Assignment 2: Sheepy

version: 1.4 last updated: 2023-07-22 18:00

Aims

This assignment aims to give you

- practice in Python programming generally
- experience in translating between complex formats with Python
- clarify your understanding of Shell syntax & semantics
- introduce you to Python syntax & semantics

Introduction

Your task in this assignment is to write a POSIX Shell Transpiler.
Generally, compilers take a high-level language as input and output assembler, which can then can be directly executed.
A Transpiler (or Source-to-Source Compiler) takes a high-level language as input and outputs a different high-level language.
Your transpiler will take Shell scripts as input and output Python.
Such a translation is useful because programmers sometimes convert Shell scripts to Python.
Most commonly this is done because extra functionality is needed, e.g. a GUI.
And this functionality is much easier to implement in Python.
Your task in this assignment is to automate this conversion.
You must write a Python program that takes as input a Shell script and outputs an equivalent Python program.

The translation of some POSIX Shell code to Python is straightforward.
The translation of other Shell code is difficult or infeasible.
So your program will not be able to translate all Shell code to Python.
But a tool that performs only a partial translation of shell to Python could still be very useful.

You should assume the Python code output by your program will be subsequently read and modified by humans.
In other words, you have to output readable Python code.
For example, you should aim to preserve variable names and comments.

Your compiler must be written in Python.
You must call your Python program `sheepy.py`.
It will be given a single argument, the path to a Shell script as its first command line argument.
It should output, to standard output, the equivalent Python code.
For example:

```
$ cat gcc.sh
#!/bin/dash

for c_file in *.c
do
    gcc -c $c_file
done
$ ./sheepy.py gcc.sh
#!/usr/bin/python3 -u

import glob, subprocess

for c_file in sorted(glob.glob("*.c")):
    subprocess.run(["gcc", "-c", c_file])
```

If you look carefully at the example above you will notice the Python code does not have exactly the same semantics as the shell code.
If there are no `.c` files in the current directory the for loop in the shell program executes once and tries to compile a non-existent file named `*.c` whereas the Python for loop does not execute.
And if the file name contains spaces the shell code will pass it as multiple arguments to gcc but the Python code will pass it as a single argument - in other words the shell breaks but the Python works.

This is a general issue with translating Shell to Python.
In many cases, the natural translation of the shell code will have slightly different semantics.
For some purposes, it might be desirable to produce more complex Python code that matches the semantics exactly.
For example:

```
#!/usr/bin/python3 -u

import glob, subprocess

if glob.glob("*.c"):
    for c_file in sorted(glob.glob("*.c")):
        subprocess.run(["gcc", "-c"] + c_file.split())
else:
    subprocess.run(["gcc", "-c", "*.c"])
```

This is not desirable for our purposes.
Our goal is to produce the clearest most human-readable code so the first (simpler) translation is more desirable.

Subsets

The shell features you need to implement is described below as a series of subsets.

It suggested you tackle the subset in the order listed but this is not required.

Subset 0

echo

The `echo` builtin is used to output a string to stdout.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`echo hello world`
`echo 42 is the meaning of life, the universe, and everything`
`echo To be or not to be: that is the question`
	`#!/usr/bin/python3 -u`
	
	`print("hello world")`
	`print("42 is the meaning of life, the universe, and everything")`
	`print("To be or not to be: that is the question")`

HINT:

The `echo` builtin outputs each argument separated by a single space.

i.e:

```
echo Hello      World
```

is output as

```
Hello World
```

=

The `=` operator is used to assign a value to a variable.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`x=1`
`y=2`
`foo=hello`
`bar=world`
`course_code=COMP2041`
`AssignmentName=Sheepy`
	`#!/usr/bin/python3 -u`
	
	`x = "1"`
	`y = "2"`
	`foo = "hello"`
	`bar = "world"`
	`course_code = "COMP2041"`
	`AssignmentName = "Sheepy"`

HINT:

Remember that Shell assignment operator requires no spaces around the `=` operator.

Whereas Python allows spaces around the `=` operator.

> As you should aim to produce readable Python code you should include spaces around the `=` operator.
>
> Remember that Shell doesn't have integers, so all values are strings.

$

The `$` operator is used to access the value of a variable.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`theAnswer=42`
`echo The meaning of life, the universe, and everything is $theAnswer`
 	
`name=COMP2041`
`echo I hope you are enjoying $name this semester`
 	
`H=Hello`
`W=World`
`echo $H, $W`
 	
`P1=race`
`P2=car`
`palindrome=$P1$P2`
`echo $palindrome`
	`#!/usr/bin/python3 -u`
	
	`theAnswer = "42"`
	`print(f"The meaning of life, the universe, and everything is {theAnswer}")`
	
	`name = "COMP2041"`
	`print(f"I hope you are enjoying {name} this semester")`
	
	`H = "Hello"`
	`W = "World"`
	`print(f"{H}, {W}")`
	
	`P1 = "race"`
	`P2 = "car"`
	`palindrome = f"{P1}{P2}"`
	`print(f"{palindrome}")`

> HINT:
>
> There may be multiple correct ways to translate a Shell script.
>
> Again: remember that Shell doesn't have integers, so all values are strings.
> This will simplify your translation immensely.

> HINT:
>
> Care needs to be taken as to not redefine Python function and keywords.
> The shell code
>
> ```
> print=Oops
> ```
>
> could result in the Python code that makes the `print` function inaccessible.
> The shell code
>
> ```
> pass=Oops
> ```
>
> would result in a syntax error, as `pass` is a Python keyword and cannot be redefined.

#

The `#` operator is used to start a comment.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`# This is a comment`
 	
`echo hello world # This is also a comment`
	`#!/usr/bin/python3 -u`
	
	`# This is a comment`
	
	`print("hello world") # This is also a comment`

> NOTE:

> As comments do not effect the output of your program, your program will work correctly if you simply remove all comments.
>
> This will be penalized during manual marking, your generated python program should contain all the comments from the input shell script.

> HINT:
>
> Remember that Shell comments start with a `#` character and continue to the end of the line.

Subset 1

globbing

The `*`, `?`, `[`, and `]` characters are used in globbing.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`echo *`
 	
`C_files=*.[ch]`
`echo $C_files`
 	
`echo all of the single letter Python files are: ?.py`
	`#!/usr/bin/python3 -u`
	
	`import glob`
	
	`print(" ".join(sorted(glob.glob("*"))))`
	
	`C_files = "*.[ch]"`
	`print("`
	`".join(sorted(glob.glob(C_files))))`
	
	`print("all of the single letter Python`
	`files are: " + "`
	`".join(sorted(glob.glob("?.py"))))`

Extra whitespace has been added to the Python code so that the position of each line matches the Shell code

You do not need to add extra whitespace to your Python code.

> NOTE:
>
> In this subset globbing characters will only be used for globbing or in a comment.
> That is, if you see a `*`, `?`, `[`, or `]` character it *is* a globbing character (unless it is in a comment).

> HINT:
>
> `glob.glob()` can be used to perform globbing in Python.

> NOTE:
>
> As shown in the example above, globbing characters aren't expanded until they are used, not when they are assigned to a variable.
>
> This may be difficult to implement, and in most cases expanding globbing characters when they are assigned to a variable is acceptable.

for

The `for`, `in`, `do`, and `done` keywords are used to start and end for loops.
For example:

Shell	Possible **Python translation**

```
#!/bin/dash

for i in 1 2 3
do
    echo $i
done

for word in this is a string
do
    echo $word
done

for file in *.c
do
    echo $file
done
```

```
#!/usr/bin/python3 -u

for i in ["1", "2", "3"]:

    print(i)


for word in ["this", "is", "a", "string"]:

    print(word)


for file in sorted(glob.glob("*.c")):

    print(file)
```

Extra whitespace has been added to the Python code so that the position of each line matches the Shell code

You do not need to add extra whitespace to your Python code.

> NOTE:
>
> In this subset you do not need to handle nested `for` loops.

exit

The `exit` builtin is used to exit the shell.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`echo hello world`
`exit`
`echo this will not be printed`
`exit 0`
`echo this will double not be printed`
`exit 3`
	`#!/usr/bin/python3 -u`
	
	`import sys`
	
	`print("hello world")`
	`sys.exit()`
	`print("this will not be printed")`
	`sys.exit(0)`
	`print("this will double not be printed")`
	`sys.exit(3)`

> HINT:
>
> `sys.exit()` can be used to exit the Python program.

cd

The `cd` builtin is used to change the current working directory.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`echo *`
`cd /tmp`
`echo *`
`cd ..`
`echo *`
	`#!/usr/bin/python3 -u`
	
	`import glob, os`
	
	`print(" ".join(sorted(glob.glob("*"))))`
	`os.chdir("/tmp")`
	`print(" ".join(sorted(glob.glob("*"))))`
	`os.chdir("..")`
	`print(" ".join(sorted(glob.glob("*"))))`

> HINT:
>
> `os.chdir()` can be used to change the current working directory.

read

The `read` builtin is used to read a line from stdin.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`echo What is your name:`
`read name`
 	
`echo What is your quest:`
`read quest`
 	
`echo What is your favourite colour:`
`read colour`
 	
`echo What is the airspeed velocity of an unladen swallow:`
`read velocity`
 	
`echo Hello $name, my favourite colour is $colour too.`
	`#!/usr/bin/python3 -u`
	
	`print("What is your name:")`
	`name = input()`
	
	`print("What is your quest:")`
	`quest = input()`
	
	`print("What is your favourite colour:")`
	`colour = input()`
	
	`print("What is the airspeed velocity of an unladen swallow:")`
	`velocity = input()`
	
	`print (f"Hello {name}, my favourite colour is {colour} too.")`

> HINT:
>
> `input()` can be used to read a line from stdin.

External Commands

Any line that is not a known builtin, keyword, or other shell syntax should be treated as an external command.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`touch test_file.txt`
`ls -l test_file.txt`
 	
`for course in COMP1511 COMP1521 COMP2511 COMP2521 # keyword`
`do`
`# keyword`
`    echo $course`
`# builtin`
`    mkdir $course`
`# external command`
`    chmod 700 $course`
`# external command`
`done`
`# keyword`
	`#!/usr/bin/python3 -u`
	
	`import subprocess`
	
	`subprocess.run(["touch", "test_file.txt"])`
	`subprocess.run(["ls", "-l", "test_file.txt"])`
	
	`for course in ["COMP1511", "COMP1521", "COMP2511", "COMP2521"]:`
	
	`    print(f"{course}")`
	`    subprocess.run(["mkdir", course])`
	`    subprocess.run(["chmod", "700", course])`

Extra whitespace has been added to the Python code so that the position of each line matches the Shell code

You do not need to add extra whitespace to your Python code.

> HINT:

> `subprocess.run()` can be used to run external commands.

Subset 2

Command Line Arguments

The `$0`, `$1`, `$2`, etc. variables are used to access the command line arguments.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`echo This program is: $0`
 	
`file_name=$2`
`number_of_lines=$5`
 	
`echo going to print the first`
`$number_of_lines lines of $file_name`
	`#!/usr/bin/python3 -u`
	
	`import sys`
	
	`print(f"This program is: {sys.argv[0]}")`
	
	`file_name = sys.argv[2]`
	`number_of_lines = sys.argv[5]`
	
	`print(f"going to print the first`
	`{number_of_lines} lines of {file_name}")`

> NOTE:
>
> Only the numbers 0-9 can be used with the `$` operator.
>
> numbers 10 and above require the `${}` syntax.
> (See the next section)

${}

The `${}` operator is used to access the value of a variable.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`string=BAR`
`echo FOO${string}BAZ`
	`#!/usr/bin/python3 -u`
	
	`string = "BAR"`
	`print(f"FOO{string}BAZ")`

> NOTE:
>
> This is similar to the `$` operator but allows for strings to be appended immediately after the variable name.
>
> Only simple expressions will be used. Not `${var:default}`, `${var#word}`, `${var%%word}`, etc

test

The `test` builtin is used to test a condition.

> HINT:
>
> `os.access()` and `os.path` will be useful for implementing `test`.

> NOTE:
>
> In this subset it is only to be used as the condition in an `if` statements and `while` loops.

> NOTE:
>
> You **do** have to handle **all** `test` operators such `=`, `-eq`, `-z`, `-r`, `-d`, etc.

> **Except** for `test`'s `-ef` and `-t` operators as they don't have a simple translation to Python.

if

The `if`, `then`, `elif`, `else`, and `fi` keywords are used to start and end if statements.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`if test -w /dev/null`
`then`
`    echo /dev/null is writeable`
`fi`
	`#!/usr/bin/python3 -u`
	
	`import os`
	
	`if os.access("/dev/null", os.W_OK):`
	
	`    print("/dev/null is writeable")`

> NOTE:
>
> In this subset you do not need to handle nested `if` statements.
>
> In this subset the `if` condition must be a single `test` expression.

while

The `while`, `do`, and `done` keywords are used to start and end while loops.
For example:

Shell	Possible **Python translation**

`#!/bin/dash`
 	
`row=1`
`while test $row != 11111111111`
`do`
`    echo $row`
`    row=1$row`
`done`
	`#!/usr/bin/python3 -u`
	
	`row = "1"`
	`while row != "11111111111":`
	
	`    print(row)`
	`    row = f"1{row}"`

> NOTE:
>
> In this subset you do not need to handle nested `while` statements.
>
> In this subset the `while` condition must be a single `test` expression.

> NOTE:
>
> As we do not have the `$(())` operator yet (see subset 4), we cannot increment a loop counters.
>
> Instead we are using string concatenation to "expand" the loop counter.

Single Quotes

The `'` character is used to start and end a single-quoted string.
For example:

```
#!/bin/dash

echo 'hello    world'

echo 'This is not a $variable'

echo 'This is not a glob *.sh'
```

> NOTE:
>
> Strings inside single-quotes do not have variables expanded,
> do not have globbing characters expanded,
> and do not have whitespace removed.

Subset 3

Double Quotes

The `"` character is used to start and end a double-quoted string.
For example:

```
#!/bin/dash

echo "hello    world"

echo "This is sill a $variable"

echo "This is not a glob *.sh"
```

> NOTE:
>
> Strings inside double-quotes **do** have variables expanded,
> but not globbing characters expanded,
> or have whitespace removed.

backticks

A command substitution can be started and ended with a `` ` `` character (backtick).

For example:

```
#!/bin/dash

date=`date +%Y-%m-%d`

echo Hello `whoami`, today is $date

echo "command substitution still works in double quotes: `hostname`"

echo 'command substitution does not work in single quotes: `not a command`'
```

> NOTE:
>
> Backticks will not be nested.
>
> Backticks are the original syntax for command substitution.
>
> The new and better syntax for command substitution `$()` is in subset 4.

echo -n

The *-n* flag for `echo` tells it not to print a newline at the end of the output
For example:

```
#!/bin/dash

echo -n "How many? "
read n
```

> NOTE:
>
> Echo only accepts a single flag, *-n*, and it must be the first argument.
>
> *-n* at any other location, or any other flag, should be treated as a normal string.

number of command line arguments

The `$#` variable is used to access the number of command line arguments.
For example:

```
#!/bin/dash

echo I have $# arguments
```

command line argument lists

The `$@` variable is used to access all the command line arguments.
For example:

```
#!/bin/dash

echo "My arguments are $@"
```

> NOTE:
>
> Make sure you translate `$@` appropriately both when quoted and not quoted.
> As the behaviour of `$@` differs depending on this context.

if/while/for nesting

In this subset, `if`, `while`, and `for` expressions can now be nested inside themselves and each other.
For example:

```
#!/bin/dash

i='!'
while test $i != '!!!!!!'
do
    j='!'
    while test $j != '!!!!!!'
    do
        echo -n ". "
        j="!$j"
    done
    echo
    i="!$i"
done

for file in *.txt
do
    if test -f "$file"
    then
        dos2unix "$file"
    fi
done
```

Subset 4

`[` and `]`

The `[` and `]` characters are used to start and end a test expression.
The `[` builtin is the same as the `test` builtin.
Except that the `[` builtin requires a `]` as the last argument.

case

The `case`, `in`, `|`, `)`, `esac`, and `;;` keywords are used to start and end case statements.
For example:

```
#!/bin/dash

case $# in
    0)
        echo no arguments
        ;;
    1)
        echo one argument
        ;;
    2|3|4)
        echo some arguments
        ;;
    *)
        echo many arguments
        ;;
esac
```

> HINT:
>
> The conditions (expressions before the `)`) in a `case` statement is a glob
>
> Multiple conditions can be OR'ed together using `|`

> NOTE:
>
> `|`, `)`, and `;` are only used in case statements.
>
> apart from in strings and comments: `|`, `)`, and `;` will only be used in case statements, not for their other purposes in Shell scripts.

$()

The `$()` operator is used for command substitution.
The `$()` operator is the same as the `` ` `` operator.
Except that The `$()` operator may be nested.
For example:

```
#!/bin/dash

date=$(date +%Y-%m-%d)

echo Hello $(whoami), today is $date

echo "command substitution still works in double quotes: $(hostname)"

echo 'command substitution does not work in single quotes: $(not a command)'

echo "The groups I am part of are $(groups $(whoami))"
```

$(())

The `$(())` operator is used to evaluate an arithmetic expression For example:

```
#!/bin/dash

x=6
y=7
echo $((x  + y))
```

<, >, and >>

The `<`, `>`, and `>>` operators are used to redirect stdin and stdout respectively.
For example:

```
#!/bin/dash

echo hello >file
echo world >> file
cat <file
```

> NOTE:
>
> You do not need to implement stderr redirection. (i.e. `2>`)
>
> You do not need to implement redirection to a file descriptor. (i.e. `>&2`)
>
> Input and Output redirection will only be used after the command and it's arguments.
>
> At is:
> While
>
> ```
> <file command arg1 arg2
> ```
>
> is valid shell, it will not be tested.
> Only
>
> ```
> command arg1 arg2 <file
> ```
>
> will be tested.
>
> Note that both input and output redirection can be used at the same time.

&& and ||

The `&&` and `||` operators are used to perform boolean logic.
For example:

```
#!/bin/dash

test -w /dev/null && echo /dev/null is writeable
test -x /dev/null || echo /dev/null is not executable
```

if/while conditions

In this subset, `if` and `while` conditions can now be any external command, and/or multiple commands joined by `&&` or `||`.
For example:

```
#!/bin/dash

if test -w /dev/null && test -x /dev/null
then
    echo /dev/null is writeable and executable
fi

if grep -Eq $(whoami) enrolments.tsv
then
    echo I am enrolled in COMP2041/9044
fi
```

Examples

Some examples of shell code and possible translations are available as a table or a zip file

These examples should provide most the information you need to tackle subsets 0 & 1.

Translating subsets 2-4 will require you to discover information from online or other resources.
This is a deliberately part of the assignment.

The Python you output can and probably will be different to the examples you've been given.

So there is no way to directly test if your Python output is correct.

But the Python you output when run has to behave the same as the input shell script it was generated from.
So a good check of any translation is to execute the Shell and the Python and then use `diff` to check that their output is identical.

Assumptions/Clarifications

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

You can assume the code you are given is Shell which works with the version of on CSE systems (essentially POSIX compatible).

Other shells such as Bash contain many other features. If these features and not present in `/bin/dash` on CSE machines you do not have to handle these

You do not need to implement keywords & builtins not listed above, for example the pipe operator ('|') does not appear above so you do not need to translate pipelines.

You should implement the keywords & builtins listed above directly in Python, you cannot execute them indirectly via `subprocess` or other Python modules. For example, this is not an acceptable translation.

```
subprocess.call("for c_file in *.c; do gcc -c $c_file; done", shell=True)
```

The only shell builtins which you must translate directly into Python are:

```
exit read cd test echo
```

The builtins (`exit read cd`) must be translated into Python to work. For example this Python code does not work:

```
subprocess.call(['exit'])
```

The last 2 (`test echo`) can be, and often are also implemented by stand-alone programs. So, for example, this Python code will work:

```
subprocess.call(['echo','hello','world'])
```

Doing this will receive no marks, instead of using `subprocess.call` you should translate uses of `test` , and `echo` directly to Python, e.g.:

```
print "hello world"
```

The only Shell builtin option you need to handle is echo's -n option.
You do not need to handle other echo options such as `-e` .
You do not need to handle options for other builtins.
For example, you do not need to handle the various (rarely-used) *read* options.

Dash has many special variables.
You need only handle a few of these, which indicate the shell script's arguments.
These special variables need be translated:

```
$# $@ $0 $1 $2 $3 ...
```

You assume the shell scripts you are given execute correctly on a CSE lab machine.
Your program does not have to detect errors in the shell script it is given.

You should assume as little as possible about the formatting of the shell script you are given but most of the evaluation of your program will be on sensibly formatted shell scripts. Copying the indenting will mostly but not always give you legal Python.

You should transfer any comments in the shell code. With some approaches it can be difficult to transfer comments in exactly the same position, in this case it is OK if comments are shifted to some degree.

You don't have to preserve white-space and you will not be penalized for example for removing or adding trailing white-space.

If there are shell keywords, e.g. `case` , that you cannot translate the preferred behaviour is to include the untranslated shell construct as a comment. Other sensible behaviour is acceptable.

Hints

Get the easiest transformations working first, make simplifying assumptions as needed, and get some simple small shell scripts successfully transformed. Then look at handling more constructs and removing the assumptions.

You won't be able to output Python as you generate it e.g. you won't know which import statements are needed to be printed first. Append the Python code to a list as you generate it.

If you want a good mark, you'll need to be careful in your handling of syntax which has no special meaning in shell but does in Python.

The bulk of knowledge about shell syntax & semantics you need to know has been covered in lectures. But if you want to get a high mark, you may need to discover more. Similarly much of the knowledge of Python you need has been covered but if you want to get a high mark you may need to discover more.

Python in sheepy.py

Your `sheepy.py` should work with the default Python on a CSE lab machine.

You are only permitted to import these modules in `sheepy.py`

```
argparse array atexit bisect builtins collections copy
dataclasses datetime decimal enum fileinput fnmatch fractions
functools itertools keyword locale math operator os pathlib
pprint random re statistics string sys tempfile textwrap time
traceback turtle typing unicodedata uuid
```

You are not permitted to use other modules.

For example, you are not permitted to import `shlex` in `sheepy.py` You are also not permitted to import `subprocess` in `sheepy.py` but the Python you generate can import `subprocess` .

You can request modules be added to the permitted list in the course forum.

Most of the modules listed above are little or no use for the assignment.

Three modules, `os` , `re` and `sys` are important for the assignment.

Python Translated from Shell

The Python `sheepy.py` generates from the input Shell script should work with Python on a CSE lab machine.

Any import statements should be at the top of the generated Python to ensure it is readable.

The generated Python should only import modules it uses.

The generated code is permitted only to import these modules.

```
glob os fnmatch subprocess sys stat
```

The generated Python should not generate warnings from the Python static checker `pyflakes3` . The automarking will run `pyflakes3` and if it does generate warnings there may be a small penalty. For example, `pyflakes3` will generate a warning if you have an For example unnecessary import statement and there might be a small penalty for this.

You are encouraged to generate Python which does not generate warnings from the Python style checker ' pycodestyle' but there will be no penalty if you fail to do so and some warnings, e.g. "line too long", are hard to avoid.

You'll have subtle problems with output ordering unless you use Python's `-u` flag in the '#!' line for your shell script. This is easy to do - just copy the first line of the example Python.

The '#!' line in the example generated code assumes `/usr/bin/python3` is the pathname of the Python interpreter, which it is on CSE systems and many other places.

If `/usr/bin/python3` isn't the the appropriate pathname on your computer try this '#!' line which searches all the directories in `$PATH` for the Python interpreter:.

```
#! /usr/bin/env -S python3 -u
```

Demo Shell Scripts

You should submit five shell scripts named `demo00.sh .. demo04.sh` which your program translates correctly (or at least well). These should be realistic shell scripts containing features whose successful translation indicates the performance of your assignment. Your demo scripts don't have to be original, e.g. they might be lecture examples. If they are not original they should be correctly attributed.

If you have implemented most of the subsets,r these should be longer shell scripts (20+ lines). They should if possible test many aspects of shell to Python translation.

Test Shell Scripts

You should submit five shell scripts named `test00.sh .. test04.sh` which each test a single aspect of translation. They should be short scripts containing shell code which is likely to be mis-translated. The `test??.sh` scripts do not have to be examples that your program translates 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. They should be as short as possible (even just a single line).

Change Log

Version 1.0
(2023-07-19 19:00)
- Initial release

Version 1.1
(2023-07-20 19:30)
- [Spec (subset 4)] Added `|` as a valid character to appear in `case` (and only `case`) statements

Version 1.2
(2023-07-21 18:00)
- [Spec] Added some example translations

Version 1.3
(2023-07-21 18:30)
- [Spec (subset 3)] Added the ability for `if` / `while` / `for` to be nested

Version 1.4
(2023-07-22 18:00)
- [Spec (subset 4)] Added `[` and `]` (equivalent to test)

Assessment

Testing

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

```
$ 2041 autotest sheepy
```

`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_sheepy sheepy.py demo??.sh test??.sh [any-other-files]
```

You must run `give` before **Week 11 Monday 10:00:00 AM 2023** 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_sheepy
```

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 2023** (2023-08-07 10:00:00).

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

Your assignment mark will be reduced by 0.2% for each hour (or part thereof) late past the submission deadline.

For example, if an assignment worth 60% was submitted half an hour late, it would be awarded 59.8%, whereas if it was submitted past 10 hours late, it would be awarded 57.8%.

Beware - submissions 5 or more 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:

100	All subsets working; code is beautiful; execellent test suite
HD (90+)	Subset 3 working; well-documented highable-readable code; good test suite
DN-HD (80+)	Subset 2 working; good clear code; good test suite
CR-DN (70+)	Subset 1 working; good clear code; good test suite
CR (65+)	Subset 0 working; some of subset 1 working, clear code; reasonable test suite
PS (55+)	Subset 0 passing some tests; code readability is OK; OK 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.

COMP(2041|9044) 23T2: Software Construction is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2041@cse.unsw.edu.au
CRICOS Provider 00098G