must **NOT** preserve any data temporarily or permanently in any actual files on disk, but the memory.

Please be mindful: breaching the above restrictions will result in harsh deduction for the entire assignment.

How to run Simple Nautilus

```
python3 nautilus.py
```

Submitting your code

An Ed Lesson workspace will be available for you to test and submit your code. Public test cases will be released up to **Week 7, Thursday, September 15th**. Additionally, there could be a set of unreleased test cases which will be run against your code after the due date.

You will need to use `git` to submit your code. To learn what `git` is and how to setup `git` (and SSH key) properly, please refer to Lecture 4.

Where to start

Note: the following numbering does not necessarily suggest the ordering that you should consider them.

1. Consider the data structure needed to maintain the file system.

   It can be class-based (e.g. tree structure) or non-class-based (e.g. dictionary in dictionary).

2. If you wish to reach Task 3, consider the data structure needed to maintain the user system and metadata of files.

   It could be easier if tree structure is used.

3. Consider the hierarchy and the order of error handling.

   What error should be processed first?

4. Consider what information need to be parsed and attempt to write reusable parser function.

   How do you verify a command line is syntactically valid?

   How do you break down a command line to multiple arguments?

5. Prepare tests gradually.

6. Start planning earlier.

   Note, planning does not mean coding. In fact, as an analogy, it could mean you know how to solve the problem with pens and paper. If you know how to deal with the problem, coding is just a way to express your strategy.

7. `ls` can be helpful.

   Although no mark will be awarded to `ls` if not all public test cases in Task 1 and 2 have been completed, it can be very helpful to implement a partially working `ls` in the system. It will help you navigate in the file system and verify if the system is functioning.

Marking Criteria

The assignment will be marked with an automatic testing system on Ed. A mark will be given based on a percentage of tests passed (15%) and a manual mark will be given for overall style, quality, readability, etc (2%).

Each task has an associated weighting that is a portion of the total marks on passing test cases.

As a condition, you will NOT be able to receive marks for Task 3 unless all public test cases in Task 1 and 2 have been completed.

- Task 1 (6%)

  To complete this task, you need to implement the following commands:

  - `exit`
  - `pwd`
  - `cd`
  - `mkdir`
  - `touch`

  Note: since `root` can do anything without permission error, Task 1 can be implemented without considering the user management and the permission and the ownership of files.

- Task 2 (5%)

  To complete this task, you need to implement the following commands:

  - `cp`
  - `mv`
  - `rm`
  - `rmdir`

  Note: since `root` can do anything without permission error, Task 2 can be implemented without considering the user management and the permission and the ownership of files.

- Task 3 (4%)

  To complete this task, you need to implement the following commands:

  - `chmod`
  - `chown`
  - `adduser`
  - `deluser`
  - `su`
  - `ls`

  Note: Task 3 cannot be implemented without considering the user management and the permission and the ownership of files.

You are expected to write your own tests and submit them with your code, and a mark will be given based on coverage and manual inspection of your tests (3%).

Your test cases must cover as many as possible execution paths and designed to test specific features. These test cases are for the executable code section written in Python for all three tasks.

There will be public test cases made available for you to test against, but there will also be extra non-public tests used for marking. Success with the public tests doesn't guarantee your program will pass the private tests.

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 7, Tuesday, September 13th**. No major changes will be made. Revised versions will be clearly marked and the most recent version announced to the class via Ed Discussions.

Census date is your last opportunity to withdraw from Semester 2 (S2C) units you are enrolled in without financial or academic penalty. The main census date for Semester 2 (S2C) is Wednesday, 31 August, 2022.

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 you do not properly follow the assignment description, or your code is unnecessarily or deliberately obfuscated.

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.

Changes

Major changes will be announced at here.