# Python Command Line Arguments

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](https://realpython.com/read-write-files-python/), 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!

**Free Bonus:** [**5 Thoughts On Python Mastery**](https://realpython.com/bonus/python-mastery-course/), a free course for Python developers that shows you the roadmap and the mindset you’ll need to take your Python skills to the next level.

**The Command Line Interface**

A [**command line interface (CLI)**](https://en.wikipedia.org/wiki/Command-line_interface) provides a way for a user to interact with a program running in a text-based [shell](https://en.wikipedia.org/wiki/Shell_%28computing%29) interpreter. Some examples of shell interpreters are [Bash](https://en.wikipedia.org/wiki/Bash_%28Unix_shell%29) on Linux or [Command Prompt](https://en.wikipedia.org/wiki/Cmd.exe) on Windows. A command line interface is enabled by the shell interpreter that exposes a [command prompt](https://en.wikipedia.org/wiki/Command-line_interface#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](https://en.wikipedia.org/wiki/Domain-specific_language). For example, some programs may launch web documentation from the command line or start an [interactive shell interpreter](https://docs.python.org/tutorial/interpreter.html#interactive-mode) like Python.

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

$ 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:

$ 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.

[Remove ads](https://realpython.com/account/join/)

**The C Legacy**

Python command line arguments directly inherit from the [C](https://realpython.com/build-python-c-extension-module/) programming language. As [Guido Van Rossum](https://en.wikipedia.org/wiki/Guido_van_Rossum) wrote in [An Introduction to Python for Unix/C Programmers](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.4180) 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:

1// main.c

2#include <stdio.h>

3

4int main(int argc, char \*argv[]) {

5 printf("Arguments count: %d\n", argc);

6 for (int i = 0; i < argc; i++) {

7 printf("Argument %6d: %s\n", i, argv[i]);

8 }

9 return 0;

10}

Line 4 defines [main()](https://en.wikipedia.org/wiki/Entry_point#C_and_C++), 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:

$ 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](https://en.wikipedia.org/wiki/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:

$ ./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](https://en.wikibooks.org/wiki/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):**](https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux) It’s available in a few Linux distributions, like [Ubuntu](https://ubuntu.com/), [OpenSUSE](https://www.opensuse.org/), and [Debian](https://www.debian.org/), among others. You can install it from the Microsoft Store.
* [**Windows Build Tools:**](https://visualstudio.microsoft.com/downloads/#build-tools-for-visual-studio-2019) This includes the Windows command line build tools, the Microsoft C/C++ compiler [cl.exe](https://docs.microsoft.com/en-us/cpp/build/walkthrough-compiling-a-cpp-cli-program-on-the-command-line?view=vs-2019), and a compiler front end named [clang.exe](https://en.wikipedia.org/wiki/Clang) for C/C++.
* [**Microsoft Visual Studio:**](https://visualstudio.microsoft.com/downloads/) 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)](https://realpython.com/python-ides-code-editors-guide/).
* [**mingw-64 project:**](http://mingw-w64.org/) This supports the [GCC compiler](https://gcc.gnu.org/) on Windows.

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

C:/>cl main.c

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

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:

# 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](https://realpython.com/python-variables/) 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()](https://docs.python.org/library/functions.html#len) if the number of arguments is needed by your program.

Also, note that [enumerate()](https://realpython.com/python-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:

$ 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.

## 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](https://docs.python.org/library/sys.html?highlight=sys%20argv#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:

1# argv.py

2import sys

3

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

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

Here’s how this code works:

* **Line 2** imports the internal Python module [sys](https://docs.python.org/library/sys.html).
* **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](https://realpython.com/python-f-strings/) 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](https://realpython.com/python38-new-features/).

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:

$ 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:

1# reverse.py

2

3import sys

4

5arg = sys.argv[1]

6print(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](https://stackoverflow.com/questions/3705670/best-way-to-create-a-reversed-list-in-python/3705676#3705676).

You execute the script as follows:

$ 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](https://realpython.com/python-command-line-arguments/#escaping-whitespace-characters).

### **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:

# argv\_pop.py

import sys

print(sys.argv)

sys.argv.pop()

print(sys.argv)

You invoke [.pop()](https://docs.python.org/tutorial/datastructures.html#more-on-lists) to remove and return the last item in sys.argv.

Execute the script above:

$ 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:

# 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](https://realpython.com/courses/what-is-pip/), uses this [approach](https://github.com/pypa/pip/blob/ce46f8524e194ea81c47dbf8a547698f12e61329/src/pip/_internal/__init__.py#L53). Here’s a short excerpt of the pip source code:

def main(args=None):

if args is None:

args = sys.argv[1:]

In this snippet of code taken from the [pip](https://realpython.com/what-is-pip/) source code, [main()](https://realpython.com/python-main-function/) 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.

### **scaping Whitespace Characters**

In the reverse.py example you saw [earlier](https://realpython.com/python-command-line-arguments/#reversing-the-first-argument), 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]:

$ 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:

$ 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)](https://en.wikipedia.org/wiki/Internal_field_separator) defines characters used as **delimiters**. The content of the shell variable, IFS, can be displayed by running the following command:

$ 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:

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:

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](http://www.windowsinspired.com/understanding-the-command-line-string-and-arguments-received-by-a-windows-program/).

### **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:

1$ python reverse.py

2Traceback (most recent call last):

3 File "reverse.py", line 5, in <module>

4 arg = sys.argv[1]

5IndexError: list index out of range

The Python [exception](https://realpython.com/python-exceptions/) IndexError is raised, and the corresponding [traceback](https://realpython.com/python-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:

1# reverse\_exc.py

2

3import sys

4

5try:

6 arg = sys.argv[1]

7except IndexError:

8 raise SystemExit(f"Usage: {sys.argv[0]} <string\_to\_reverse>")

9print(arg[::-1])

The expression on line 4 is included in a try block. Line 8 raises the built-in exception [SystemExit](https://docs.python.org/library/exceptions.html#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:

$ 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 [$?](https://en.wikipedia.org/wiki/Bash_(Unix_shell)#Conditional_execution) with [echo](https://en.wikipedia.org/wiki/Echo_%28command%29).

### **Windows**

On Windows, the conventions regarding Python command line arguments are slightly different, in particular, those regarding [command line options](https://en.wikipedia.org/wiki/Command-line_interface#Option_conventions_in_DOS,_Windows,_OS/2). 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:

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, taskslist, and the option /FI, but it’s just as correct to type taskslist/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:

$ 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](https://en.wikipedia.org/wiki/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](https://docs.python.org/3/using/cmdline.html#cmdoption-o) is a means to optimize the execution of a Python program by removing assert and \_\_debug\_\_ statements. There are other [Python options](https://docs.python.org/using/cmdline.html#interface-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.

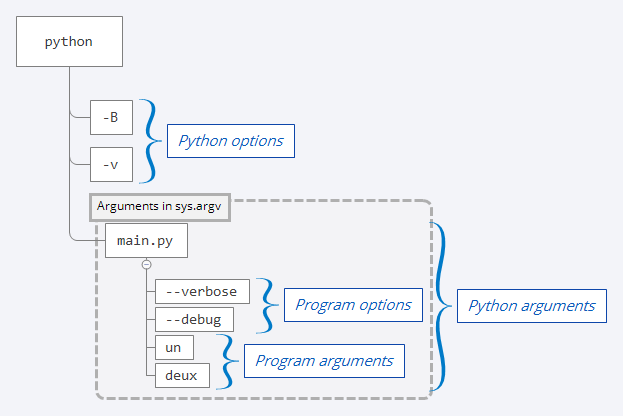
Take the following command that’s intended to execute the program main.py, which takes options and arguments. Note that, in this example, the Python interpreter also takes some options, which are [-B](https://docs.python.org/using/cmdline.html#id1) and [-v](https://docs.python.org/using/cmdline.html#id4).

$ 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?](https://realpython.com/cpython-source-code-guide/#what-does-a-compiler-do) in [Your Guide to the CPython Source Code](https://realpython.com/cpython-source-code-guide).
* **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:

[](https://files.realpython.com/media/python_blue.37b9170f4345.png)

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](https://realpython.com/list-comprehension-python/) 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:

doesn’t handle option-arguments, may be enough:

# 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](https://realpython.com/list-comprehension-python/):

* **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:

$ 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](https://realpython.com/regex-python/) 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](https://en.wikipedia.org/wiki/Seq_%28Unix%29), which is a program that prints a sequence of numbers. Following the docopt conventions, a specification for seq.py could be this:

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:

1args\_pattern = re.compile(

2 r"""

3 ^

4 (

5 (--(?P<HELP>help).\*)|

6 ((?:-s|--separator)\s(?P<SEP>.\*?)\s)?

7 ((?P<OP1>-?\d+))(\s(?P<OP2>-?\d+))?(\s(?P<OP3>-?\d+))?

8 )

9 $

10""",

11 re.VERBOSE,

12)

To experiment with the regular expression above, you may use the snippet recorded on [Regular Expression 101](https://regex101.com/r/Wh2Rce/3). 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](https://www.regular-expressions.info/named.html) 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](https://docs.python.org/library/re.html#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()](https://docs.python.org/3/library/stdtypes.html#str.join):

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:

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()](https://docs.python.org/library/re.html#re.match) to the argument line to extract the proper values and store the data in a dictionary.

The [dictionary](https://realpython.com/courses/dictionaries-python/) 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 ExpressionsShow/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:

# 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](https://docs.python.org/glossary.html#term-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](https://docs.python.org/library/stdtypes.html#bytes). For more information about handling file content, check out [Reading and Writing Files in Python](https://realpython.com/read-write-files-python), and in particular, the section [Working With Bytes](https://realpython.com/read-write-files-python/#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:

$ sha1sum main main.c

9a6f82c245f5980082dbf6faac47e5085083c07d main

125a0f900ff6f164752600550879cbfabb098bc3 main.c

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

$ 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:

$ 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](https://docs.python.org/library/glob.html). 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:

1# main\_win.py

2

3import sys

4import glob

5import itertools

6from typing import List

7

8def expand\_args(args: List[str]) -> List[str]:

9 arguments = args[:1]

10 glob\_args = [glob.glob(arg) for arg in args[1:]]

11 arguments += itertools.chain.from\_iterable(glob\_args)

12 return arguments

13

14if \_\_name\_\_ == "\_\_main\_\_":

15 args = expand\_args(sys.argv)

16 print(f"Arguments count: {len(args)}")

17 for i, arg in enumerate(args):

18 print(f"Argument {i:>6}: {arg}")

In main\_win.py, expand\_args relies on [glob.glob()](https://docs.python.org/library/glob.html#glob.glob) to process the shell-style wildcards. You can verify the result on Windows and any other operating system:

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](https://docs.python.org/library/sys.html#sys.stdin), you’ll get closer to the original sha1sum:

# 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](https://www.python.org/dev/peps/pep-0020/), then complete the entry with the keyboard shortcut Ctrl+D on Unix-like systems or Ctrl+Z on Windows:

$ 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:

$ 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:

$ 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](https://realpython.com/python-command-line-arguments/#a-few-methods-for-validating-python-command-line-arguments).

### **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](https://en.wikipedia.org/wiki/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:

$ 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:

# 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:

$ 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:

$ 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:

# 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:

$ 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:

1def parse(args: List[str]) -> Tuple[str, List[int]]:

2 arguments = collections.deque(args)

3 separator = "\n"

4 operands: List[int] = []

5 while arguments:

6 arg = arguments.popleft()

7 if not operands:

8 if arg == "--help":

9 print(USAGE)

10 sys.exit(0)

11 if arg in ("-s", "--separator"):

12 separator = arguments.popleft()

13 continue

14 try:

15 operands.append(int(arg))

16 except ValueError:

17 raise SystemExit(USAGE)

18 if len(operands) > 3:

19 raise SystemExit(USAGE)

20

21 return separator, operands

parse() is given the list of arguments without the Python file name and uses [collections.deque()](https://docs.python.org/library/collections.html#collections.deque) to get the benefit of [.popleft()](https://docs.python.org/library/collections.html#collections.deque.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:

# 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()](https://docs.python.org/3/library/stdtypes.html#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:

$ 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:

$ 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:

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](https://realpython.com/python-data-classes/).

Similarly, you could also use a [NamedTuple](https://docs.python.org/3/library/typing.html#typing.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:

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)](https://realpython.com/python-type-checking/#annotations).

As highlighted in [Python Type Checking (Guide)](https://realpython.com/python-type-checking/#using-types-at-runtime), you could also leverage existing packages like [Enforce](https://pypi.org/project/enforce/), [Pydantic](https://pydantic-docs.helpmanual.io/), and [Pytypes](https://pypi.org/project/pytypes/) for advanced validation.

### **Custom Validation**

Not unlike what you’ve already explored [earlier](https://realpython.com/python-command-line-arguments/#custom-parsers), 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:

$ 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:

1def 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.pyShow/Hide

When you execute this modified script, you get this:

$ 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:

$ 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](https://en.wikipedia.org/wiki/Cut_%28Unix%29) 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](https://docs.python.org/library/argparse.html)
2. [getopt](https://docs.python.org/library/getopt.html)
3. [optparse](https://docs.python.org/library/optparse.html)

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:

1import argparse

2

3def init\_argparse() -> argparse.ArgumentParser:

4 parser = argparse.ArgumentParser(

5 usage="%(prog)s [OPTION] [FILE]...",

6 description="Print or check SHA1 (160-bit) checksums."

7 )

8 parser.add\_argument(

9 "-v", "--version", action="version",

10 version = f"{parser.prog} version 1.0.0"

11 )

12 parser.add\_argument('files', nargs='\*')

13 return parser

14

15def main() -> None:

16 parser = init\_argparse()

17 args = parser.parse\_args()

18 if not args.files:

19 output\_sha1sum(process\_stdin())

20 for file in args.files:

21 if file == "-":

22 output\_sha1sum(process\_stdin(), "-")

23 continue

24 try:

25 output\_sha1sum(process\_file(file), file)

26 except (FileNotFoundError, IsADirectoryError) as err:

27 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](https://docs.python.org/library/argparse.html#argparse.Namespace). This object is populated on line 17 by calling [parse\_args()](https://docs.python.org/library/argparse.html#argparse.ArgumentParser.parse_args).

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

Complete Source Code of sha1sum\_argparse.pyShow/Hide

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

$ 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](https://realpython.com/command-line-interfaces-python-argparse/).

### **getopt**

getopt finds its origins in the [getopt](https://en.wikipedia.org/wiki/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:

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()](https://docs.python.org/library/getopt.html#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.pyShow/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)](https://pypi.org/) that take many more steps to facilitate the implementation and maintenance of command line interfaces.

The following sections offer a glance at [Click](https://click.palletsprojects.com/en/7.x/) and [Python Prompt Toolkit](https://python-prompt-toolkit.readthedocs.io/en/master/index.html). 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](https://palletsprojects.com/p/flask/) and [Black](https://black.readthedocs.io/en/stable/). Before you try the following example, you need to install Click in either a [Python virtual environment](https://docs.python.org/tutorial/venv.html) or your local environment. If you’re not familiar with the concept of virtual environments, then check out [Python Virtual Environments: A Primer](https://realpython.com/python-virtual-environments-a-primer/).

To install Click, proceed as follows:

$ 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:

# 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](https://click.palletsprojects.com/en/7.x/api/#click.Context.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](https://www.python.org/dev/peps/pep-0318/) 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](https://realpython.com/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](https://dbader.org/blog/python-commandline-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](https://github.com/docopt/docopt). So, you may find the choice of the [Prompt Toolkit](https://python-prompt-toolkit.readthedocs.io/en/master/index.html) 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](https://realpython.com/python-gui-with-wxpython/). 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](https://wiki.python.org/moin/GuiProgramming). To use prompt\_toolkit, you need to install it with pip:

$ 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:

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](https://realpython.com/courses/emulating-switch-case-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.pyShow/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](https://github.com/CITGuru/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](https://realpython.com/python-print/#building-console-user-interfaces) and the [Third Party section](https://realpython.com/python-print/#third-party) in [Your Guide to the Python Print Function](https://realpython.com/python-print/).

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](https://realpython.com/python-gui-with-wxpython/)
* [Python and PyQt: Building a GUI Desktop Calculator](https://realpython.com/python-pyqt-gui-calculator/)
* [Build a Mobile Application With the Kivy Python Framework](https://realpython.com/mobile-app-kivy-python/)

## 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](https://realpython.com/comparing-python-command-line-parsing-libraries-argparse-docopt-click/)
* [Python, Ruby, and Golang: A Command-Line Application Comparison](https://realpython.com/python-ruby-and-golang-a-command-line-application-comparison/)
* [Python Command Line Arguments – Real Python](https://realpython.com/python-command-line-arguments/) (this document you are reading)

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

* [Typer](https://typer.tiangolo.com/)
* [Plac](http://micheles.github.io/plac/)
* [Cliff](https://docs.openstack.org/cliff/latest/)
* [Cement](https://builtoncement.com/)
* [Python Fire](https://github.com/google/python-fire/)