Lesson 3 - Scripting with Python 3 - Intermediate Scripting

February 14, 2020

Lesson 3: Scripting with Python 3 - Intermediate Scripting

3.1 Parsing Command Line Parameters

A lot of times, scripts become more useful if we can pass in arguments when we type the command rather than having a second step to get user input.

- 3.1.1 Accepting Simple Positional Arguments (using `sys.argv`)

Most scripts and utilities accept positional arguments instead of prompting us for input after we run the command.

The simplest way to allow our Python scripts to accept positional arguments is by using `sys.argv`.

Create a new file, name it `lab_sys_argv.py` and add the code below:

[ ]:
```
import sys

print(f"Positional arguments: {sys.argv[1:]}")
print(f"First argument: {sys.argv[1]}")
```

Let's try to run the script without passing arguments:

(command-line)

```
$ python3 lab_sys_argv.py
Positional arguments: []
Traceback (most recent call last):
  File "lab_sys_argv.py", line 4, in <module>
    print(f"First argument: {sys.argv[1]}")
IndexError: list index out of range
```

Let's try to run the script again and pass multiple arguments:

(command-line)

```
$ python3 lab_sys_argv.py arg1 arg2 arg3 'some arg'
Positional arguments: ['arg1', 'arg2', 'arg3', 'some arg']
First argument: arg1
```

This demonstrates the following about working with `argv`:

1. We risk an `IndexError` if we assume that there will be an argument for a specific position and one isn't given
2. Positional arguments are based on spaces (unless we explicitly wrap the argument in quotes)
3. We can get a slice of the first index and after without worrying about it being empty

Using sys.argv is the simplest way to allow our scripts to accept positional arguments.

- 3.1.2 More Robust CLIs (using `argparse`)

To provide a better command-line user experience, we should use something that allows us to provide contextual information and documentation.

`argparse` is standard library package that allows us to provide help text, named arguments, and flags.

— 3.1.2.1 Required Parameters and Help Flag

Let's build a CLI tool with the following specifications:

1. Requires a filename argument so it knows what file to read.
2. Print the contents of the file backward (bottom of the script first, each line printed backward)
3. Provides help text and documentation when it receives the –help flag.
4. Accepts an optional –limit or -l flag to specify how many lines to read from the file.
5. Accepts a –version flag to print out the current version of the tool.

This sounds like a lot but it's easy to do it with `argparse` module.

Create a new file, name it `lab_file_reverser.py` and add the code below:

Scenario: Without passing a required argument (filename)

[ ]:
```python
import argparse

# create an instance of ArgumentParser without any arguments
parser = argparse.ArgumentParser()

# use the add_argument method to specify a positional argument called filename␣
↪and
# provide some help text using the help argument.
parser.add_argument('filename', help='the file to read')

# tell the parser to parse the arguments from stdin using the parse_args method␣
↪and
# store the parsed arguments as to the variable args
args = parser.parse_args()
print(args)
```

Let's run the script without passing any arguments:

(command-line)

```
$ python3 lab_file_reverser.py
usage: lab_file_reverser.py [-h] filename
lab_file_reverser.py: error: the following arguments are required: filename
```

Since `filename` is required but wasn't given, the `ArgumentParser` object recognized the problem and returned a useful error message.

— 3.1.2.2 Help Flag (-h or --help) Notice that it takes the `-h` or `--help` flag already, let's try that now:

(command-line)

```
$ python3 lab_file_reverser.py -h
usage: lab_file_reverser.py [-h] filename

positional arguments:
  filename    the file to read

optional arguments:
  -h, --help  show this help message and exit
```

Scenario: Passing the required parameter (filename) Let's try running the script again but this time pass the required parameter for filename:

```
$ python3 lab_file_reverser.py xmen_base.txt
Namespace(filename='xmen_base.txt')
```

— 3.1.2.3 Optional Parameters Let's add a `--limit` flag with a `-l` alias

[ ]:
```
import argparse

parser = argparse.ArgumentParser(description='Read a file in reverse')
parser.add_argument('filename', help='the file to read')

# add a --limit flag
# - to specify that an argument is a flag, we need to place two hyphens at the␣
 ↪beginning the flag's name.
# - we specified a shorter version of the flag as our second argument
# - we used the type option for add_argument to state that we want the value␣
 ↪converted to an integer
parser.add_argument('--limit', '-l', type=int, help='the number of lines to␣
 ↪read')

args = parser.parse_args()
print(args)
```

Let's test the script:

(command-line)

```
$ python3 lab_file_reverser.py --limit 5 xmen_base.txt
Namespace(filename='xmen_base.txt', limit=5)
```

Let's add a `--version` flag with a `-v` alias

```
[ ]: import argparse

     parser = argparse.ArgumentParser(description='Read a file in reverse')
     parser.add_argument('filename', help='the file to read')
     parser.add_argument('--limit', '-l', type=int, help='the number of lines to␣
      ↪read')
     # add a --version flag
     # - use the action option to specify a string to print out when this flag is␣
      ↪received
     parser.add_argument('--version', '-v', action='version', version='%(prog)s 1.0')

     args = parser.parse_args()
     print(args)
```

Run the script:

(command-line)

```
python3 lab_file_reverser.py --version
lab_file_reverser.py 1.0
```

Notice that it executed the `version` action and didn't continue going running the script.

This uses a built-in action type of `version` which can be found on the official documentation

— 3.1.2.4 Add Business Logic

```
[ ]: import argparse

     parser = argparse.ArgumentParser(description='Read a file in reverse')
     parser.add_argument('filename', help='the file to read')
     parser.add_argument('--limit', '-l', type=int, help='the number of lines to␣
      ↪read')
     parser.add_argument('--version', '-v', action='version', version='%(prog)s 1.0')

     args = parser.parse_args()

     # add the actual business logic for reversing the contents of the file
     with open(args.filename) as f:
         lines = f.readlines()
         lines.reverse()

         if args.limit:
```

```
        lines = lines[:args.limit]

    for line in lines:
        print(line.strip()[::-1])
```

Let's run the script (without passing a limit flag)

(command-line)

```
$ python3 lab_file_reverser.py xmen_base.txt
gnihtemoS
reivaX rosseforPrelwarcthgiN
pohsiB
spolcyC
enirevloW
mrotS
```

Let's run the script (passing a limit flag)

(command-line)

```
$ python3 lab_file_reverser.py -l 3 xmen_base.txt
gnihtemoS
reivaX rosseforPrelwarcthgiN
pohsiB
```

3.3 Error Handling with try/except/else/finally

In our `lab_file_reverser.py` script, what happens if the filename doesn't exist?

(command-line)

```
$ python3 lab_file_reverser.py somefile.txt
Traceback (most recent call last):
  File "lab_file_reverser.py", line 19, in <module>
    with open(args.filename) as f:
FileNotFoundError: [Errno 2] No such file or directory: 'somefile.txt'
```

This `FileNotFoundError` is something that we can expect to happen often.

Our script should be able handle this scenario.

To handle these errors, let's utilize the keywords `try`, `except`, and `else`.

```
[ ]: import argparse

     parser = argparse.ArgumentParser(description='Read a file in reverse')
     parser.add_argument('filename', help='the file to read')
     parser.add_argument('--limit', '-l', type=int, help='the number of lines to␣
      ↪read')
     parser.add_argument('--version', '-v', action='version', version='%(prog)s 1.0')
```

```python
args = parser.parse_args()

# We utilize the try statement to denote that it's quite possible for an error to happen within the try block
try:
    f = open(args.filename)
    limit = args.limit
# We can handle specific types of errors using the except keyword (we can have more than one).
except FileNotFoundError as err:
    print(f"Error: {err}")
# If there isn't an error, we want to carry out the code that is in the else block
else:
    with f:
        lines = f.readlines()
        lines.reverse()

        if args.limit:
            lines = lines[:args.limit]

        for line in lines:
            print(line.strip()[::-1])

# If we want to execute some code regardless of there being an error or not,
# we can put that in a finally block at the very end of our try/except code
```

Now when we try to run our script (passing a non-existent file), we get a better and much more informative response:

```
$ python3 lab_file_reverser.py somefile.txt
Error: [Errno 2] No such file or directory: 'somefile.txt'
```

3.4 Shell Command Exit Statuses

When we're writing scripts, we want to be able to set exit statuses if something goes wrong.

For that, we can use the `sys` module.

When our `lab_file_reverser.py` script receives a file that doesn't exist, we show an error message.

However, we don't set the exit status to `1` to indicate an error:

(command-line)

```
$ python3 lab_file_reverser.py somefile.txt
Error: [Errno 2] No such file or directory: 'somefile.txt'
$ echo $?
0
```

Let's modify our script:

```
[ ]: import argparse
     # add import for sys module
     import sys

     parser = argparse.ArgumentParser(description='Read a file in reverse')
     parser.add_argument('filename', help='the file to read')
     parser.add_argument('--limit', '-l', type=int, help='the number of lines to␣
      ↪read')
     parser.add_argument('--version', '-v', action='version', version='%(prog)s 1.0')

     args = parser.parse_args()

     try:
         f = open(args.filename)
         limit = args.limit
     except FileNotFoundError as err:
         print(f"Error: {err}")
         # set exit status to 1 to indicate error
         sys.exit(1)
     else:
         with f:
             lines = f.readlines()
             lines.reverse()

             if args.limit:
                 lines = lines[:args.limit]

             for line in lines:
                 print(line.strip()[::-1])
```

Now, if we try to run our script passing a non-existent file, we will exit with the proper code:

(command-line)

```
$ python3 lab_file_reverser.py somefile.txt
Error: [Errno 2] No such file or directory: 'somefile.txt'
$ echo $?
1
```

3.4 Execute Shell Commands from Python (using `subprocess`)

- 3.4.1 Executing Shell Commands (using `subprocess.run`)

We can use the `subprocess` module for working with external processes

The `subprocess.run` function provides us with a lot of flexibility:

(REPL)

```
>>> import subprocess
>>> proc = subprocess.run(['ls', '-l'])
total 44
-rw-rw-r-- 1 micaela micaela  230 Jan 14 21:26 exercise1.py
-rw-rw-r-- 1 micaela micaela  458 Jan 15 14:59 exercise2.py
-rw-rw-r-- 1 micaela micaela  570 Jan 15 16:32 exercise3.py
-rw-rw-r-- 1 micaela micaela  223 Jan 17 16:47 lab_age.py
-rw-rw-r-- 1 micaela micaela  907 Jan 28 21:31 lab_bmi.py
-rw-rw-r-- 1 micaela micaela  173 Jan 29 02:46 lab_check_stage.py
-rw-rw-r-- 1 micaela micaela 1488 Jan 31 18:14 lab_file_reverser.py
-rw-rw-r-- 1 micaela micaela   98 Jan 31 01:39 lab_sys_argv.py
-rw-rw-r-- 1 micaela micaela   43 Jan 29 15:43 new_xmen.txt
-rw-rw-r-- 1 micaela micaela  258 Jan 29 16:09 references.txt
-rw-rw-r-- 1 micaela micaela   70 Jan 29 16:04 xmen_base.txt
>>> proc
CompletedProcess(args=['ls', '-l'], returncode=0)
>>>
```

Notice the following: - our `proc` variable is a `CompletedProcess` object. - our `proc` variable gives us access to the `returncode` attribute which we can use to ensure that the process succeeded and returned `0`. - the `ls` command was executed and printed to the screen without us specifying to print anything.

- 3.4.2 Capturing STDOUT (using `subprocess.PIPE`)

We can capture the STDOUT using `subprocess.PIPE`

(REPL)

```
>>> proc = subprocess.run(
...     ['ls', '-l'],
...     stdout=subprocess.PIPE,
...     stderr=subprocess.PIPE,
... )
>>> proc
CompletedProcess(args=['ls', '-l'], returncode=0, stdout=b'total 44\n-rw-rw-r-- 1 micaela mica
>>>
>>> proc.stdout
b'total 44\n-rw-rw-r-- 1 micaela micaela  230 Jan 14 21:26 exercise1.py\n-rw-rw-r-- 1 micaela m
>>>
>>> proc.stderr
b''
>>>
```

- 3.4.3 Handling a `bytes` object

Notice that the output `proc.stdout` and `proc.stdout` is prefixed with a `b` character.

This indicates that it's a `bytes` object and not a string.

The `bytes` type can only contain ASCII characters and doesn't do anything special with escape sequences when printed.

We can use the `bytes.decode()` method if we want to utilize the bytes object as a string:

(REPL)

```
>>> print(proc.stdout)
b'total 44\n-rw-rw-r-- 1 micaela micaela  230 Jan 14 21:26 exercise1.py\n-rw-rw-r-- 1 micaela 
>>>
>>> print(proc.stdout.decode())
total 44
-rw-rw-r-- 1 micaela micaela  230 Jan 14 21:26 exercise1.py
-rw-rw-r-- 1 micaela micaela  458 Jan 15 14:59 exercise2.py
-rw-rw-r-- 1 micaela micaela  570 Jan 15 16:32 exercise3.py
-rw-rw-r-- 1 micaela micaela  223 Jan 17 16:47 lab_age.py
-rw-rw-r-- 1 micaela micaela  907 Jan 28 21:31 lab_bmi.py
-rw-rw-r-- 1 micaela micaela  173 Jan 29 02:46 lab_check_stage.py
-rw-rw-r-- 1 micaela micaela 1488 Jan 31 18:14 lab_file_reverser.py
-rw-rw-r-- 1 micaela micaela   98 Jan 31 01:39 lab_sys_argv.py
-rw-rw-r-- 1 micaela micaela   43 Jan 29 15:43 new_xmen.txt
-rw-rw-r-- 1 micaela micaela  258 Jan 29 16:09 references.txt
-rw-rw-r-- 1 micaela micaela   70 Jan 29 16:04 xmen_base.txt

>>>
```

- 3.4.4 Intentionally Raising Errors

By default, the `subprocess.run` function doesn't raise an error when you execute something that returns a non-zero exit status:

(REPL)

```
>>> proc = subprocess.run(['cat', 'somefile.txt'])
cat: somefile.txt: No such file or directory
>>> proc
CompletedProcess(args=['cat', 'somefile.txt'], returncode=1)
>>>
```

In this scenario, we might want to raise an error.

We can pass the `check` argument to the function so it raises a `subprocess.CalledProcessError` if something goes wrong.

(REPL)

```
>>> error_proc = subprocess.run(['cat', 'somefile.txt'], check=True)
cat: somefile.txt: No such file or directory
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.6/subprocess.py", line 438, in run
    output=stdout, stderr=stderr)
subprocess.CalledProcessError: Command '['cat', 'somefile.txt']' returned non-zero exit status
>>>
```

3.5 Advanced Iteration (using List Comprehension)

It's very likely that we're going to need to work with large amounts of data and we'll probably need to manipulate lists to do the following:

- Filter out items in that list
- Modify every item in the list

Create a new file named `lab_search_words.py` and add the following lines:

```
[ ]: import argparse

parser = argparse.ArgumentParser(description='Search for words including␣
 ↪partial word')
parser.add_argument('snippet', help='partial (or complete) string to search for␣
 ↪in words')

args = parser.parse_args()
snippet = args.snippet.lower()

with open('/usr/share/dict/words') as f:
  words = f.readlines()

matches = []

for word in words:
   if snippet in word.lower():
       matches.append(word)

print(matches)
```

Let's try to run the script:

(command-line)

```
$ python3 lab_search_words.py hello
['Othello\n', "Othello's\n", 'hello\n', "hello's\n", 'hellos\n']
```

We can replace the loop part of the code above to utilize list comprehension.

So instead of 8 lines of code, it will be replaced by 2 lines of code:

Original chunk of code:

```
[ ]: words = open('/usr/share/dict/words').readlines()
     matches = []

     for word in words:
         if snippet in word.lower():
             matches.append(word)

     print(matches)
```

New code that utilizes list comprehension:

```
[ ]: words = open('/usr/share/dict/words').readlines()
     print([word for word in words if snippet in word.lower()])
```

Full Script:

```
[ ]: import argparse

     parser = argparse.ArgumentParser(description='Search for words including␣
      ↪partial word')
     parser.add_argument('snippet', help='partial (or complete) string to search for␣
      ↪in words')

     args = parser.parse_args()
     snippet = args.snippet.lower()

     words = open('/usr/share/dict/words').readlines()
     print([word for word in words if snippet in word.lower()])
```

Let's try to run the script again:

(command-line)

```
$ python3 lab_search_words.py hello
['Othello\n', "Othello's\n", 'hello\n', "hello's\n", 'hellos\n']
```

Notice that the output is the same except that out code is much more concise.