Command Line Programs

A Key To Automation

Much of the power of software comes from its automation.

A good command line interface makes the software you write more automatable. This ALWAYS increases the value of your program.

```
find . -name __pycache__ -type d
ls -l
git commit -m "Fix for #742"
grep -i 'python' *.txt
grep '^WARNING:' logs/*.log
```

sys.argv

One approach: use `sys.argv`.

```
# findpattern.py
import sys
def grepfile(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')

if __name__ == "__main__":
    pattern, path = sys.argv[1], sys.argv[2]
    for line in grepfile(pattern, path):
        print(line)
```

```
$ python3 findpattern.py  'WARNING' webserver.log
```

This works. But it makes you do the heavy lifting.

sys.argv

Also, the error messages are weird:

```
$ python3 findpattern.py
Traceback (most recent call last):
  File "findpattern.py", line 10, in <module>
    pattern, path = sys.argv[1], sys.argv[2]
IndexError: list index out of range
```

Optional Arguments

If you have multiple optional arguments, it gets really complex:

- `-i` for case-insensitive search
- `-c` for line count (instead of printing matching lines)
- Detect their presence in any order

```
$ python3 findpattern.py -i 'WARNING' webserver.log -c
```

The complexity explodes. We need a better approach.

argparse

The modern tool for building a command line interface is `argparse`.

- Easily specify complex sets of required and optional arguments
- Uses declarative syntax. Python handles the logic for you
- Boolean flags and other type conversions
- Automatic, user-friendly online help
- Clear and understandable error messages

```
import argparse
parser = argparse.ArgumentParser()
```

Using argparse

```
# findpattern.py, version 2:
# Use the same grepfile() function.

import argparse # instead of "import sys"

parser = argparse.ArgumentParser()
parser.add_argument('pattern')
parser.add_argument('path')

if __name__ == "__main__":
    args = parser.parse_args()
    for line in grepfile(args.pattern, args.path):
        print(line)
```

Parsing Args

`parse_args()` returns the parsed arguments:

```
>>> from findpattern import parser
>>> args = parser.parse_args(["WARNING", "webserver.log"])
>>> args
Namespace(path='webserver.log', pattern='WARNING')
>>> args.pattern
'WARNING'
>>> args.path
'webserver.log'
```

`parse_args()` operates on `sys.argv` by default:

```
# These two are equivalent.
parser.parse_args()
parser.parse_args(sys.argv[1:])
```

Helpful Errors

`argparse` includes automatic, user-friendly validation:

```
$ python3 findpattern.py
usage: findpattern.py [-h] pattern path
findpattern.py: error: the following arguments are required: pattern, path
```

And look, there's a -h option!

```
$ python3 findpattern.py  -h
usage: findpattern.py [-h] pattern path

positional arguments:
  pattern
  path

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

More Help

```
parser = argparse.ArgumentParser(description='Find patterns in file')
parser.add_argument('pattern', help='Substring pattern')
parser.add_argument('path', help='File to search in')
```

```
$ python3 findpattern.py -h
usage: findpattern.py [-h] pattern path

Find patterns in file

positional arguments:
  pattern     Substring pattern
  path        File to search in

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

Similar to grep, but with substring matching instead of regexes.
```

Boolean Flags

```
parser.add_argument('pattern', help='Substring pattern')
parser.add_argument('path', help='File to search in')
parser.add_argument('-i', '--ignore-case', default=False,
action='store_true')
```

```
>>> args = parser.parse_args(["-i", "WARNING", "webserver.log"])
>>> args.ignore_case
True

>>> args = parser.parse_args(["--ignore-case", "WARNING", "webserver.log"])
>>> args.ignore_case
True

>>> args = parser.parse_args(["WARNING", "webserver.log"])
>>> args.ignore_case
False
```

You can also do `default=True` and `action="store_false"`.

Typed Arguments

By default, arguments are parsed as strings. But you can specify a type.

```
parser.add_argument('--limit', default=None, type=int,
    help='Show only this many matches. Default is show all')
```

```
>>> args = parser.parse_args(["--limit", "42", "WARNING", "webserver.log"])
>>> print(args.limit)
42
>>> type(args.limit)
<class 'int'>

>>> args = parser.parse_args(["WARNING", "webserver.log"])
>>> print(args.limit)
None
```

```
$ python3 findpattern.py --limit notanumber WARNING webserver.log
usage: findpattern.py [-h] [-i] [--limit LIMIT] pattern path
findpattern.py: error: argument --limit: invalid int value: 'notanumber'
```

Custom Validation

```
parser = argparse.ArgumentParser()
parser.add_argument('-c', '--count', default=False, action='store_true',
                    help='Show # of matches instead of matching lines')
parser.add_argument('--limit', default=None, type=int,
                    help='Show only this many matches. Default is show all')
args = parser.parse_args()
if args.count and args.limit:
    parser.error('Cannot combine --limit and --count options')
```

```
$ python3 findpattern.py --limit 7 -c WARNING webserver.log
usage: findpattern.py [-h] [-i] [-c] [--limit LIMIT] pattern path
findpattern.py: error: Cannot combine --limit and --count options
```

Encapsulating

If your parser gets a little complex, encapsulate in a function.

```
import argparse
def get_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('pattern', help='Substring pattern')
    parser.add_argument('path', help='File to search in')
    parser.add_argument('-i', '--ignore-case', default=False,
                        action='store_true')
    parser.add_argument('-c', '--count', default=False, action='store_true',
                        help='Show # of matches instead of matching lines')
    parser.add_argument('--limit', default=None, type=int,
        help='Show only this many matches. Default is show all')
    args = parser.parse_args()
    if args.count and args.limit:
        parser.error('Cannot combine --limit and --count options')
    return args

if __name__ == "__main__":
    args = get_args()
```

Better UX

```
if __name__ == "__main__":
    args = get_args()
```

For a better user experience, parse the args immediately inside the main block.

Quicker feedback from `--help`, error checking, etc.

Lab: Command Line Arguments

Lab file: `commandline/argparselab.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up.

Reference:

https://docs.python.org/3/library/argparse.html

(Link in top of your lab file, for easy copy-pasting)