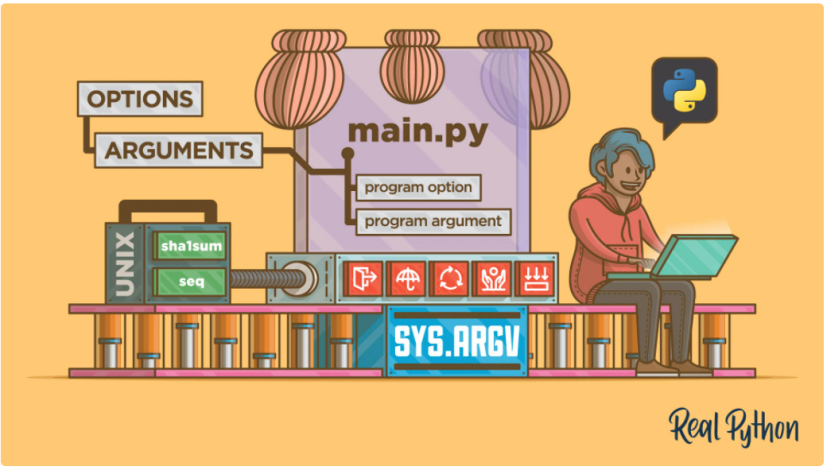


Python Command Line Arguments

by Andre Burgaud Feb 05, 2020 6 Comments best-practices intermediate tools

Table of Contents

Table of Contents

- The Command Line Interface
- The C Legacy
- Two Utilities From the Unix World
- The sys.argv Array
- The Anatomy of Python Command Line Arguments
- A Few Methods for Parsing Python Command Line Arguments
- A Few Methods for Validating Python Command Line Arguments
- The Python Standard Library
- A Few External Python Packages
- Conclusion
- Additional Resources

Mark as Completed

Tweet Share Email

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Command Line Interfaces in Python**

Adding the capability of processing **Python command line arguments** provides a user-friendly interface to your text-based command line program. It's similar to what a graphical user interface is for a visual application that's manipulated by graphical elements or widgets.

Python exposes a mechanism to capture and extract your Python command line arguments. These values can be used to modify the behavior of a program. For example, if your program processes data read from a file, then you can pass the name of the file to your program, rather than hard-coding the value in your source code.

By the end of this tutorial, you'll know:

- **The origins** of Python command line arguments
- **The underlying support** for Python command line arguments
- **The standards** guiding the design of a command line interface
- **The basics** to manually customize and handle Python command line arguments
- **The libraries** available in Python to ease the development of a complex command line interface

If you want a user-friendly way to supply Python command line arguments to your program without importing a dedicated library, or if you want to better understand the common basis for the existing libraries that are dedicated to building the Python command line interface, then keep on reading!

The Command Line Interface

A **command line interface (CLI)** provides a way for a user to interact with a program running in a text-based shell interpreter. Some examples of shell interpreters are Bash on Linux or Command Prompt on Windows. A command line interface is enabled by the shell interpreter that exposes a command prompt. It can be characterized by the following elements:

- A **command** or program
- Zero or more command line **arguments**
- An **output** representing the result of the command
- Textual documentation referred to as **usage** or **help**

Not every command line interface may provide all these elements, but this list isn't exhaustive, either. The complexity of the command line ranges from the ability to pass a single argument, to numerous arguments and options, much like a Domain Specific Language. For example, some programs may launch web documentation from the command line or start an interactive shell interpreter like Python.

The two following examples with the Python command illustrates the description of a command line interface:

Shell

```
$ python -c "print('Real Python')"
```

```
Real Python
```

In this first example, the Python interpreter takes option `-c` for **command**, which says to execute the Python command line arguments following the option `-c` as a Python program.

Another example shows how to invoke Python with `-h` to display the help:

Shell

```
$ python -h
usage: python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
[ ... complete help text not shown ... ]
```

Try this out in your terminal to see the complete help documentation.

The C Legacy

Python command line arguments directly inherit from the C programming language. As Guido Van Rossum wrote in An Introduction to Python for Unix/C Programmers in 1993, C had a strong influence on Python. Guido mentions the definitions of literals, identifiers, operators, and statements like `break`, `continue`, or `return`. The use of Python command line arguments is also strongly influenced by the C language.

To illustrate the similarities, consider the following C program:

C

```c
// main.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Arguments count: %d\n", argc);
    for (int i = 0; i < argc; i++) {
        printf("Argument %6d: %s\n", i, argv[i]);
    }
    return 0;
}
```

Line 4 defines `main()`, which is the entry point of a C program. Take good note of the parameters:

1. **`argc`** is an integer representing the number of arguments of the program.
2. **`argv`** is an array of pointers to characters containing the name of the program in the first element of the array, followed by the arguments of the program, if any, in the remaining elements of the array.

You can compile the code above on Linux with `gcc -o main main.c`, then execute with `./main` to obtain the following:

Shell

```
$ gcc -o main main.c
$ ./main
Arguments count: 1
Argument      0: ./main
```

Unless explicitly expressed at the command line with the option `-o`, `a.out` is the default name of the executable generated by the **gcc** compiler. It stands for **assembler output** and is reminiscent of the executables that were generated on older UNIX systems. Observe that the name of the executable `./main` is the sole argument.

Let's spice up this example by passing a few Python command line arguments to the same

program:

Shell

```
$ ./main Python Command Line Arguments
Arguments count: 5
Argument     0: ./main
Argument     1: Python
Argument     2: Command
Argument     3: Line
Argument     4: Arguments
```

The output shows that the number of arguments is `5`, and the list of arguments includes the name of the program, `main`, followed by each word of the phrase `"Python Command Line Arguments"`, which you passed at the command line.

> **Note**: `argc` stands for **argument count**, while `argv` stands for **argument vector**. To learn more, check out A Little C Primer/C Command Line Arguments.

The compilation of `main.c` assumes that you used a Linux or a Mac OS system. On Windows, you can also compile this C program with one of the following options:

- **Windows Subsystem for Linux (WSL):** It's available in a few Linux distributions, like Ubuntu, OpenSUSE, and Debian, among others. You can install it from the Microsoft Store.
- **Windows Build Tools:** This includes the Windows command line build tools, the Microsoft C/C++ compiler `cl.exe`, and a compiler front end named `clang.exe` for C/C++.
- **Microsoft Visual Studio:** This is the main Microsoft integrated development environment (IDE). To learn more about IDEs that can be used for both Python and C on various operating systems, including Windows, check out Python IDEs and Code Editors (Guide).
- **mingw-64 project:** This supports the GCC compiler on Windows.

If you've installed Microsoft Visual Studio or the Windows Build Tools, then you can compile `main.c` as follows:

Windows Command Prompt

```
C:/>cl main.c
```

You'll obtain an executable named `main.exe` that you can start with:

Windows Command Prompt

```
C:/>main
Arguments count: 1
Argument     0: main
```

You could implement a Python program, `main.py`, that's equivalent to the C program, `main.c`, you saw above:

Python

```
# main.py
import sys

if __name__ == "__main__":
    print(f"Arguments count: {len(sys.argv)}")
    for i, arg in enumerate(sys.argv):
        print(f"Argument {i:>6}: {arg}")
```

You don't see an `argc` variable like in the C code example. It doesn't exist in Python because `sys.argv` is sufficient. You can parse the Python command line arguments in `sys.argv` without having to know the length of the list, and you can call the built-in `len()` if the number of arguments is needed by your program.

Also, note that `enumerate()`, when applied to an iterable, returns an `enumerate` object that can emit pairs associating the index of an element in `sys.arg` to its corresponding value. This allows looping through the content of `sys.argv` without having to maintain a counter for the

index in the list.

Execute `main.py` as follows:

Shell

```
$ python main.py Python Command Line Arguments
Arguments count: 5
Argument      0: main.py
Argument      1: Python
Argument      2: Command
Argument      3: Line
Argument      4: Arguments
```

`sys.argv` contains the same information as in the C program:

- **The name of the program** `main.py` is the first item of the list.
- **The arguments** `Python`, `Command`, `Line`, and `Arguments` are the remaining elements in the list.

With this short introduction into a few arcane aspects of the C language, you're now armed with some valuable knowledge to further grasp Python command line arguments.

Two Utilities From the Unix World

To use Python command line arguments in this tutorial, you'll implement some partial features of two utilities from the Unix ecosystem:

1. sha1sum
2. seq

You'll gain some familiarity with these Unix tools in the following sections.

`sha1sum`

`sha1sum` calculates SHA-1 hashes, and it's often used to verify the integrity of files. For a given input, a **hash function** always returns the same value. Any minor changes in the input will result in a different hash value. Before you use the utility with concrete parameters, you may try to display the help:

Shell

```
$ sha1sum --help
Usage: sha1sum [OPTION]... [FILE]...
Print or check SHA1 (160-bit) checksums.

With no FILE, or when FILE is -, read standard input.

  -b, --binary         read in binary mode
  -c, --check          read SHA1 sums from the FILEs and check them
      --tag            create a BSD-style checksum
  -t, --text           read in text mode (default)
  -z, --zero           end each output line with NUL, not newline,
                       and disable file name escaping
[ ... complete help text not shown ... ]
```

Displaying the help of a command line program is a common feature exposed in the command line interface.

To calculate the SHA-1 hash value of the content of a file, you proceed as follows:

Shell

```
$ sha1sum main.c
125a0f900ff6f164752600550879cbfabb098bc3  main.c
```

The result shows the SHA-1 hash value as the first field and the name of the file as the second field. The command can take more than one file as arguments:

Shell

```
$ sha1sum main.c main.py
125a0f900ff6f164752600550879cbfabb098bc3  main.c
d84372fc77a90336b6bb7c5e959bcb1b24c608b4  main.py
```

Thanks to the wildcards expansion feature of the Unix terminal, it's also possible to provide Python command line arguments with wildcard characters. One such a character is the asterisk or star (`*`):

Shell

```
$ sha1sum main.*
3f6d5274d6317d580e2ffc1bf52beee0d94bf078  main.c
f41259ea5835446536d2e71e566075c1c1bfc111  main.py
```

The shell converts `main.*` to `main.c` and `main.py`, which are the two files matching the pattern `main.*` in the current directory, and passes them to `sha1sum`. The program calculates the **SHA1 hash** of each of the files in the argument list. You'll see that, on Windows, the behavior is different. Windows has no wildcard expansion, so the program may have to accommodate for that. Your implementation may need to expand wildcards internally.

Without any argument, `sha1sum` reads from the standard input. You can feed data to the program by typing characters on the keyboard. The input may incorporate any characters, including the carriage return `Enter ↩`. To terminate the input, you must signal the end of file with `Enter ↩`, followed by the sequence `^ Ctrl` + `D`:

Shell

```
$ sha1sum
Real
Python
87263a73c98af453d68ee4aab61576b331f8d9d6  -
```

You first enter the name of the program, `sha1sum`, followed by `Enter ↩`, and then `Real` and `Python`, each also followed by `Enter ↩`. To close the input stream, you type `^ Ctrl` + `D`. The result is the value of the SHA1 hash generated for the text `Real\nPython\n`. The name of the file is `-`. This is a convention to indicate the standard input. The hash value is the same when you execute the following commands:

Shell

```
$ python -c "print('Real\nPython\n', end='')" | sha1sum
87263a73c98af453d68ee4aab61576b331f8d9d6  -
$ python -c "print('Real\nPython')" | sha1sum
87263a73c98af453d68ee4aab61576b331f8d9d6  -
$ printf "Real\nPython\n" | sha1sum
87263a73c98af453d68ee4aab61576b331f8d9d6  -
```

Up next, you'll read a short description of `seq`.

seq

`seq` generates a **sequence** of numbers. In its most basic form, like generating the sequence from 1 to 5, you can execute the following:

Shell

```
$ seq 5
1
2
3
```

```
4
5
```

To get an overview of the possibilities exposed by `seq`, you can display the help at the command line:

Shell

```
$ seq --help
Usage: seq [OPTION]... LAST
  or:  seq [OPTION]... FIRST LAST
  or:  seq [OPTION]... FIRST INCREMENT LAST
Print numbers from FIRST to LAST, in steps of INCREMENT.

Mandatory arguments to long options are mandatory for short options too.
  -f, --format=FORMAT      use printf style floating-point FORMAT
  -s, --separator=STRING   use STRING to separate numbers (default: \n)
  -w, --equal-width        equalize width by padding with leading zeroes
      --help     display this help and exit
      --version  output version information and exit
[ ... complete help text not shown ... ]
```

For this tutorial, you'll write a few simplified variants of `sha1sum` and `seq`. In each example, you'll learn a different facet or combination of features about Python command line arguments.

On Mac OS and Linux, `sha1sum` and `seq` should come pre-installed, though the features and the help information may sometimes differ slightly between systems or distributions. If you're using Windows 10, then the most convenient method is to run `sha1sum` and `seq` in a Linux environment installed on the WSL. If you don't have access to a terminal exposing the standard Unix utilities, then you may have access to online terminals:

- **Create** a free account on PythonAnywhere and start a Bash Console.
- **Create** a temporary Bash terminal on repl.it.

These are two examples, and you may find others.

The `sys.argv` Array

Before exploring some accepted conventions and discovering how to handle Python command line arguments, you need to know that the underlying support for all Python command line arguments is provided by `sys.argv`. The examples in the following sections show you how to handle the Python command line arguments stored in `sys.argv` and to overcome typical issues that occur when you try to access them. You'll learn:

- How to **access** the content of `sys.argv`
- How to **mitigate** the side effects of the global nature of `sys.argv`
- How to **process** whitespaces in Python command line arguments
- How to **handle** errors while accessing Python command line arguments
- How to **ingest** the original format of the Python command line arguments passed by bytes

Let's get started!

Displaying Arguments

The `sys` module exposes an array named `argv` that includes the following:

1. **`argv[0]`** contains the name of the current Python program.
2. **`argv[1:]`**, the rest of the list, contains any and all Python command line arguments passed to the program.

The following example demonstrates the content of `sys.argv`:

Python

```
1 # argv.py
2 import sys
```

```
4 print(f"Name of the script      : {sys.argv[0]=}")
5 print(f"Arguments of the script : {sys.argv[1:]=}")
```

Here's how this code works:

- **Line 2** imports the internal Python module `sys`.
- **Line 4** extracts the name of the program by accessing the first element of the list `sys.argv`.
- **Line 5** displays the Python command line arguments by fetching all the remaining elements of the list `sys.argv`.

> **Note**: The f-string syntax used in `argv.py` leverages the new debugging specifier in Python 3.8. To read more about this new f-string feature and others, check out Cool New Features in Python 3.8.
>
> If your Python version is less than 3.8, then simply remove the equals sign (`=`) in both f-strings to allow the program to execute successfully. The output will only display the value of the variables, not their names.

Execute the script `argv.py` above with a list of arbitrary arguments as follows:

Shell

```
$ python argv.py un deux trois quatre
Name of the script      : sys.argv[0]='argv.py'
Arguments of the script : sys.argv[1:]=['un', 'deux', 'trois', 'quatre']
```

The output confirms that the content of `sys.argv[0]` is the Python script `argv.py`, and that the remaining elements of the `sys.argv` list contains the arguments of the script, `['un', 'deux', 'trois', 'quatre']`.

To summarize, `sys.argv` contains all the `argv.py` Python command line arguments. When the Python interpreter executes a Python program, it parses the command line and populates `sys.argv` with the arguments.

Reversing the First Argument

Now that you have enough background on `sys.argv`, you're going to operate on arguments passed at the command line. The example `reverse.py` reverses the first argument passed at the command line:

Python

```
1 # reverse.py
2
3 import sys
4
5 arg = sys.argv[1]
6 print(arg[::-1])
```

In `reverse.py` the process to reverse the first argument is performed with the following steps:

- **Line 5** fetches the first argument of the program stored at index `1` of `sys.argv`. Remember that the program name is stored at index `0` of `sys.argv`.
- **Line 6** prints the reversed string. `args[::-1]` is a Pythonic way to use a slice operation to reverse a list.

You execute the script as follows:

Shell

```
$ python reverse.py "Real Python"
nohtyP laeR
```

As expected, `reverse.py` operates on `"Real Python"` and reverses the only argument to output `"nohtyP laeR"`. Note that surrounding the multi-word string `"Real Python"` with quotes ensures that the interpreter handles it as a unique argument, instead of two arguments. You'll delve into **argument separators** in a later section.

Mutating `sys.argv`

`sys.argv` is **globally available** to your running Python program. All modules imported during the execution of the process have direct access to `sys.argv`. This global access might be convenient, but `sys.argv` isn't immutable. You may want to implement a more reliable mechanism to expose program arguments to different modules in your Python program, especially in a complex program with multiple files.

Observe what happens if you tamper with `sys.argv`:

Python

```python
# argv_pop.py

import sys

print(sys.argv)
sys.argv.pop()
print(sys.argv)
```

You invoke `.pop()` to remove and return the last item in `sys.argv`.

Execute the script above:

Shell

```
$ python argv_pop.py un deux trois quatre
['argv_pop.py', 'un', 'deux', 'trois', 'quatre']
['argv_pop.py', 'un', 'deux', 'trois']
```

Notice that the fourth argument is no longer included in `sys.argv`.

In a short script, you can safely rely on the global access to `sys.argv`, but in a larger program, you may want to store arguments in a separate variable. The previous example could be modified as follows:

Python

```python
# argv_var_pop.py

import sys

print(sys.argv)
args = sys.argv[1:]
print(args)
sys.argv.pop()
print(sys.argv)
print(args)
```

This time, although `sys.argv` lost its last element, `args` has been safely preserved. `args` isn't global, and you can pass it around to parse the arguments per the logic of your program. The Python package manager, `pip`, uses this approach. Here's a short excerpt of the `pip` source code:

Python

```python
def main(args=None):
    if args is None:
        args = sys.argv[1:]
```

In this snippet of code taken from the `pip` source code, `main()` saves into `args` the slice of `sys.argv` that contains only the arguments and not the file name. `sys.argv` remains untouched, and `args` isn't impacted by any inadvertent changes to `sys.argv`.

Escaping Whitespace Characters

In the `reverse.py` example you saw earlier, the first and only argument is `"Real Python"`, and the result is `"nohtyP laeR"`. The argument includes a whitespace separator between `"Real"` and `"Python"`, and it needs to be escaped.

On Linux, whitespaces can be escaped by doing one of the following:

1. **Surrounding** the arguments with single quotes (`'`)
2. **Surrounding** the arguments with double quotes (`"`)
3. **Prefixing** each space with a backslash (`\`)

Without one of the escape solutions, `reverse.py` stores two arguments, `"Real"` in `sys.argv[1]` and `"Python"` in `sys.argv[2]`:

Shell

```
$ python reverse.py Real Python
laeR
```

The output above shows that the script only reverses `"Real"` and that `"Python"` is ignored. To ensure both arguments are stored, you'd need to surround the overall string with double quotes (`"`).

You can also use a backslash (`\`) to escape the whitespace:

Shell

```
$ python reverse.py Real\ Python
nohtyP laeR
```

With the backslash (`\`), the command shell exposes a unique argument to Python, and then to `reverse.py`.

In Unix shells, the internal field separator (IFS) defines characters used as **delimiters**. The content of the shell variable, `IFS`, can be displayed by running the following command:

Shell

```
$ printf "%q\n" "$IFS"
$' \t\n'
```

From the result above, `' \t\n'`, you identify three delimiters:

1. **Space** (`' '`)
2. **Tab** (`\t`)
3. **Newline** (`\n`)

Prefixing a space with a backslash (`\`) bypasses the default behavior of the space as a delimiter in the string `"Real Python"`. This results in one block of text as intended, instead of two.

Note that, on Windows, the whitespace interpretation can be managed by using a combination of double quotes. It's slightly counterintuitive because, in the Windows terminal, a double quote (`"`) is interpreted as a switch to disable and subsequently to enable special characters like **space**, **tab**, or **pipe** (`|`).

As a result, when you surround more than one string with double quotes, the Windows terminal interprets the first double quote as a command to **ignore special characters** and the second double quote as one to **interpret special characters**.

With this information in mind, it's safe to assume that surrounding more than one string with double quotes will give you the expected behavior, which is to expose the group of

strings as a single argument. To confirm this peculiar effect of the double quote on the Windows command line, observe the following two examples:

Windows Command Prompt

```
C:/>python reverse.py "Real Python"
nohtyP laeR
```

In the example above, you can intuitively deduce that `"Real Python"` is interpreted as a single argument. However, realize what occurs when you use a single double quote:

Windows Command Prompt

```
C:/>python reverse.py "Real Python
nohtyP laeR
```

The command prompt passes the whole string `"Real Python"` as a single argument, in the same manner as if the argument was `"Real Python"`. In reality, the Windows command prompt sees the unique double quote as a switch to disable the behavior of the whitespaces as separators and passes anything following the double quote as a unique argument.

For more information on the effects of double quotes in the Windows terminal, check out A Better Way To Understand Quoting and Escaping of Windows Command Line Arguments.

Handling Errors

Python command line arguments are **loose strings**. Many things can go wrong, so it's a good idea to provide the users of your program with some guidance in the event they pass incorrect arguments at the command line. For example, `reverse.py` expects one argument, and if you omit it, then you get an error:

Shell

```
$ python reverse.py
Traceback (most recent call last):
  File "reverse.py", line 5, in <module>
    arg = sys.argv[1]
IndexError: list index out of range
```

The Python exception `IndexError` is raised, and the corresponding traceback shows that the error is caused by the expression `arg = sys.argv[1]`. The message of the exception is `list index out of range`. You didn't pass an argument at the command line, so there's nothing in the list `sys.argv` at index `1`.

This is a common pattern that can be addressed in a few different ways. For this initial example, you'll keep it brief by including the expression `arg = sys.argv[1]` in a `try` block. Modify the code as follows:

Python

```
# reverse_exc.py

import sys

try:
    arg = sys.argv[1]
except IndexError:
    raise SystemExit(f"Usage: {sys.argv[0]} <string_to_reverse>")
print(arg[::-1])
```

The expression on line 4 is included in a `try` block. Line 8 raises the built-in exception `SystemExit`. If no argument is passed to `reverse_exc.py`, then the process exits with a status code of `1` after printing the usage. Note the integration of `sys.argv[0]` in the error message. It exposes the name of the program in the usage message. Now, when you execute the same

program without any Python command line arguments, you can see the following output:

Shell

```
$ python reverse.py
Usage: reverse.py <string_to_reverse>

$ echo $?
1
```

`reverse.py` didn't have an argument passed at the command line. As a result, the program raises `SystemExit` with an error message. This causes the program to exit with a status of `1`, which displays when you print the special variable `$?` with `echo`.

Calculating the `sha1sum`

You'll write another script to demonstrate that, on Unix-like systems, Python command line arguments are passed by bytes from the OS. This script takes a string as an argument and outputs the hexadecimal SHA-1 hash of the argument:

Python

```
1 # sha1sum.py
2
3 import sys
4 import hashlib
5
6 data = sys.argv[1]
7 m = hashlib.sha1()
8 m.update(bytes(data, 'utf-8'))
9 print(m.hexdigest())
```

This is loosely inspired by `sha1sum`, but it intentionally processes a string instead of the contents of a file. In `sha1sum.py`, the steps to ingest the Python command line arguments and to output the result are the following:

- **Line 6** stores the content of the first argument in `data`.
- **Line 7** instantiates a SHA1 algorithm.
- **Line 8** updates the SHA1 hash object with the content of the first program argument. Note that `hash.update` takes a byte array as an argument, so it's necessary to convert `data` from a string to a bytes array.
- **Line 9** prints a hexadecimal representation of the SHA1 hash computed on line 8.

When you run the script with an argument, you get this:

Shell

```
$ python sha1sum.py "Real Python"
0554943d034f044c5998f55dac8ee2c03e387565
```

For the sake of keeping the example short, the script `sha1sum.py` doesn't handle missing Python command line arguments. Error handling could be addressed in this script the same way you did it in `reverse_exc.py`.

> **Note:** Checkout `hashlib` for more details about the hash functions available in the Python standard library.

From the `sys.argv` documentation, you learn that in order to get the original bytes of the Python command line arguments, you can use `os.fsencode()`. By directly obtaining the bytes from `sys.argv[1]`, you don't need to perform the string-to-bytes conversion of `data`:

Python

```
1 # sha1sum_bytes.py
2
3 import os
4 import sys
5 import hashlib
6
7 data = os.fsencode(sys.argv[1])
```

```
8  m = hashlib.sha1()
9  m.update(data)
10 print(m.hexdigest())
```

The main difference between `sha1sum.py` and `sha1sum_bytes.py` are highlighted in the following lines:

- **Line 7** populates `data` with the original bytes passed to the Python command line arguments.
- **Line 9** passes `data` as an argument to `m.update()`, which receives a bytes-like object.

Execute `sha1sum_bytes.py` to compare the output:

Shell

```
$ python sha1sum_bytes.py "Real Python"
0554943d034f044c5998f55dac8ee2c03e387565
```

The hexadecimal value of the SHA1 hash is the same as in the previous `sha1sum.py` example.

The Anatomy of Python Command Line Arguments

Now that you've explored a few aspects of Python command line arguments, most notably `sys.argv`, you're going to apply some of the standards that are regularly used by developers while implementing a command line interface.

Python command line arguments are a subset of the command line interface. They can be composed of different types of arguments:

1. **Options** modify the behavior of a particular command or program.
2. **Arguments** represent the source or destination to be processed.
3. **Subcommands** allow a program to define more than one command with the respective set of options and arguments.

Before you go deeper into the different types of arguments, you'll get an overview of the accepted standards that have been guiding the design of the command line interface and arguments. These have been refined since the advent of the computer terminal in the mid-1960s.

Standards

A few available **standards** provide some definitions and guidelines to promote consistency for implementing commands and their arguments. These are the main UNIX standards and references:

- POSIX Utility Conventions
- GNU Standards for Command Line Interfaces
- docopt

The standards above define guidelines and nomenclatures for anything related to programs and Python command line arguments. The following points are examples taken from those references:

- **POSIX**:
  - A program or utility is followed by options, option-arguments, and operands.
  - All options should be preceded with a hyphen or minus (-) delimiter character.
  - Option-arguments should not be optional.
- **GNU**:
  - All programs should support two standard options, which are `--version` and `--`

All programs should support two standard options, which are `--version` and `--help`.

- Long-named options are equivalent to the single-letter Unix-style options. An example is `--debug` and `-d`.

- **docopt**:
  - Short options can be stacked, meaning that `-abc` is equivalent to `-a -b -c`.
  - Long options can have arguments specified after a space or the equals sign (`=`). The long option `--input=ARG` is equivalent to `--input ARG`.

These standards define notations that are helpful when you describe a command. A similar notation can be used to display the usage of a particular command when you invoke it with the option `-h` or `--help`.

The GNU standards are very similar to the POSIX standards but provide some modifications and extensions. Notably, they add the **long option** that's a fully named option prefixed with two hyphens (`--`). For example, to display the help, the regular option is `-h` and the long option is `--help`.

> **Note:** You don't need to follow those standards rigorously. Instead, follow the conventions that have been used successfully for years since the advent of UNIX. If you write a set of utilities for you or your team, then ensure that you **stay consistent** across the different utilities.

In the following sections, you'll learn more about each of the command line components, options, arguments, and sub-commands.

Options

An **option**, sometimes called a **flag** or a **switch**, is intended to modify the behavior of the program. For example, the command `ls` on Linux lists the content of a given directory. Without any arguments, it lists the files and directories in the current directory:

Shell

```
$ cd /dev
$ ls
autofs
block
bsg
btrfs-control
bus
char
console
```

Let's add a few options. You can combine `-l` and `-s` into `-ls`, which changes the information displayed in the terminal:

Shell

```
$ cd /dev
$ ls -ls
total 0
0 crw-r--r--  1 root root       10,   235 Jul 14 08:10 autofs
0 drwxr-xr-x  2 root root             260 Jul 14 08:10 block
0 drwxr-xr-x  2 root root              60 Jul 14 08:10 bsg
0 crw-------  1 root root       10,   234 Jul 14 08:10 btrfs-control
0 drwxr-xr-x  3 root root              60 Jul 14 08:10 bus
0 drwxr-xr-x  2 root root            4380 Jul 14 15:08 char
0 crw-------  1 root root        5,     1 Jul 14 08:10 console
```

An **option** can take an argument, which is called an **option-argument**. See an example in action with `od` below:

Shell

```
$ od -t x1z -N 16 main
0000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00  >.ELF............<
0000020
```

`od` stands for **octal dump**. This utility displays data in different printable representations, like octal (which is the default), hexadecimal, decimal, and ASCII. In the example above, it takes the binary file `main` and displays the first 16 bytes of the file in hexadecimal format. The option `-t` expects a type as an option-argument, and `-N` expects the number of input bytes.

In the example above, `-t` is given type `x1`, which stands for hexadecimal and one byte per integer. This is followed by `z` to display the printable characters at the end of the input line. `-N` takes `16` as an option-argument for limiting the number of input bytes to 16.

Arguments

The **arguments** are also called **operands** or parameters in the POSIX standards. The arguments represent the source or the destination of the data that the command acts on. For example, the command `cp`, which is used to copy one or more files to a file or a directory, takes at least one source and one target:

Shell

```
1 $ ls main
2 main
3
4 $ cp main main2
5
6 $ ls -lt
7 main
8 main2
9 ...
```

In line 4, `cp` takes two arguments:

1. **`main`:** the source file
2. **`main2`:** the target file

It then copies the content of `main` to a new file named `main2`. Both `main` and `main2` are arguments, or operands, of the program `cp`.

Subcommands

The concept of **subcommands** isn't documented in the POSIX or GNU standards, but it does appear in **docopt**. The standard Unix utilities are small tools adhering to the Unix philosophy. Unix programs are intended to be programs that do one thing and do it well. This means no subcommands are necessary.

By contrast, a new generation of programs, including `git`, `go`, `docker`, and `gcloud`, come with a slightly different paradigm that embraces subcommands. They're not necessarily part of the Unix landscape as they span several operating systems, and they're deployed with a full ecosystem that requires several commands.

Take `git` as an example. It handles several commands, each possibly with their own set of options, option-arguments, and arguments. The following examples apply to the git subcommand `branch`:

- **`git branch`** displays the branches of the local git repository.
- **`git branch custom_python`** creates a local branch `custom_python` in a local repository.
- **`git branch -d custom_python`** deletes the local branch `custom_python`.
- **`git branch --help`** displays the help for the `git branch` subcommand.

In the Python ecosystem, `pip` has the concept of subcommands, too. Some `pip` subcommands include `list`, `install`, `freeze`, or `uninstall`.

Windows

On Windows, the conventions regarding Python command line arguments are slightly different, in particular, those regarding command line options. To validate this difference, take `tasklist`, which is a native Windows executable that displays a list of the currently running processes. It's similar to `ps` on Linux or macOS systems. Below is an example of how to execute `tasklist` in a command prompt on Windows:

Windows Command Prompt

```
C:/>tasklist /FI "IMAGENAME eq notepad.exe"

Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
notepad.exe                  13104 Console                    6     13,548 K
notepad.exe                   6584 Console                    6     13,696 K
```

Note that the separator for an option is a forward slash (`/`) instead of a hyphen (`-`) like the conventions for Unix systems. For readability, there's a space between the program name, `tasklist`, and the option `/FI`, but it's just as correct to type `tasklist/FI`.

The particular example above executes `tasklist` with a filter to only show the Notepad processes currently running. You can see that the system has two running instances of the Notepad process. Although it's not equivalent, this is similar to executing the following command in a terminal on a Unix-like system:

Shell

```
$ ps -ef | grep vi | grep -v grep
andre     2117     4  0 13:33 tty1     00:00:00 vi .gitignore
andre     2163  2134  0 13:34 tty3     00:00:00 vi main.c
```

The `ps` command above shows all the current running `vi` processes. The behavior is consistent with the Unix Philosophy, as the output of `ps` is transformed by two `grep` filters. The first `grep` command selects all the occurrences of `vi`, and the second `grep` filters out the occurrence of `grep` itself.

With the spread of Unix tools making their appearance in the Windows ecosystem, non-Windows-specific conventions are also accepted on Windows.

Visuals

At the start of a Python process, Python command line arguments are split into two categories:

1. **Python options:** These influence the execution of the Python interpreter. For example, adding option `-O` is a means to optimize the execution of a Python program by removing `assert` and `__debug__` statements. There are other Python options available at the command line.

2. **Python program and its arguments:** Following the Python options (if there are any), you'll find the Python program, which is a file name that usually has the extension `.py`, and its arguments. By convention, those can also be composed of options and arguments.

Take the following command that's intended to execute the program `main.py`, which takes options and arguments. Note that, in this example, the Python interpreter also takes some options, which are `-B` and `-v`.

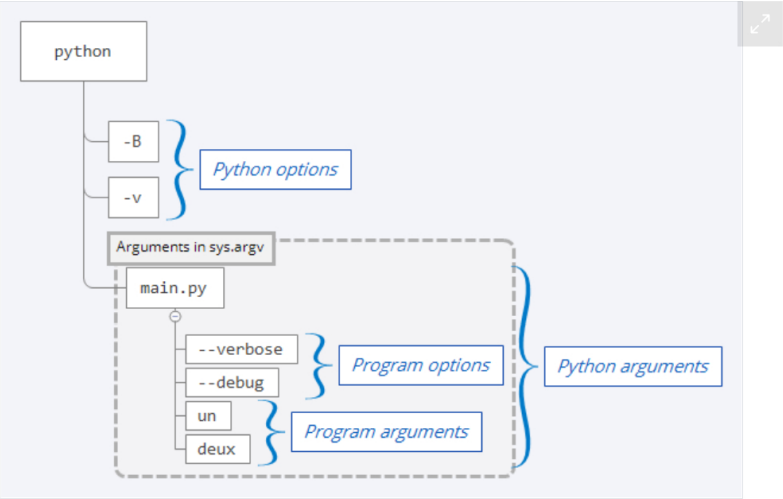
Shell

```
$ python -B -v main.py --verbose --debug un deux
```

In the command line above, the options are Python command line arguments and are organized as follows:

- **The option** `-B` tells Python not to write `.pyc` files on the import of source modules. For more details about `.pyc` files, check out the section What Does a Compiler Do? in Your Guide to the CPython Source Code.
- **The option** `-v` stands for **verbose** and tells Python to trace all import statements.
- **The arguments passed to** `main.py` are fictitious and represent two long options (`--verbose` and `--debug`) and two arguments (`un` and `deux`).

This example of Python command line arguments can be illustrated graphically as follows:



Within the Python program `main.py`, you only have access to the Python command line arguments inserted by Python in `sys.argv`. The Python options may influence the behavior of the program but are not accessible in `main.py`.

A Few Methods for Parsing Python Command Line Arguments

Now you're going to explore a few approaches to apprehend options, option-arguments, and operands. This is done by **parsing** Python command line arguments. In this section, you'll see some concrete aspects of Python command line arguments and techniques to handle them. First, you'll see an example that introduces a straight approach relying on list comprehensions to collect and separate options from arguments. Then you will:

- **Use** regular expressions to extract elements of the command line
- **Learn** how to handle files passed at the command line
- **Apprehend** the standard input in a way that's compatible with the Unix tools
- **Differentiate** the regular output of the program from the errors
- **Implement** a custom parser to read Python command line arguments

This will serve as a preparation for options involving modules in the standard libraries or from external libraries that you'll learn about later in this tutorial.

For something uncomplicated, the following pattern, which doesn't enforce ordering and doesn't handle option-arguments, may be enough:

Python

```
# cul.py

import sys

opts = [opt for opt in sys.argv[1:] if opt.startswith("-")]
args = [arg for arg in sys.argv[1:] if not arg.startswith("-")]

if "-c" in opts:
    print(" ".join(arg.capitalize() for arg in args))
elif "-u" in opts:
    print(" ".join(arg.upper() for arg in args))
elif "-l" in opts:
    print(" ".join(arg.lower() for arg in args))
else:
```

```
    raise SystemExit(f"Usage: {sys.argv[0]} (-c | -u | -l) <arguments>...")
```

The intent of the program above is to modify the case of the Python command line arguments. Three options are available:

- `-c` to capitalize the arguments
- `-u` to convert the arguments to uppercase
- `-l` to convert the argument to lowercase

The code collects and separates the different argument types using list comprehensions:

- **Line 5** collects all the **options** by filtering on any Python command line arguments starting with a hyphen (-).
- **Line 6** assembles the program **arguments** by filtering out the options.

When you execute the Python program above with a set of options and arguments, you get the following output:

Shell

```
$ python cul.py -c un deux trois
Un Deux Trois
```

This approach might suffice in many situations, but it would fail in the following cases:

- If the order is important, and in particular, if options should appear before the arguments
- If support for option-arguments is needed
- If some arguments are prefixed with a hyphen (-)

You can leverage other options before you resort to a library like `argparse` or `click`.

Regular Expressions

You can use a regular expression to enforce a certain order, specific options and option-arguments, or even the type of arguments. To illustrate the usage of a regular expression to parse Python command line arguments, you'll implement a Python version of `seq`, which is a program that prints a sequence of numbers. Following the docopt conventions, a specification for `seq.py` could be this:

Text

```
Print integers from <first> to <last>, in steps of <increment>.

Usage:
  python seq.py --help
  python seq.py [-s SEPARATOR] <last>
  python seq.py [-s SEPARATOR] <first> <last>
  python seq.py [-s SEPARATOR] <first> <increment> <last>

Mandatory arguments to long options are mandatory for short options too.
  -s, --separator=STRING use STRING to separate numbers (default: \n)
      --help             display this help and exit

If <first> or <increment> are omitted, they default to 1. When <first> is
larger than <last>, <increment>, if not set, defaults to -1.
The sequence of numbers ends when the sum of the current number and
<increment> reaches the limit imposed by <last>.
```

First, look at a regular expression that's intended to capture the requirements above:

Python

```
1 args_pattern = re.compile(
2     r"""
```

```
    ^
    (
        (--(?P<HELP>help).*)|
        ((?:-s|--separator)\s(?P<SEP>.*?)\s)?
        ((?P<OP1>-?\d+))(\s(?P<OP2>-?\d+))?(\s(?P<OP3>-?\d+))?
    )
    $
""",
    re.VERBOSE,
)
```

To experiment with the regular expression above, you may use the snippet recorded on Regular Expression 101. The regular expression captures and enforces a few aspects of the requirements given for `seq`. In particular, the command may take:

1. **A help option**, in short (`-h`) or long format (`--help`), captured as a named group called `HELP`
2. **A separator option**, `-s` or `--separator`, taking an optional argument, and captured as named group called `SEP`
3. **Up to three integer operands**, respectively captured as `OP1`, `OP2`, and `OP3`

For clarity, the pattern `args_pattern` above uses the flag `re.VERBOSE` on line 11. This allows you to spread the regular expression over a few lines to enhance readability. The pattern validates the following:

- **Argument order**: Options and arguments are expected to be laid out in a given order. For example, options are expected before the arguments.
- **Option values****: Only `--help`, `-s`, or `--separator` are expected as options.
- **Argument mutual exclusivity**: The option `--help` isn't compatible with other options or arguments.
- **Argument type**: Operands are expected to be positive or negative integers.

For the regular expression to be able to handle these things, it needs to see all Python command line arguments in one string. You can collect them using str.join():

Python
```
arg_line = " ".join(sys.argv[1:])
```

This makes `arg_line` a string that includes all arguments, except the program name, separated by a space.

Given the pattern `args_pattern` above, you can extract the Python command line arguments with the following function:

Python
```
def parse(arg_line: str) -> Dict[str, str]:
    args: Dict[str, str] = {}
    if match_object := args_pattern.match(arg_line):
        args = {k: v for k, v in match_object.groupdict().items()
                if v is not None}
    return args
```

The pattern is already handling the order of the arguments, mutual exclusivity between options and arguments, and the type of the arguments. `parse()` is applying `re.match()` to the argument line to extract the proper values and store the data in a dictionary.

The dictionary includes the names of each group as keys and their respective values. For example, if the `arg_line` value is `--help`, then the dictionary is `{'HELP': 'help'}`. If `arg_line` is `-s T 10`, then the dictionary becomes `{'SEP': 'T', 'OP1': '10'}`. You can expand the code block below to see an implementation of `seq` with regular expressions.

An Implementation of seq With Regular Expressions Show/Hide

At this point, you know a few ways to extract options and arguments from the command line. So far, the Python command line arguments were only strings or integers. Next, you'll

learn how to handle files passed as arguments.

File Handling

It's time now to experiment with Python command line arguments that are expected to be **file names**. Modify `sha1sum.py` to handle one or more files as arguments. You'll end up with a downgraded version of the original `sha1sum` utility, which takes one or more files as arguments and displays the hexadecimal SHA1 hash for each file, followed by the name of the file:

Python

```
# sha1sum_file.py

import hashlib
import sys

def sha1sum(filename: str) -> str:
    hash = hashlib.sha1()
    with open(filename, mode="rb") as f:
        hash.update(f.read())
    return hash.hexdigest()

for arg in sys.argv[1:]:
    print(f"{sha1sum(arg)}  {arg}")
```

`sha1sum()` is applied to the data read from each file that you passed at the command line, rather than the string itself. Take note that `m.update()` takes a bytes-like object as an argument and that the result of invoking `read()` after opening a file with the mode `rb` will return a `bytes` object. For more information about handling file content, check out Reading and Writing Files in Python, and in particular, the section Working With Bytes.

The evolution of `sha1sum_file.py` from handling strings at the command line to manipulating the content of files is getting you closer to the original implementation of `sha1sum`:

Shell

```
$ sha1sum main main.c
9a6f82c245f5980082dbf6faac47e5085083c07d  main
125a0f900ff6f164752600550879cbfabb098bc3  main.c
```

The execution of the Python program with the same Python command line arguments gives this:

Shell

```
$ python sha1sum_file.py main main.c
9a6f82c245f5980082dbf6faac47e5085083c07d  main
125a0f900ff6f164752600550879cbfabb098bc3  main.c
```

Because you interact with the shell interpreter or the Windows command prompt, you also get the benefit of the wildcard expansion provided by the shell. To prove this, you can reuse `main.py`, which displays each argument with the argument number and its value:

Shell

```
$ python main.py main.*
Arguments count: 5
Argument      0: main.py
Argument      1: main.c
Argument      2: main.exe
Argument      3: main.obj
Argument      4: main.py
```

You can see that the shell automatically performs wildcard expansion so that any file with a base name matching `main`, regardless of the extension, is part of `sys.argv`.

The wildcard expansion isn't available on Windows. To obtain the same behavior, you need to implement it in your code. To refactor `main.py` to work with wildcard expansion, you can use `glob`. The following example works on Windows and, though it isn't as concise as the

original `main.py`, the same code behaves similarly across platforms:

Python

```python
# main_win.py

import sys
import glob
import itertools
from typing import List

def expand_args(args: List[str]) -> List[str]:
    arguments = args[:1]
    glob_args = [glob.glob(arg) for arg in args[1:]]
    arguments += itertools.chain.from_iterable(glob_args)
    return arguments

if __name__ == "__main__":
    args = expand_args(sys.argv)
    print(f"Arguments count: {len(args)}")
    for i, arg in enumerate(args):
        print(f"Argument {i:>6}: {arg}")
```

In `main_win.py`, `expand_args` relies on `glob.glob()` to process the shell-style wildcards. You can verify the result on Windows and any other operating system:

Windows Command Prompt

```
C:/>python main_win.py main.*
Arguments count: 5
Argument      0: main_win.py
Argument      1: main.c
Argument      2: main.exe
Argument      3: main.obj
Argument      4: main.py
```

This addresses the problem of handling files using wildcards like the asterisk (`*`) or question mark (`?`), but how about `stdin`?

If you don't pass any parameter to the original `sha1sum` utility, then it expects to read data from the **standard input**. This is the text you enter at the terminal that ends when you type `^ Ctrl` + `D` on Unix-like systems or `^ Ctrl` + `Z` on Windows. These control sequences send an end of file (EOF) to the terminal, which stops reading from `stdin` and returns the data that was entered.

In the next section, you'll add to your code the ability to read from the standard input stream.

Standard Input

When you modify the previous Python implementation of `sha1sum` to handle the standard input using `sys.stdin`, you'll get closer to the original `sha1sum`:

Python

```python
# sha1sum_stdin.py

from typing import List
import hashlib
import pathlib
import sys

def process_file(filename: str) -> bytes:
    return pathlib.Path(filename).read_bytes()

def process_stdin() -> bytes:
    return bytes("".join(sys.stdin), "utf-8")

def sha1sum(data: bytes) -> str:
    sha1_hash = hashlib.sha1()
    sha1_hash.update(data)
    return sha1_hash.hexdigest()

def output_sha1sum(data: bytes, filename: str = "-") -> None:
```

```
    print(f"{sha1sum(data)}  {filename}")

def main(args: List[str]) -> None:
    if not args:
        args = ["-"]
    for arg in args:
        if arg == "-":
            output_sha1sum(process_stdin(), "-")
        else:
            output_sha1sum(process_file(arg), arg)

if __name__ == "__main__":
    main(sys.argv[1:])
```

Two conventions are applied to this new `sha1sum` version:

1. Without any arguments, the program expects the data to be provided in the standard input, `sys.stdin`, which is a readable file object.
2. When a hyphen (`-`) is provided as a file argument at the command line, the program interprets it as reading the file from the standard input.

Try this new script without any arguments. Enter the first aphorism of The Zen of Python, then complete the entry with the keyboard shortcut `^ Ctrl` + `D` on Unix-like systems or `^ Ctrl` + `Z` on Windows:

Shell

```
$ python sha1sum_stdin.py
Beautiful is better than ugly.
ae5705a3efd4488dfc2b4b80df85f60c67d998c4  -
```

You can also include one of the arguments as `stdin` mixed with the other file arguments like so:

Shell

```
$ python sha1sum_stdin.py main.py - main.c
d84372fc77a90336b6bb7c5e959bcb1b24c608b4  main.py
Beautiful is better than ugly.
ae5705a3efd4488dfc2b4b80df85f60c67d998c4  -
125a0f900ff6f164752600550879cbfabb098bc3  main.c
```

Another approach on Unix-like systems is to provide `/dev/stdin` instead of `-` to handle the standard input:

Shell

```
$ python sha1sum_stdin.py main.py /dev/stdin main.c
d84372fc77a90336b6bb7c5e959bcb1b24c608b4  main.py
Beautiful is better than ugly.
ae5705a3efd4488dfc2b4b80df85f60c67d998c4  /dev/stdin
125a0f900ff6f164752600550879cbfabb098bc3  main.c
```

On Windows there's no equivalent to `/dev/stdin`, so using `-` as a file argument works as expected.

The script `sha1sum_stdin.py` isn't covering all necessary error handling, but you'll cover some of the missing features later in this tutorial.

Standard Output and Standard Error

Command line processing may have a direct relationship with `stdin` to respect the conventions detailed in the previous section. The standard output, although not immediately relevant, is still a concern if you want to adhere to the Unix Philosophy. To allow small programs to be combined, you may have to take into account the three standard streams:

1. `stdin`
2. `stdout`
3. `stderr`

The output of a program becomes the input of another one, allowing you to chain small utilities. For example, if you wanted to sort the aphorisms of the Zen of Python, then you could execute the following:

Shell

```
$ python -c "import this" | sort
Although never is often better than *right* now.
Although practicality beats purity.
Although that way may not be obvious at first unless you're Dutch.
...
```

The output above is truncated for better readability. Now imagine that you have a program that outputs the same data but also prints some debugging information:

Python

```
# zen_sort_debug.py

print("DEBUG >>> About to print the Zen of Python")
import this
print("DEBUG >>> Done printing the Zen of Python")
```

Executing the Python script above gives:

Shell

```
$ python zen_sort_debug.py
DEBUG >>> About to print the Zen of Python
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
...
DEBUG >>> Done printing the Zen of Python
```

The ellipsis (`...`) indicates that the output was truncated to improve readability.

Now, if you want to sort the list of aphorisms, then execute the command as follows:

Shell

```
$ python zen_sort_debug.py | sort

Although never is often better than *right* now.
Although practicality beats purity.
Although that way may not be obvious at first unless you're Dutch.
Beautiful is better than ugly.
Complex is better than complicated.
DEBUG >>> About to print the Zen of Python
DEBUG >>> Done printing the Zen of Python
Errors should never pass silently.
...
```

You may realize that you didn't intend to have the debug output as the input of the `sort` command. To address this issue, you want to send traces to the standard errors stream, `stderr`, instead:

Python

```
# zen_sort_stderr.py
import sys

print("DEBUG >>> About to print the Zen of Python", file=sys.stderr)
import this
print("DEBUG >>> Done printing the Zen of Python", file=sys.stderr)
```

Execute `zen_sort_stderr.py` to observe the following:

Shell

```
$ python zen_sort_stderr.py | sort
DEBUG >>> About to print the Zen of Python
```

```
DEBUG >>> Done printing the Zen of Python

Although never is often better than *right* now.
Although practicality beats purity.
Although that way may not be obvious at first unless you're Dutch
....
```

Now, the traces are displayed to the terminal, but they aren't used as input for the `sort` command.

Custom Parsers

You can implement `seq` by relying on a regular expression if the arguments aren't too complex. Nevertheless, the regex pattern may quickly render the maintenance of the script difficult. Before you try getting help from specific libraries, another approach is to create a **custom parser**. The parser is a loop that fetches each argument one after another and applies a custom logic based on the semantics of your program.

A possible implementation for processing the arguments of `seq_parse.py` could be as follows:

Python

```python
def parse(args: List[str]) -> Tuple[str, List[int]]:
    arguments = collections.deque(args)
    separator = "\n"
    operands: List[int] = []
    while arguments:
        arg = arguments.popleft()
        if not operands:
            if arg == "--help":
                print(USAGE)
                sys.exit(0)
            if arg in ("-s", "--separator"):
                separator = arguments.popleft()
                continue
        try:
            operands.append(int(arg))
        except ValueError:
            raise SystemExit(USAGE)
        if len(operands) > 3:
            raise SystemExit(USAGE)

    return separator, operands
```

`parse()` is given the list of arguments without the Python file name and uses `collections.deque()` to get the benefit of `.popleft()`, which removes the elements from the left of the collection. As the items of the arguments list unfold, you apply the logic that's expected for your program. In `parse()` you can observe the following:

- The **`while` loop** is at the core of the function, and terminates when there are no more arguments to parse, when the help is invoked, or when an error occurs.
- If the **`separator`** option is detected, then the next argument is expected to be the separator.
- **`operands`** stores the integers that are used to calculate the sequence. There should be at least one operand and at most three.

A full version of the code for `parse()` is available below:

Click to expand the full example. Show/Hide

This manual approach of parsing the Python command line arguments may be sufficient for a simple set of arguments. However, it becomes quickly error-prone when complexity increases due to the following:

- **A large number** of arguments
- **Complexity and interdependency** between arguments
- **Validation** to perform against the arguments

The custom approach isn't reusable and requires reinventing the wheel in each program. By the end of this tutorial, you'll have improved on this hand-crafted solution and learned a few better methods.

A Few Methods for Validating Python Command Line Arguments

You've already performed validation for Python command line arguments in a few examples like `seq_regex.py` and `seq_parse.py`. In the first example, you used a regular expression, and in the second example, a custom parser.

Both of these examples took the same aspects into account. They considered the expected **options** as short-form (`-s`) or long-form (`--separator`). They considered the **order** of the arguments so that options would not be placed after **operands**. Finally, they considered the type, integer for the operands, and the number of arguments, from one to three arguments.

Type Validation With Python Data Classes

The following is a proof of concept that attempts to validate the type of the arguments passed at the command line. In the following example, you validate the number of arguments and their respective type:

Python

```
# val_type_dc.py

import dataclasses
import sys
from typing import List, Any

USAGE = f"Usage: python {sys.argv[0]} [--help] | firstname lastname age]"

@dataclasses.dataclass
class Arguments:
    firstname: str
    lastname: str
    age: int = 0

def check_type(obj):
    for field in dataclasses.fields(obj):
        value = getattr(obj, field.name)
        print(
            f"Value: {value}, "
            f"Expected type {field.type} for {field.name}, "
            f"got {type(value)}"
        )
        if type(value) != field.type:
            print("Type Error")
        else:
            print("Type Ok")

def validate(args: List[str]):
    # If passed to the command line, need to convert
    # the optional 3rd argument from string to int
    if len(args) > 2 and args[2].isdigit():
        args[2] = int(args[2])
    try:
        arguments = Arguments(*args)
    except TypeError:
        raise SystemExit(USAGE)
    check_type(arguments)

def main() -> None:
    args = sys.argv[1:]
    if not args:
        raise SystemExit(USAGE)

    if args[0] == "--help":
        print(USAGE)
    else:
        validate(args)
```

```
if __name__ == "__main__":
    main()
```

Unless you pass the `--help` option at the command line, this script expects two or three arguments:

1. **A mandatory string:** `firstname`
2. **A mandatory string:** `lastname`
3. **An optional integer:** `age`

Because all the items in `sys.argv` are strings, you need to convert the optional third argument to an integer if it's composed of digits. `str.isdigit()` validates if all the characters in a string are digits. In addition, by constructing the **data class** `Arguments` with the values of the converted arguments, you obtain two validations:

1. **If the number of arguments** doesn't correspond to the number of mandatory fields expected by `Arguments`, then you get an error. This is a minimum of two and a maximum of three fields.
2. **If the types after conversion** aren't matching the types defined in the `Arguments` data class definition, then you get an error.

You can see this in action with the following execution:

Shell

```
$ python val_type_dc.py Guido "Van Rossum" 25
Value: Guido, Expected type <class 'str'> for firstname, got <class 'str'>
Type Ok
Value: Van Rossum, Expected type <class 'str'> for lastname, got <class 'str'>
Type Ok
Value: 25, Expected type <class 'int'> for age, got <class 'int'>
Type Ok
```

In the execution above, the number of arguments is correct and the type of each argument is also correct.

Now, execute the same command but omit the third argument:

Shell

```
$ python val_type_dc.py Guido "Van Rossum"
Value: Guido, Expected type <class 'str'> for firstname, got <class 'str'>
Type Ok
Value: Van Rossum, Expected type <class 'str'> for lastname, got <class 'str'>
Type Ok
Value: 0, Expected type <class 'int'> for age, got <class 'int'>
Type Ok
```

The result is also successful because the field `age` is defined with a **default value**, `0`, so the data class `Arguments` doesn't require it.

On the contrary, if the third argument isn't of the proper type—say, a string instead of integer—then you get an error:

Shell

```
python val_type_dc.py Guido Van Rossum
Value: Guido, Expected type <class 'str'> for firstname, got <class 'str'>
Type Ok
Value: Van, Expected type <class 'str'> for lastname, got <class 'str'>
Type Ok
Value: Rossum, Expected type <class 'int'> for age, got <class 'str'>
Type Error
```

The expected value `Van Rossum`, isn't surrounded by quotes, so it's split. The second word of the last name, `Rossum`, is a string that's handled as the age, which is expected to be an `int`. The validation fails.

> **Note:** For more details about the usage of data classes in Python, check out The Ultimate Guide to Data Classes in Python 3.7.

Similarly, you could also use a `NamedTuple` to achieve a similar validation. You'd replace the data class with a class deriving from `NamedTuple`, and `check_type()` would change as follows:

Python

```
from typing import NamedTuple

class Arguments(NamedTuple):
    firstname: str
    lastname: str
    age: int = 0

def check_type(obj):
    for attr, value in obj._asdict().items():
        print(
            f"Value: {value}, "
            f"Expected type {obj.__annotations__[attr]} for {attr}, "
            f"got {type(value)}"
        )
        if type(value) != obj.__annotations__[attr]:
            print("Type Error")
        else:
            print("Type Ok")
```

A `NamedTuple` exposes functions like `_asdict` that transform the object into a dictionary that can be used for data lookup. It also exposes attributes like `__annotations__`, which is a dictionary storing types for each field, and For more on `__annotations__`, check out Python Type Checking (Guide).

As highlighted in Python Type Checking (Guide), you could also leverage existing packages like Enforce, Pydantic, and Pytypes for advanced validation.

Custom Validation

Not unlike what you've already explored earlier, detailed validation may require some custom approaches. For example, if you attempt to execute `sha1sum_stdin.py` with an incorrect file name as an argument, then you get the following:

Shell

```
$ python sha1sum_stdin.py bad_file.txt
Traceback (most recent call last):
  File "sha1sum_stdin.py", line 32, in <module>
    main(sys.argv[1:])
  File "sha1sum_stdin.py", line 29, in main
    output_sha1sum(process_file(arg), arg)
  File "sha1sum_stdin.py", line 9, in process_file
    return pathlib.Path(filename).read_bytes()
  File "/usr/lib/python3.8/pathlib.py", line 1222, in read_bytes
    with self.open(mode='rb') as f:
  File "/usr/lib/python3.8/pathlib.py", line 1215, in open
    return io.open(self, mode, buffering, encoding, errors, newline,
  File "/usr/lib/python3.8/pathlib.py", line 1071, in _opener
    return self._accessor.open(self, flags, mode)
FileNotFoundError: [Errno 2] No such file or directory: 'bad_file.txt'
```

`bad_file.txt` doesn't exist, but the program attempts to read it.

Revisit `main()` in `sha1sum_stdin.py` to handle non-existing files passed at the command line:

Python

```
1  def main(args):
2      if not args:
3          output_sha1sum(process_stdin())
4      for arg in args:
5          if arg == "-":
6              output_sha1sum(process_stdin(), "-")
7              continue
8          try:
9              output_sha1sum(process_file(arg), arg)
10         except FileNotFoundError as err:
11             print(f"{sys.argv[0]}: {arg}: {err.strerror}", file=sys.stderr)
```

To see the complete example with this extra validation, expand the code block below:

Complete Source Code of sha1sum_val.py Show/Hide

When you execute this modified script, you get this:

Shell

```
$ python sha1sum_val.py bad_file.txt
sha1sum_val.py: bad_file.txt: No such file or directory
```

Note that the error displayed to the terminal is written to `stderr`, so it doesn't interfere with the data expected by a command that would read the output of `sha1sum_val.py`:

Shell

```
$ python sha1sum_val.py bad_file.txt main.py | cut -d " " -f 1
sha1sum_val.py: bad_file.txt: No such file or directory
d84372fc77a90336b6bb7c5e959bcb1b24c608b4
```

This command pipes the output of `sha1sum_val.py` to `cut` to only include the first field. You can see that `cut` ignores the error message because it only receives the data sent to `stdout`.

The Python Standard Library

Despite the different approaches you took to process Python command line arguments, any complex program might be better off **leveraging existing libraries** to handle the heavy lifting required by sophisticated command line interfaces. As of Python 3.7, there are three command line parsers in the standard library:

1. `argparse`
2. `getopt`
3. `optparse`

The recommended module to use from the standard library is `argparse`. The standard library also exposes `optparse` but it's officially deprecated and only mentioned here for your information. It was superseded by `argparse` in Python 3.2 and you won't see it discussed in this tutorial.

`argparse`

You're going to revisit `sha1sum_val.py`, the most recent clone of `sha1sum`, to introduce the benefits of `argparse`. To this effect, you'll modify `main()` and add `init_argparse` to instantiate `argparse.ArgumentParser`:

Python

```python
import argparse

def init_argparse() -> argparse.ArgumentParser:
    parser = argparse.ArgumentParser(
        usage="%(prog)s [OPTION] [FILE]...",
        description="Print or check SHA1 (160-bit) checksums."
    )
    parser.add_argument(
        "-v", "--version", action="version",
        version = f"{parser.prog} version 1.0.0"
    )
    parser.add_argument('files', nargs='*')
    return parser

def main() -> None:
    parser = init_argparse()
    args = parser.parse_args()
    if not args.files:
        output_sha1sum(process_stdin())
    for file in args.files:
        if file == "-":
```

```
            output_sha1sum(process_stdin(), "-")
            continue
        try:
            output_sha1sum(process_file(file), file)
        except (FileNotFoundError, IsADirectoryError) as err:
            print(f"{sys.argv[0]}: {file}: {err.strerror}", file=sys.stderr)
```

For the cost of a few more lines compared to the previous implementation, you get a clean approach to add `--help` and `--version` options that didn't exist before. The expected arguments (the files to be processed) are all available in field `files` of object `argparse.Namespace`. This object is populated on line 17 by calling `parse_args()`.

To look at the full script with the modifications described above, expand the code block below:

Complete Source Code of sha1sum_argparse.py Show/Hide

To illustrate the immediate benefit you obtain by introducing `argparse` in this program, execute the following:

Shell

```
$ python sha1sum_argparse.py --help
usage: sha1sum_argparse.py [OPTION] [FILE]...

Print or check SHA1 (160-bit) checksums.

positional arguments:
  files

optional arguments:
  -h, --help     show this help message and exit
  -v, --version  show program's version number and exit
```

To delve into the details of `argparse`, check out How to Build Command Line Interfaces in Python With argparse.

getopt

`getopt` finds its origins in the `getopt` C function. It facilitates parsing the command line and handling options, option arguments, and arguments. Revisit `parse` from `seq_parse.py` to use `getopt`:

Python

```
def parse():
    options, arguments = getopt.getopt(
        sys.argv[1:],                     # Arguments
        'vhs:',                           # Short option definitions
        ["version", "help", "separator="]) # Long option definitions
    separator = "\n"
    for o, a in options:
        if o in ("-v", "--version"):
            print(VERSION)
            sys.exit()
        if o in ("-h", "--help"):
            print(USAGE)
            sys.exit()
        if o in ("-s", "--separator"):
            separator = a
    if not arguments or len(arguments) > 3:
        raise SystemExit(USAGE)
    try:
        operands = [int(arg) for arg in arguments]
    except ValueError:
        raise SystemExit(USAGE)
    return separator, operands
```

`getopt.getopt()` takes the following arguments:

1. The usual arguments list minus the script name, `sys.argv[1:]`

2. A string defining the short options
3. A list of strings for the long options

Note that a short option followed by a colon (`:`) expects an option argument, and that a long option trailed with an equals sign (`=`) expects an option argument.

The remaining code of `seq_getopt.py` is the same as `seq_parse.py` and is available in the collapsed code block below:

Complete Source Code of seq_getopt.py Show/Hide

Next, you'll take a look at some external packages that will help you parse Python command line arguments.

A Few External Python Packages

Building upon the existing conventions you saw in this tutorial, there are a few libraries available on the Python Package Index (PyPI) that take many more steps to facilitate the implementation and maintenance of command line interfaces.

The following sections offer a glance at Click and Python Prompt Toolkit. You'll only be exposed to very limited capabilities of these packages, as they both would require a full tutorial—if not a whole series—to do them justice!

Click

As of this writing, **Click** is perhaps the most advanced library to build a sophisticated command line interface for a Python program. It's used by several Python products, most notably Flask and Black. Before you try the following example, you need to install Click in either a Python virtual environment or your local environment. If you're not familiar with the concept of virtual environments, then check out Python Virtual Environments: A Primer.

To install Click, proceed as follows:

Shell

```
$ python -m pip install click
```

So, how could Click help you handle the Python command line arguments? Here's a variation of the `seq` program using Click:

Python

```
# seq_click.py

import click

@click.command(context_settings=dict(ignore_unknown_options=True))
@click.option("--separator", "-s",
              default="\n",
              help="Text used to separate numbers (default: \\n)")
@click.version_option(version="1.0.0")
@click.argument("operands", type=click.INT, nargs=-1)
def seq(operands, separator) -> str:
    first, increment, last = 1, 1, 1
    if len(operands) == 1:
        last = operands[0]
    elif len(operands) == 2:
        first, last = operands
        if first > last:
            increment = -1
    elif len(operands) == 3:
        first, increment, last = operands
    else:
        raise click.BadParameter("Invalid number of arguments")
    last = last - 1 if first > last else last + 1
    print(separator.join(str(i) for i in range(first, last, increment)))

if __name__ == "__main__":
    seq()
```

Setting `ignore_unknown_options` to `True` ensures that Click doesn't parse negative arguments as options. Negative integers are valid `seq` arguments.

As you may have observed, you get a lot for free! A few well-carved decorators are sufficient to bury the boilerplate code, allowing you to focus on the main code, which is the content of `seq()` in this example.

> **Note:** For more about Python decorators, check out Primer on Python Decorators.

The only import remaining is `click`. The declarative approach of decorating the main command, `seq()`, eliminates repetitive code that's otherwise necessary. This could be any of the following:

- **Defining** a help or usage procedure
- **Handling** the version of the program
- **Capturing** and **setting up** default values for options
- **Validating** arguments, including the type

The new `seq` implementation barely scratches the surface. Click offers many niceties that will help you craft a very professional command line interface:

- Output coloring
- Prompt for omitted arguments
- Commands and sub-commands
- Argument type validation
- Callback on options and arguments
- File path validation
- Progress bar

There are many other features as well. Check out Writing Python Command-Line Tools With Click to see more concrete examples based on Click.

Python Prompt Toolkit

There are other popular Python packages that are handling the command line interface problem, like docopt for Python. So, you may find the choice of the Prompt Toolkit a bit counterintuitive.

The **Python Prompt Toolkit** provides features that may make your command line application drift away from the Unix philosophy. However, it helps to bridge the gap between an arcane command line interface and a full-fledged graphical user interface. In other words, it may help to make your tools and programs more user-friendly.

You can use this tool in addition to processing Python command line arguments as in the previous examples, but this gives you a path to a UI-like approach without you having to depend on a full Python UI toolkit. To use `prompt_toolkit`, you need to install it with `pip`:

Shell

```
$ python -m pip install prompt_toolkit
```

You may find the next example a bit contrived, but the intent is to spur ideas and move you slightly away from more rigorous aspects of the command line with respect to the conventions you've seen in this tutorial.

As you've already seen the core logic of this example, the code snippet below only presents the code that significantly deviates from the previous examples:

Python

```
def error_dlg():
    message_dialog(
        title="Error",
        text="Ensure that you enter a number",
    ).run()
```

```
def seq_dlg():
    labels = ["FIRST", "INCREMENT", "LAST"]
    operands = []
    while True:
        n = input_dialog(
            title="Sequence",
            text=f"Enter argument {labels[len(operands)]}:",
        ).run()
        if n is None:
            break
        if n.isdigit():
            operands.append(int(n))
        else:
            error_dlg()
        if len(operands) == 3:
            break

    if operands:
        seq(operands)
    else:
        print("Bye")

actions = {"SEQUENCE": seq_dlg, "HELP": help, "VERSION": version}

def main():
    result = button_dialog(
        title="Sequence",
        text="Select an action:",
        buttons=[
            ("Sequence", "SEQUENCE"),
            ("Help", "HELP"),
            ("Version", "VERSION"),
        ],
    ).run()
    actions.get(result, lambda: print("Unexpected action"))()
```

The code above involves ways to interact and possibly guide users to enter the expected input, and to validate the input interactively using three dialog boxes:

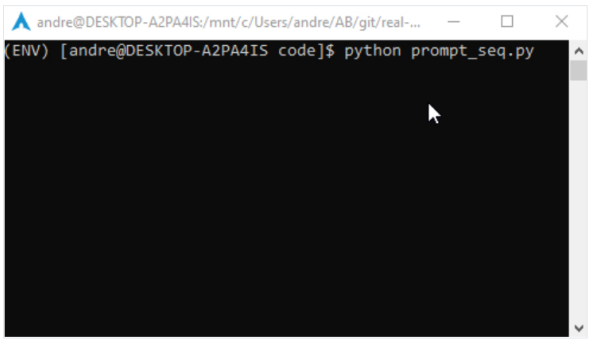
1. `button_dialog`
2. `message_dialog`
3. `input_dialog`

The Python Prompt Toolkit exposes many other features intended to improve interaction with users. The call to the handler in `main()` is triggered by calling a function stored in a dictionary. Check out Emulating switch/case Statements in Python if you've never encountered this Python idiom before.

You can see the full example of the program using `prompt_toolkit` by expanding the code block below:

Complete Source Code for seq_prompt.py Show/Hide

When you execute the code above, you're greeted with a dialog prompting you for action. Then, if you choose the action *Sequence*, another dialog box is displayed. After collecting all the necessary data, options, or arguments, the dialog box disappears, and the result is printed at the command line, as in the previous examples:



As the command line evolves and you can see some attempts to interact with users more creatively, other packages like PyInquirer also allow you to capitalize on a very interactive approach.

To further explore the world of the **Text-Based User Interface (TUI)**, check out Building Console User Interfaces and the Third Party section in Your Guide to the Python Print Function.

If you're interested in researching solutions that rely exclusively on the graphical user interface, then you may consider checking out the following resources:

- How to Build a Python GUI Application With wxPython
- Python and PyQt: Building a GUI Desktop Calculator
- Build a Mobile Application With the Kivy Python Framework

Conclusion

In this tutorial, you've navigated many different aspects of Python command line arguments. You should feel prepared to apply the following skills to your code:

- The **conventions and pseudo-standards** of Python command line arguments
- The **origins** of `sys.argv` in Python
- The **usage** of `sys.argv` to provide flexibility in running your Python programs
- The **Python standard libraries** like `argparse` or `getopt` that abstract command line processing
- The **powerful Python packages** like `click` and `python_toolkit` to further improve the usability of your programs

Whether you're running a small script or a complex text-based application, when you expose a **command line interface** you'll significantly improve the user experience of your Python software. In fact, you're probably one of those users!

Next time you use your application, you'll appreciate the documentation you supplied with the `--help` option or the fact that you can pass options and arguments instead of modifying the source code to supply different data.

Additional Resources

To gain further insights about Python command line arguments and their many facets, you may want to check out the following resources:

- Comparing Python Command-Line Parsing Libraries – Argparse, Docopt, and Click
- Python, Ruby, and Golang: A Command-Line Application Comparison

You may also want to try other Python libraries that target the same problems while providing you with different solutions:

- Typer
- Plac
- Cliff
- Cement
- Python Fire

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Andre Burgaud**

Andre is a seasoned software engineer passionate about technology and programming languages, in particular, Python.

» More about Andre

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

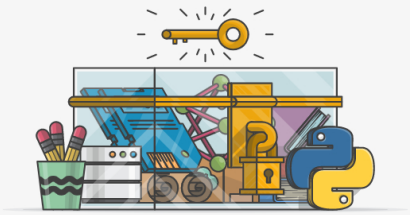
 Aldren

 Geir Arne

 Jaya

 Joanna

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written

with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Keep Learning

Related Tutorial Categories: best-practices intermediate tools

Recommended Video Course: Command Line Interfaces in Python