Systems programming with bash and Python 3

On 30 April 2020, I had a chance to talk with Dustin Reybrouck for The SysAdminShow Podcast. We mostly discussed why sysadmins might like to add Python as a tool in addition to shell scripting. Most of his audience is probably familiar with `bash` or Powershell, so I showed how I might write somewhat simple but parameterized and documented version of `head` in `bash` and how it would be translated to Python 3. Following is a summary of our discussion.

Sysadmins spend much of their lives on the command-line, so it makes sense that they would seek to automate tasks using the language of their command line — something like `bash` or Powershell. While it is possible to write many useful programs in these languages, a higher-level language like Python might prove to be a better choice especially given that a Python program is portable between systems that natively understand `bash` (e.g., Linux or Mac) and Powershell (Window). As an exercise, let's write a `bash` implementation of the `head` command, then we'll compare how we could write it in Python.

Writing `head` in `bash`

Let's start by imagining how we might write our own implementation of the venerable `head` command. For example, given the text of the US Constitution, we would expect to see the first few lines of a given file, usually 10:

```
$ head ~/work/tiny_python_projects/inputs/const.txt
We the People of the United States, in Order to form a more perfect Union,
establish Justice, insure domestic Tranquility, provide for the common
defence, promote the general Welfare, and secure the Blessings of Liberty to
ourselves and our Posterity, do ordain and establish this Constitution for the
United States of America.

Article 1.

Section 1
All legislative Powers herein granted shall be vested in a Congress of the
```

And we would expect to be able to modify that number using an option like `-n`:

```
$ head -n 3 const.txt
We the People of the United States, in Order to form a more perfect Union,
establish Justice, insure domestic Tranquility, provide for the common
defence, promote the general Welfare, and secure the Blessings of Liberty to
```

It’s commonplace for command-line tools to respond to `-h` or `--help` with a "usage" statement about how the program should be invoked. In the case of `head` [1], it does not give a usage because we ask for it but rather because it does not recognize these as valid options. Still, it manages to produce a "usage" under some circumstances which is better than nothing:

```
$ head -h
head: illegal option -- h
usage: head [-n lines | -c bytes] [file ...]
```

A `bash` version that handles one file

Let’s start off with a version in `bash` that handles just one file and a possible number of lines which will default to 10. If run with no arguments, it will print a "usage" statement:

```
$ ./simple-head.sh
Usage: simple-head.sh FILE [NUM]
```

When run with a file as the only argument, it will print the first 10 lines:

```
$ ./simple-head.sh const.txt
We the People of the United States, in Order to form a more perfect Union,
establish Justice, insure domestic Tranquility, provide for the common
defence, promote the general Welfare, and secure the Blessings of Liberty to
ourselves and our Posterity, do ordain and establish this Constitution for the
United States of America.

Article 1.

Section 1
All legislative Powers herein granted shall be vested in a Congress of the
```

We can provide an optional second argument to change the number of lines we show:

```
$ ./simple-head.sh const.txt 2
We the People of the United States, in Order to form a more perfect Union,
establish Justice, insure domestic Tranquility, provide for the common
```

Note that the program fails rather gracelessly. For instance, if we give it a non-existent file:

```
$ ./simple-head.sh foo
./simple-head.sh: line 24: foo: No such file or directory
```

Our program also fails to show an error if the second argument is not a number and will, in fact, print the entire file. Try running the program like so:

```
$ ./simple-head.sh const.txt foo
```

Still, it’s instructive to look at this program:

```
#!/bin/bash ①

#
# Author : Ken Youens-Clark <kyclark@gmail.com> ②
# Purpose: simple bash implementation of `head`
#

# Check number of arguments is 1 or 2
if [[ $# -lt 1 ]] || [[ $# -gt 2 ]]; then       ③
    echo "Usage: $(basename "$0") FILE [NUM]"   ④
    exit 1                                      ⑤
fi

FILE=$1       ⑥
NUM=${2:-10}  ⑦
LINE_NUM=0    ⑧

while read -r LINE; do               ⑨
    echo "$LINE"                     ⑩
    LINE_NUM=$((LINE_NUM+1))         ⑪
    if [[ $LINE_NUM -eq $NUM ]]; then ⑫
        break                        ⑬
    fi
done < "$FILE"
```

① This line is often called the "shebang," and it is common to see the path to bash hard-coded like this. It’s not necessarily best practice, however, as bash might well be located at /usr/local/bin/bash.

② Any text following # is ignored by bash. Here we add comments to the program, but you can also use this to temporarily disable code. It’s polite to document your code so that other might contact you with questions.

③ Everything in bash is a string, but we can use operators like -lt (less than) and -gt (greater than to get numeric comparisons. The $# variable holds the number of arguments to our program, so we’re trying to see if we do not have exactly 1 or 2 arguments.

④ We print a "usage"-type statement to explain to the user how to invoke the program. The FILE is a required position argument while the [NUM] is shown in [] to indicate that it is optional.

⑤ We exit with a non-zero value (1 is fine) to indicate that the program failed to run as expected.

⑥ Since we know we have at least 1 argument, we can copy the value of the first argument in $1 to our FILE variable.

⑦ We may or may not have a second argument, so we can either copy $2 or a default value of 10 to our NUM variable.

⑧ Initialize a LINE_NUM variable to 0 so we can count how many lines of our file we have shown.

⑨ A while loop is a common idiom for reading a file line-by-line in bash.

⑩ The echo command will print text to the terminal.

⑪ The $(()) evaluation will allow us to perform a bit of arithmetic with what is otherwise a string value. Here we want to add 1 to the value of LINE_NUM.

⑫ The -eq is the numeric equality operator in bash. Here we check if the LINE_NUM is equal to the number of lines we mean to show.

⑬ The break statement will cause the while loop to exit.

A complete implementation in bash

The previous simple-head.sh version shows some basic ideas of how to handle many systems-level tasks in bash such as:

- Documenting the language of the program with a shebang line
- Documenting the author and purpose program with comments
- Parameterizing your program so as to values as arguments rather than hard-coding values
- Documenting the program parameters with an automatically generated "usage" when needed by the user
- Exiting the program with non-zero values when the program does not complete as normally expected
- Defining reasonable default values for optional arguments

Still, this is a rather sophomoric replacement for head because:

- It does not handle multiple files
- It fails to validate if the arguments are actually readable files
- There is no -n option because the program handles only *positional* arguments and so cannot handle *options*
- The program will not print a "usage" for -h, again because it fails to handle options

Let’s write a better implementation that is a complete replacement for head:

```
#!/usr/bin/env bash ①

#
# Author : Ken Youens-Clark <kyclark@gmail.com> ②
# Purpose: bash implementation of `head`
#

# Die on use of uninitialize variables
set -u ③
```

```
# Default value for the argument
NUM_LINES=10 ④

# A function to print the "usage"
function USAGE() { ⑤
    printf "Usage:\n  %s -n NUM_LINES [FILE ...]\n\n" "$(basename "$0")"

    echo "Options:"
    echo " -n NUM_LINES"
    echo
    exit "${1:-0}"
}

# Die if we have no arguments at all
[[ $# -eq 0 ]] && USAGE 1 ⑥

# Process command line options
while getopts :n:h OPT; do ⑦
    case $OPT in           ⑧
        n)
            NUM_LINES="$OPTARG" ⑨
            shift 2             ⑩
            ;;
        h)
            USAGE               ⑪
            ;;
        :)
            echo "Error: Option -$OPTARG requires an argument." ⑫
            exit 1
            ;;
        \?)
            echo "Error: Invalid option: -${OPTARG:-""}" ⑬
            exit 1
    esac
done

# Verify that NUM_LINES looks like a positive integer
if [[ $NUM_LINES -lt 1 ]]; then            ⑭
    echo "-n \"${NUM_LINES}\" must be > 0"
    exit 1
fi

# Process the positional arguments
FNUM=0                ⑮
for FILE in "$@"; do  ⑯
    FNUM=$((FNUM+1))  ⑰

    # Verify this argument is a readable file
    if [[ ! -f "$FILE" ]] || [[ ! -r "$FILE" ]]; then ⑱
        echo "\"${FILE}\" is not a readable file"
        continue ⑲
    fi

    # Print a header in case of mulitiple files
    [[ $# -gt 1 ]] && echo "==> ${FILE} <==" ⑳

    # Initialize a counter variable
    LINE_NUM=0

    # Loop through each line of the file
    while read -r LINE; do
        echo $LINE

        # Increment the counter and see if it's time to break
        LINE_NUM=$((LINE_NUM+1))
        [[ $LINE_NUM -eq $NUM_LINES ]] && break
    done < "$FILE"

    [[ $# -gt 1 ]] && [[ $FNUM -lt $# ]] && echo
done

exit 0
```

① Using the `env` program (which is pretty universally located at `/usr/bin/env`) to find `bash` is more flexible than hard-coding the path as `/bin/bash`.

② Same documentation as comments.

③ This will cause `bash` to die if we attempt to use an uninitialized variable and is one of the few safety features offered by the language.

④ Here we set a default value for the `NUM_LINES` to show which can be overridden by an option.

⑤ Since there are a multiple times I might want to show the usage and exit with an error (e.g., no arguments or as requested by `-h`), I can put this into a `function` to call later.

⑥ If the number of arguments to the program `$#` is 0, then exit with a "usage" statement and a non-zero value.

⑦ We can use `getopts` in `bash` to manually parse the command-line arguments. We are specifically looking for flags `-n` which takes a value and `-h` which does not.

⑧ `$OPT` will have the flag value such as `n` for `-n` or `h` for `-h`.

⑨ The `$OPTARG` will have the value for the `-n` flag. We can copy that to our `NUM_LINES` variable to save it.

⑩ Now that we have processed `-n 3`, for instance, we use `shift 2` to discard those two values from the program arguments `$@`.

⑪ If processing the `-h` flag, call the `USAGE` function which will cause the program to exit.

⑫ This handles when an option like `-n` does not have an accompanying value.

⑬ This handles an option we didn’t define.

⑭ This use the `-lt` operator to coerce the `NUM_LINES` to a numeric value. If it is less than `-lt` 1, we throw an error and exit with a non-zero value.

⑮ Now that we have handled the optional arguments, we can handle the rest of the *positional* arguments found in `$@`. We start off by defining a `FNUM` so we can track the file number we are working with. That is, this is the index value of the current file.

⑯ We can use a `for` loop to iterate through the positional arguments found in `$@`.

⑰ Add 1 to the `FNUM` variable.

⑱ The `-f` test will return a "true" value if the given argument is a file, and `!` will negate this. Ditto as `-r` will report if the argument is a readable file.

⑲ The `continue` statement will cause the `for` loop to immediately advance to the next iteration, skipping all the code below.

⑳ If the number of positional arguments is greater than `-gt` 1, then print a header showing the current file's name.

Initialize a line count variable for reading the file.

This is the same loop as before that we used to read a given number of lines from the file. This one is improved, however, because we check if the number argument from the user is actually a positive integer!

This is a shorter way to write a single-line `if` statement.

If there are multiple files to process and we're not currently on the last file, then print an extra newline to separate the outputs.

If you are new to `bash` programming, the syntax will probably look rather cryptic! The entirely manual handling of the command-line options and positional arguments is especially cumbersome. I will admit this is not an easy program to write correctly, and, even when it finally works on my Linux and Max machines, I won't be able to give it to a Windows user unless they have something like WSL (Windows Subsystem for Linux) or Cygwin installed.

Still, this program works rather well! It will print nice documentation if we run with no arguments or if you run `./head.sh -h`, which is actually an improvement over `head`:

```
$ ./head.sh
Usage:
  head.sh -n NUM_LINES [FILE ...]

Options:
 -n NUM_LINES
```

It rejects bad options:

```
$ ./head.sh -x 8 const.txt
Error: Invalid option: -x
```

It can handle both options and positional arguments, provides a reasonable default for the `-n` option, and correctly skips non-file arguments:

```
$ ./head.sh -n 3 foo const.txt
"foo" is not a readable file
==> const.txt <==
We the People of the United States, in Order to form a more perfect Union,
establish Justice, insure domestic Tranquility, provide for the common
defence, promote the general Welfare, and secure the Blessings of Liberty to
```

And it mimics the output from `head` for multiple files:

```
$ ./head.sh -n 1 const.txt simple-head.sh head.sh
==> const.txt <==
We the People of the United States, in Order to form a more perfect Union,

==> simple-head.sh <==
#!/bin/bash

==> head.sh <==
#!/usr/bin/env bash
```

For what it's worth, I used the included `new_bash.py` program to create this program. If you find yourself stuck writing a `bash` program and don't wish to start from scratch, this program might be useful to you.

Testing `head.sh`

I have included a `test.py` that is a Python program that will run the `head.sh` program to ensure it actually does what it is supposed to do. If you look at the contents of this program, you will see a number of functions with names that start with `test_`. This is because I use the `pytest` module/program to run these functions as a test suite. I like to use the `-x` flag to indicate that testing should halt at the first failing test and the `-v` flag for "verbose" output. These can be specified individually or combined like `-xv` or `-vx`:

```
$ pytest -xv test.py
============================= test session starts ==============================
...

test.py::test_exists PASSED                                              [ 14%]
test.py::test_usage PASSED                                               [ 28%]
test.py::test_bad_file PASSED                                            [ 42%]
test.py::test_bad_num PASSED                                             [ 57%]
test.py::test_default PASSED                                             [ 71%]
test.py::test_n PASSED                                                   [ 85%]
test.py::test_multiple_files PASSED                                      [100%]

============================== 7 passed in 0.56s ===============================
```

It’s a bit of a nuisance to have to write the tests for a program in a different language from the program itself, but I know of no testing framework in `bash` that I’d could use (or would like to learn) that can run a test suite such as the above!

Writing `head.py` in Python 3

To write a similar version in Python, we’ll rely heavily on the standard `argparse` module to handle the validation of all the command-line arguments as well as generating the "usage" statements. Here is a version that, similar to the `simple-head.py`, will handle just one file:

```
#!/usr/bin/env python3 ①
"""                    ②
Author : Ken Youens-Clark
Purpose: Python implementation of head
         This version only handles one file!
"""

import argparse        ③
import os
import sys


# --------------------------------------------------
def get_args():        ④
    """Get command-line arguments"""  ⑤

    parser = argparse.ArgumentParser( ⑥
        description='Python implementation of head',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',       ⑦
                        metavar='FILE',
                        type=argparse.FileType('rt'), ⑧
                        help='Input file')

    parser.add_argument('-n',         ⑨
                        '--num',
                        help='Number of lines',
                        metavar='int',
                        type=int,     ⑩
                        default=10)   ⑪

    args = parser.parse_args()        ⑫

    if args.num < 1:                  ⑬
        parser.error(f'--num "{args.num}" must be > 0') ⑭

    return args                       ⑮
```

```
# --------------------------------------------------
def main():                             ⑯
    """Make a jazz noise here"""

    args = get_args()                   ⑰

    for i, line in enumerate(args.file, start=1): ⑱
        print(line, end='') ⑲
        if i == args.num:   ⑳
            break


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

① The "shebang" uses the `env` program to find the first `python3` in our `$PATH`.

② The triple quotes allow us to create a string that spans multiple lines. Here we're creating a string but not assigning it to a variable. This is a convention for creating documentation also called a "docstring." This docstring summarizes the program itself. I like to document at least who wrote it and why.

③ We can `import` code from other modules. While we can `import` several modules separated by commas, it's recommended to put each on a separate line. Specifically we want to use `argparse` to handle the command-line arguments, and we'll also use the `os` (operating system) and `sys` (systems) modules.

④ I like to always define a `get_args()` function that exclusively deals with `argparse` for creating the program's parameters and validating the arguments. I always place this first so I can see it immediately when I'm reading the program.

⑤ This is a docstring for the function. It's ignored like a comment would be, but it has significance to Python and would appear if I were to `import` this module and ask for `help(get_args)`.

⑥ This creates a `parser` that will handle the command-line arguments. I add a description for the program that will appear in any "usage" statements, and I always like to have `argparse` display any default values for the user.

⑦ Positional arguments have no leading dashes in their names. Here we define a single positional argument that we can refer to internally as `file`.

⑧ The default `type` for all arguments is a `str` (string). We can ask `argparse` to enforce a different type like `int` and it will print an error when the user fails to provide a value that can be parsed into an integer value. Here we are using the special `argparse` type that defined a "readable" (`'r'`) "text" (`'t'`) file. If the user provides anything other than a readable text file, `argparse` will halt the program, print an error and usage, and exit with a non-zero value.

⑨ The leading `-` on `-n` (short name) and `--num` (long name) for the "number" argument means this will be an *option*.

⑩ The user must provide a value that can be parsed into a `int` value.

⑪ The default value will be 10.

⑫ After defining the program's parameters, we ask the `parser` to parse the arguments. If there are any problems like the wrong number or types of arguments, `argparse` will stop the program here.

⑬ If we get to this point, the arguments were valid as far as `argparse` is concerned. We can perform additional manual checks such as verifying that `args.num` is greater than 0.

⑭ The `parser.error()` function is a way for us to manually invoke the error-out function of `argparse`.

⑮ Functions in Python must explicitly `return` a value or the `None` will be returned by default. Here was want to return the `args` to the calling function.

⑯ Convention dictates the starting function be called `main()`, but this is not a requirement, and Python will not automatically call this function to start the program. Neither `get_args()` nor `main()` accept arguments, but, if they did, they would be listed in the parens.

⑰ All the work to define the parameters, validate the arguments, and handle help and usage has now been hidden in the `get_args()` function. We can think of this as a "unit" that encapsulates those ideas. If our program successfully calls `get_args()` and returns with some `args`, then we can move forward knowing the arguments are actually correct and useful.

⑱ We don't have to initialize a counting variable like in `bash` as we can use the `enumerate()` function to return the index and value of any sequence of items. Here the `args.file` is actually an *open file handle* provided by `argparse` because we defined the `args.file` as a "file" type. I can use the `start` option to `enumerate()` to start counting at 1 instead of 0.

⑲ The `print()` function is like the `echo` statement in `bash`. Here there will be a newline stuck to the `line` from the file, so I use the `end=''` to indicate that `print()` should not add the customary newline to the output.

⑳ While `bash` uses `-eq` for numeric comparison and `==` for string equality, Python uses `==` for both.

Both Python and `bash` use `continue` and `break` in loops to skip and leave loops, respectively.

This is the idiom in Python to detect when a program/module is being run from the command line. Here we want to execute the `main()` function to start the program to running.

The above program has been contributed as `py-head/solution1.py`, and you can run it to see how it will create usage for no arguments:

```
$ ./solution1.py
usage: solution1.py [-h] [-n int] FILE
solution1.py: error: the following arguments are required: FILE
```

Note that we did not define the `-h` and `--help` flags to `argparse` as those are reserved specifically for generating help:

```
$ ./solution1.py -h
usage: solution1.py [-h] [-n int] FILE

Python implementation of head

positional arguments:
  FILE                  Input file

optional arguments:
  -h, --help            show this help message and exit
  -n int, --num int     Number of lines (default: 10)
```

Note that `argparse` can actually handle options following positional arguments:

```
$ ./solution1.py const.txt -n 3
We the People of the United States, in Order to form a more perfect Union,
establish Justice, insure domestic Tranquility, provide for the common
defence, promote the general Welfare, and secure the Blessings of Liberty to
```

A complete implementation in Python

The previous Python version demonstrates many shortcuts to creating usable and documented programs that are easier to write and more reliable than the `bash` version. Still, this program is not yet a full replacement either for `head` or even `head.sh`. Let's see how we can expand the program to handle *one or more* positional arguments:

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark
Purpose: Python implementation of head
         This version handles multiple files
         and is very similar to the bash version.
"""

import argparse


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Python implementation of head',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        nargs='+', ①
                        help='Input file')

    parser.add_argument('-n',
                        '--num',
                        help='Number of lines',
                        metavar='int',
                        type=int,
                        default=10)

    args = parser.parse_args()

    if args.num < 1:
        parser.error(f'--num "{args.num}" must be > 0')

    return args


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    num_files = len(args.file) ②

    for fnum, fh in enumerate(args.file, start=1):    ③
        if num_files > 1:                             ④
            print(f'==> {fh.name} <==')               ⑤

        for line_num, line in enumerate(fh, start=1): ⑥
            print(line, end='')
            if line_num == args.num:
                break


        if num_files > 1 and fnum < num_files:        ⑦
            print()


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

① The `nargs` option can take `*` to indicate zero or more, `+` for one or more, and `?` for zero or one.

② Since I'll refer to the number of files (`num_files`) several times, I put them into a variable. The `args.file` argument is now a `list` of open file handles. I can use the `len()` function to ask the *length* of this `list` which will tell me the number of files provided as arguments. I know they are actually readable text files because of the `type` constraint I added to this argument.

③ Again I want both the index (position) and value of each element in `args.file`, so I can use `enumerate()`, starting the counting at 1 instead of 0.

④ Decide whether to print a header.

⑤ I call the variable `fh` to remind me that this is an open file handle. I can get the name of the file itself using `fh.name`. The `f''` (f-string) allows me to interpolate the `fh.name` value inside the string given to `print()`.

⑥ A second loop to iterate over the lines in the file. This is the same code as above.

⑦ Decide whether to `print()` an extra newline between multiple files.

This version is a full implementation of a typical `head` program and demonstrates many common systems-level programming concepts a sysadmin might need. The `os` and `sys` modules are particularly rich in functions for dealing with files and directories and permissions and the like. The `argparse` code allows one to outsource program validation to another module allowing the coder to focus on the tasks at hand rather than the implementation of tedious and repetitive tasks.

Testing the `head.py` program

> “Without requirements or design, programming is the art of adding bugs to an empty text file." - Louis Srygley

Python’s `pytest` module provides a rather simple and elegant way to construct a test suite. I have included a `test.py` to demonstrate how I typically write *integration tests* which exercise a program externally and verify that they work as intended. In my own programs, I also tend to write many *unit tests* that similarly exercise *individual functions* (the "units" of programming) to ensure they work as expected.

By combining both unit and integration tests, I come to have greater confidence that my code works. More importantly, I feel free to refactor my code to improve algorithms and add features without fearing I will break features that worked previously. When testing, I always run the entire test suite in such a way that testing halts at the first failure:

```
$ make test
pytest -xv test.py
============================= test session starts ==============================
...

test.py::test_exists PASSED                                              [ 14%]
test.py::test_usage PASSED                                               [ 28%]
test.py::test_bad_file PASSED                                            [ 42%]
test.py::test_bad_num PASSED                                             [ 57%]
test.py::test_default PASSED                                             [ 71%]
test.py::test_n PASSED                                                   [ 85%]
test.py::test_multiple_files PASSED                                      [100%]

============================== 7 passed in 0.84s ===============================
```

This `test.py` is almost identical to the one for the `head.sh`. The differences mostly account for how I felt it best to handle errors in the two programs. The best improvement, of course, is that now my tests are in the same language as the program, so it's easier for me to go back and forth between them!

A more advanced version

The previous version of `head.py` is reflected in the `solution2.py` version. We can briefly look at `solution3.py` to explore more advanced ideas in Python like function definition, list comprehensions, mock file handles, and unit testing.

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark
Purpose: Python implementation of head
         This version handles multiple files
         and uses more advanced ideas like function definition,
         list comprehensions, mock file handles, unit testing, etc.
"""

import argparse
import io


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Python implementation of head',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        type=argparse.FileType('rt'),
                        nargs='+',
                        help='Input file')

    parser.add_argument('-n',
                        '--num',
                        help='Number of lines',
                        metavar='int',
                        type=int,
                        default=10)

    args = parser.parse_args()

    if args.num < 1:
        parser.error(f'--num "{args.num}" must be > 0')
    return args


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    show_header = len(args.file) > 1 ①
    heads = [head(fh, args.num, show_header) for fh in args.file] ②
    print('\n'.join(heads)) ③


# --------------------------------------------------
def head(fh, num, show_header): ④
    """Return num lines from file handle"""

    lines = [f'==> {fh.name} <==\n'] if show_header else [] ⑤
    for line_num, line in enumerate(fh, start=1): ⑥
        lines.append(line)
        if line_num == num:
            break

    return ''.join(lines) ⑦


# --------------------------------------------------
def test_head(): ⑧
    """Test head"""

    assert head(io.StringIO('foo\nbar\nbaz\n'), 1, False) == 'foo\n' ⑨
    assert head(io.StringIO('foo\nbar\nbaz\n'), 2, False) == 'foo\nbar\n' ⑩


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

① Whether or not we show a header between each file is a function of there being 1 or more files.

② We use a list comprehension to create the `head` of each file.

③ `print()` all the values of `head()` for each file.

④ Here we define a function called `head()` that takes three arguments.

⑤ We initialize a `lines` variable using an `if` expression that hinges on whether to `show_header`.

⑥ This is the same logic as before, but now rather than calling `print()` on each `line`, we append it to the `list` of `lines`.

⑦ Return the `lines` joined on the empty string. Note that each `line` still has any newline from the file.

⑧ Any function starting with `test_` will be run by `pytest`. Here I define a *unit test* to run the `head()` function.

⑨ The `head()` function expects something like an open file handle. I can use `io.StringIO` to make a sort of mock file handle from a string. Each value ending with a newline `\n` will appear as a "line" of text.

⑩ The `assert` statement will throw an exception if the given value does not evaluate as "true" (but not necessarily as `True`). Here I want the `head()` function to return the string `foo\nbar\n` when I run the function with the given values.

I can copy this version to `head.py` (or modify `test.py` to use `solution3.py` instead of `head.py`_ and run `pytest -xv test.py` to validate the program. Additionally, I can run `pytest` directly on the program to run my *unit tests*:

```
$ pytest -xv solution3.py
============================= test session starts ==============================
...

solution3.py::test_head PASSED                                           [100%]

============================== 1 passed in 0.01s ===============================
```

As programs grow in length and complexity, it makes more sense to write small, tested functions that can be composed into larger, more stable programs.

Creating new programs with `argparse` and `new.py`

Similar to the `new_bash.py` program I mentioned in the `bash` section, I have included the `new.py` program I use to create new Python programs that use `argparse` to validate arguments. You run it with `-h` for help, of course:

```
$ ./new.py -h
usage: new.py [-h] [-n NAME] [-e EMAIL] [-p PURPOSE] [-f] program

Create Python argparse program

positional arguments:
  program               Program name

optional arguments:
  -h, --help            show this help message and exit
  -n NAME, --name NAME  Name for docstring (default: Ken Youens-Clark)
  -e EMAIL, --email EMAIL
                        Email for docstring (default: kyclark@gmail.com)
  -p PURPOSE, --purpose PURPOSE
                        Purpose for docstring (default: Rock the Casbah)
  -f, --force           Overwrite existing (default: False)
```

If I wanted to create, for instance, a Python implementation of `cat` now, I might do this:

```
$ ./new.py cat.py
Done, see new script "cat.py."
```

And now I have a `cat.py` program that I can immediately execute that shows the typical kinds of positional and optional parameters a program might have:

```
$ ./cat.py -h
usage: cat.py [-h] [-a str] [-i int] [-f FILE] [-o] str

Rock the Casbah

positional arguments:
  str                   A positional argument

optional arguments:
  -h, --help            show this help message and exit
  -a str, --arg str     A named string argument (default: )
  -i int, --int int     A named integer argument (default: 0)
  -f FILE, --file FILE  A readable file (default: None)
  -o, --on              A boolean flag (default: False)
```

You can modify the `get_args()` function to reflect the needs of your new program. Place this program into your `$PATH` to use system-wide whenever you need to create a new Python program. I hope you may find this speeds you in your development of new Python programs!

Going further

- Tiny Python Projects: All of these ideas about `argparse` and testing are discussed in greater detail in this book available now from Manning Publications.
- Make tutorial: Dustin and I also briefly discussed the use of `make` and `Makefile` to document and automate the running of tests and shortcuts and commands and such.
- GitHub repo: All the code and tests for Tiny Python Projects
- YouTube: Videos for the chapters of Tiny Python Projects that demonstrate how to start writing Python programs and work through the requirements and tests and solutions.

Author

Ken Youens-Clark

- kyclark@gmail.com
- [@kycl4rk](https://twitter.com/kycl4rk)
- [LinkedIn](https://www.linkedin.com/in/kycl4rk/)

[1] Utilities like `head` or `grep` can vary among systems and distributions. I tried `head` on both Linux and Mac, and neither recognized the help flags.