The Shell — Practice

CSCI 40

Fall 2026

Total Score: _____/4

LLM evaluation	*2026-08-26*
Re-evaluation pending after quiz-bank corrections.	

Note. Each snippet starts in a fresh, empty `quiz` directory. Setup commands are omitted.

Intro to the Unix Shell

Note. The Unix shell was invented in the early 1970s. It is the oldest programming language still in wide use today. It is the "glue" we use to connect all of our python programs together.

Note. There are three principles of the *Unix Philosophy*:

1. Write programs that do one thing and do it well.
2. Write programs to work together.
3. Write programs to handle text streams, because text is a universal interface.

1 Connecting Programs

Note. The *output redirection operator* `>` takes the output of a program and saves it to a file. If the file already exists, the contents are overwritten. The *append operator* `>>` is like `>` but it appends instead of overwriting. The *input redirection operator* `<` opens a file and passes the contents as the input to a command. The *pipe* `|` takes the output of the previous command and sends it to the input of the subsequent command. The *semicolon* `;` is used to separate multiple commands on the same line.

Problem 1. Write the output of the final command in the following shell script.

```
$ echo 'hello world' >> README
$ echo 'hola mundo' > README
$ echo 'salve munde' >> README
$ cat README | wc -l
```

Problem 2. Write the output of the final command in the following shell script.

```
$ echo 'hello world' >> README
$ echo 'hola mundo' >> README
$ echo 'salve munde' >> README
$ wc -l README
```

Problem 3. Write the output of the final command in the following shell script.

```
$ echo 'hello world' > README
$ echo 'hola mundo' >> README
$ echo 'salve munde' > README
$ wc -l < README
```

2 Python in the Shell

Note. It is common to use one programming language to write the code of another programming language. Here are some examples of using the shell to write Python.

Problem 4. Write the output of the final command in the following shell script.

```
$ echo 'x = 1' >> example.py
$ echo 'x += 1' >> example.py
$ echo 'print("x=", x)' >> example.py
$ python3 example.py
```

Problem 5. Write the output of the final command in the following shell script.

```
$ echo 'xs = [1, 2]' >> example.py
$ echo 'xs.append(5)' >> example.py
$ echo 'print("xs=", xs)' >> example.py
$ python3 example.py
```

Note. The << operator creates a *heredoc*. This operator allows you to define a document "right here" in the shell (i.e. without creating the document on disk) to be passed to the command as input. The << operator is immediately followed by a string, and the heredoc is terminated when the exact same string appears on a line by itself. EOF (end of file) is the traditional choice of string, but other strings are also valid.

Problem 6. Write the output of the final command in the following shell script.

```
$ python3 <<EOF
x = 1
x += 1
print("x=", x)
EOF
```

Problem 7. Write the output of the final command in the following shell script.

```
$ cat > example.py <<EOF
x = 1
x += 1
print("x=", x)
EOF
$ python3 example.py
```

Problem 8. Write the output of the final command in the following shell script.

```
$ cat > example.py <<ROFL
x = 1
x += 1
print("x=", x)
ROFL
$ python3 example.py
```

3 Getting Arguments into Python

Note. The Python `argparse` library is a high-level interface for getting command-line arguments into Python. This library is a wrapper around the lower-level `sys.argv` list that contains the raw, literal values passed in. The first value is always the name of the program being run, and then the remaining values are the arguments.

Note. Quotation marks are handled by the shell, and not by the program. Therefore putting quotation marks around the parameters to a program can affect the number of parameters that the program sees. Also notice that the program never observes the type of quotation marks used.

Problem 9. Write the output of the final command in the following shell script.

```
$ cat > example.py <<EOF
import sys
print("sys.argv=", sys.argv)
EOF
$ python3 example.py hello world
```

Problem 10. Write the output of the final command in the following shell script.

```
$ cat > example.py <<EOF
import sys
print("sys.argv=", sys.argv)
EOF
$ python3 example.py "hello world"
```

Problem 11. Write the output of the final command in the following shell script.

```
$ cat > example.py <<EOF
import sys
print("sys.argv=", sys.argv)
EOF
$ python3 example.py 'hello world'
```

Problem 12. Write the output of the final command in the following shell script.

```
$ cat > example.py <<EOF
import sys
num_args = len(sys.argv)
print("num_args=", num_args)
EOF
$ python3 example.py "salve munde" "hola mundo"
```

Problem 13. Write the output of the final command in the following shell script.

```
$ cat > example.py <<EOF
import sys
num_args = len(sys.argv)
print("num_args=", num_args)
EOF
$ python3 example.py "salve munde' 'hola mundo"
```

Problem 14. Write the output of the final command in the following shell script.

```
$ cat > example.py <<EOF
import sys
num_args = len(sys.argv)
print("num_args=", num_args)
EOF
$ python3 example.py
```

Note. Recall that the `touch` command (for each parameter that gets passed in) creates a new file if the file does not already exist, or updates the "modified timestamp" for files that already exist. Based on your understanding of command line parameters and quotation marks above, the problems below should make sense.

Problem 15. Write the output of the final command in the following shell script.

```
$ touch hello_world
$ touch hola_mundo
$ touch salve_munde
$ ls | wc -l
```

Problem 16. Write the output of the final command in the following shell script.

```
$ touch 'hello world'
$ touch "hola mundo"
$ touch salve munde
$ ls | wc -l
```

Problem 17. Write the output of the final command in the following shell script.

```
$ touch 'hello world' hola "mundo salve munde"
$ ls | wc -l
```

Note. Environment variables are another popular way of passing information to a Python program. They are commonly used for passwords, API keys, and other sensitive information. In the shell, they are created with the `export` keyword. There must not be any spaces around the equals sign =. In Python, the function `os.getenv` is used to get the value of an environment variable. The first parameter is the name of the variable, and the second parameter is a default value to use if the environment variable has not been set.

Problem 18. Write the output of the final command in the following shell script.

```
$ export password="hello world"
$ cat > quiz.py <<'EOF'
import os
print(os.getenv('password', 'hola mundo'))
EOF
$ python3 quiz.py
```

Problem 19. Write the output of the final command in the following shell script.

```
$ export password=hello
$ cat > quiz.py <<'EOF'
import os
print(os.getenv('password', 'hola'))
EOF
$ python3 quiz.py
```

Problem 20. Write the output of the final command in the following shell script.

```
$ export PASSWORD=hello
$ cat > quiz.py <<'EOF'
import os
print(os.getenv('password', 'hola'))
EOF
$ python3 quiz.py
```

4 Hidden Files and Folders

Note. In Unix, any file beginning with a dot is a hidden file. It is common to place configuration settings in these hidden files. Every folder contains two special hidden files `.` (which refers to the current folder) and `..` (which refers to the parent folder).

Note. The `ls` command can take an optional argument `-a` to list hidden files. It can also take an argument specifying the folder to list files inside of.

Problem 21. Write the output of the final command in the following shell script.

```
$ touch README
$ echo 'API_KEY=asd' > .env
$ ls -l
```

Problem 22. Write the output of the final command in the following shell script.

```
$ touch README
$ echo 'API_KEY=asd' > .env
$ ls -la
```

Problem 23. Write the output of the final command in the following shell script.

```
$ touch README
$ mkdir test
$ touch test/README
$ cd test
$ echo 'API_KEY=asd' > ../.env
$ ls -la
```

Problem 24. Write the output of the final command in the following shell script.

```
$ touch README
$ mkdir test
$ mkdir test/test
$ touch test/README
$ touch test/test/README
$ cd test/../test
$ ls | wc -l
```

Problem 25. Write the output of the final command in the following shell script.

```
$ touch README
$ mkdir test
$ cd test
$ mkdir test
$ touch README
$ cd ..
$ touch README
$ ls | wc -l
```

Problem 26. Write the output of the final command in the following shell script.

```
$ touch README
$ mkdir test
$ touch test/README
$ touch test/.env
$ ls -a test | wc -l
```

Problem 27. Write the output of the final command in the following shell script.

```
$ touch README
$ mkdir test
$ touch test/README
$ touch test/.env
$ ls test | wc -l
```

Problem 28. Write the output of the final command in the following shell script.

```
$ touch README
$ mkdir test
$ mkdir test/test
$ touch test/README
$ touch test/test/README
$ cd test/../test
$ ls . | wc -l
```

Problem 29. Write the output of the final command in the following shell script.

```
$ touch README
$ mkdir test
$ mkdir test/test
$ touch test/README
$ touch test/test/README
$ cd test/../test
$ ls .. -a | wc -l
```

5 Version control with git

Note. `git` is a *version control system*. It lets us track different versions of our files, and "travel back in time" to old versions if we need to. `git` is especially useful for facilitating collaboration between many humans (or AIs). A good understanding of `git` is necessary for getting a good industry job.

Note. In these problems, `git init` creates a hidden `.git` directory and starts on the `main` branch. `git add` selects files for the next saved version, and `git commit` saves that version. `git branch foo` creates a branch named `foo`; `git checkout foo` switches to it. Switching branches restores the committed versions of tracked files for that branch. A file that has not been added to a commit stays in the working directory when you switch.

Problem 30. Write the output of the final command in the following shell script.

```
$ git init
$ touch README
$ mkdir foo
$ touch foo/README2
$ touch README3
$ ls foo | wc -l
```

Problem 31. Write the output of the final command in the following shell script.

```
$ git init
$ touch README
$ mkdir foo
$ touch foo/README2
$ touch README3
$ ls -a foo | wc -l
```

Problem 32. Write the output of the final command in the following shell script.

```
$ git init
$ touch README
$ mkdir foo
$ touch foo/README2
$ touch README3
$ ls -a . | wc -l
```

Problem 33. Write the output of the final command in the following shell script.

```
$ git init
$ mkdir master
$ echo "print('hello world')" > master/foo.py
$ echo "print('hola mundo')" > foo.py
$ git add master
$ git commit -m "added master"
$ git branch foo
$ git checkout foo
$ echo "print('hola mundo')" >> foo.py
$ git add foo.py
$ git commit -m "modified foo"
$ git checkout main
$ ls -1
```

Problem 34. Write the output of the final command in the following shell script.

```
$ git init
$ echo "print('hello world')" > foo.py
$ echo "print('hola mundo')" > bar.py
$ git add foo.py
$ git commit -m "added foo"
$ git branch foo
$ git checkout foo
$ echo "print('hola otra vez')" >> foo.py
$ git add foo.py
$ git add bar.py
$ git commit -m "modified foo"
$ git checkout main
$ echo "print('hello again')" >> bar.py
$ python3 bar.py
```

Problem 35. Write the output of the final command in the following shell script.

```
$ git init
$ echo "print('hello world')" > foo.py
$ git add foo.py
$ git commit -m "added foo"
$ git branch foo
$ git checkout foo
$ echo "print('hola mundo')" >> foo.py
$ git add foo.py
$ git commit -m "modified foo"
$ git checkout main
$ python3 foo.py
```