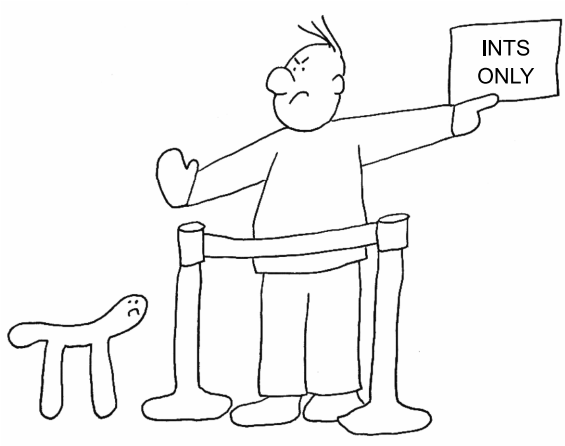
appendix
Using argparse

Often, getting the right data into your program is a real chore. The `argparse` module makes it much easier to validate arguments from users and to generate useful error messages when they provide bad input. It's like your program's "bouncer," only allowing the right kinds of values into the program. Defining the arguments properly with `argparse` is the crucial first step to making the programs in this book work.



For instance, chapter 1 discusses a very flexible program that can extend warm salutations to an optionally named entity, such as the "World" or "Universe":

```
$ ./hello.py
Hello, World!
$ ./hello.py --name Universe
Hello, Universe!
```

When the program runs with no input values, it will use "World" for the entity to greet.

The program can take an optional --name value to override the default.

The program will respond to the `-h` and `--help` flags with helpful documentation:

```
$ ./hello.py -h
usage: hello.py [-h] [-n str]
```

The argument to the program is -h, which is the "short" flag to ask for help.

This line shows a summary of all the options the program accepts. The square brackets [] around the arguments show that they are optional.

```
Say hello

optional arguments:
  -h, --help            show this help message and exit
  -n str, --name str    The name to greet (default: World)
```

This is the description of the program.

We can use either the "short" name -h or the "long" name --help to ask the program for help on how to run it.

The optional "name" parameter also has short and long names of -n and --name.

All of this is created by just two lines of code in the hello.py program:

```
parser = argparse.ArgumentParser(description='Say hello')
parser.add_argument('-n', '--name', default='World', help='Name to greet')
```

The parser will parse the arguments for us. If the user provides unknown arguments or the wrong number of arguments, the program will halt with a usage statement.

The only argument to this program is an optional --name value.

NOTE You do not need to define the `-h` or `--help` flags. Those are generated automatically by `argparse`. In fact, you should never try to use those for other values because they are almost universal options that most users will expect.

The `argparse` module helps us define a parser for the arguments and generates help messages, saving us loads of time and making our programs look professional. Every program in this book is tested on different inputs, so you'll really understand how to use this module by the end. I recommend you look over the `argparse` documentation (https://docs.python.org/3/library/argparse.html).

Now let's dig further into what this module can do for us. In this appendix, you will

- Learn how to use `argparse` to handle positional parameters, options, and flags
- Set default values for options
- Use `type` to force the user to provide values like numbers or files
- Use `choices` to restrict the values for an option

A.1 Types of arguments

Command-line arguments can be classified as follows:

- *Positional arguments*—The order and number of the arguments is what determines their meaning. Some programs might expect, for instance, a filename as the first argument and an output directory as the second. Positional arguments are generally required (not optional) arguments. Making them optional is difficult—how would you write a program that accepts two or three arguments where the second and third ones are independent and optional? In the first version of hello.py in chapter 1, the name to greet was provided as a positional argument.
- *Named options*—Most command-line programs define a *short* name like `-n` (one dash and a single character) and a *long* name like `--name` (two dashes and a word) followed by some value, like the name in the hello.py program. Named

options allow arguments to be provided in any order—their *position* is not relevant. This makes them the right choice when the user is not required to provide them (they are *options*, after all). It's good to provide reasonable default values for options. When we changed the required positional `name` argument of hello.py to the optional `--name` argument, we used "World" for the default so that the program could run with no input from the user. Note that some other languages, like Java, might define long names with a single dash, like `-jar`.

- *Flags*—A Boolean value like "yes"/"no" or `True`/`False` is indicated by something that starts off looking like a named option, but there is no value after the name; for example, the `-d` or `--debug` flag to turn on debugging. Typically the presence of the flag indicates a `True` value for the argument, and its absence would mean `False`, so `--debug` turns *on* debugging, whereas its absence means it is off.

A.2 Using a template to start a program

It's not easy to remember all the syntax for defining parameters using `argparse`, so I've created a way for you to write new programs from a template that includes this plus some other structure that will make your programs easier to read and run.

One way to start a new program is to use the new.py program. From the top level of the repository, you can execute this command:

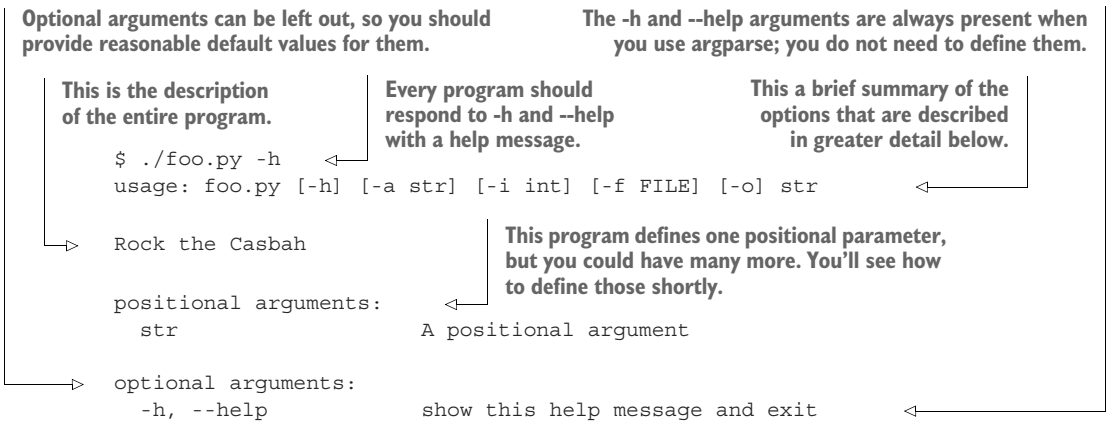
```
$ bin/new.py foo.py
```

Alternatively, you could copy the template:

```
$ cp template/template.py foo.py
```

The resulting program will be identical no matter how you create it, and it will have examples of how to declare each of the argument types outlined in the previous section. Additionally, you can use `argparse` to validate the input, such as making sure that one argument is a number while another argument is a file.

Let's look at the help generated by our new program:



```
  -a str, --arg str     A named string argument (default: )
  -i int, --int int     A named integer argument (default: 0)
  -f FILE, --file FILE  A readable file (default: None)
  -o, --on              A boolean flag (default: False)
```

The -a or --arg option accepts some text, which is often called a "string."

The -i or --int option must be an integer value. If the user provides "one" or "4.2," these will be rejected.

The -o or --on is a flag. Notice how the -f FILE description specifies that a "FILE" value should follow the -f, but for this flag no value follows the option. The flag is either present or absent, and so it's either True or False, respectively.

The -f or --file option must be a valid, readable file.

A.3 Using argparse

The code to generate the preceding usage is found in a function, called `get_args()`, that looks like the following:

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

    parser = argparse.ArgumentParser(
        description='Rock the Casbah',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('positional',
                        metavar='str',
                        help='A positional argument')

    parser.add_argument('-a',
                        '--arg',
                        help='A named string argument',
                        metavar='str',
                        type=str,
                        default='')

    parser.add_argument('-i',
                        '--int',
                        help='A named integer argument',
                        metavar='int',
                        type=int,
                        default=0)

    parser.add_argument('-f',
                        '--file',
                        help='A readable file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        default=None)

    parser.add_argument('-o',
                        '--on',
                        help='A boolean flag',
                        action='store_true')

    return parser.parse_args()
```

You are welcome to put this code wherever you like, but defining and validating the arguments can sometimes get rather long. I like to separate this code out into a function I call `get_args()`, and I always define this function first in my program. That way I can see it immediately when I'm reading the source code.

The `get_args()` function is defined like this:

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

The def keyword defines a new function, and the arguments to the function are listed in the parentheses. Even though the get_args() function takes no arguments, the parentheses are still required.

The triple-quoted line after the function def is the "docstring," which serves as a bit of documentation for the function. Docstrings are not required, but they are good style, and Pylint will complain if you leave them out.

A.3.1 Creating the parser

The following snippet creates a `parser` that will deal with the arguments from the command line. To "parse" here means to derive some meaning from the order and syntax of the bits of text provided as arguments:

```
parser = argparse.ArgumentParser(
    description='Argparse Python script',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)
```

Call the argparse.ArgumentParser() function to create a new parser.

A short summary of your program's purpose.

The formatter_class argument tells argparse to show the default values in usage.

You should read the documentation for `argparse` to see all the other options you can use to define a `parser` or the parameters. In the REPL, you can start with `help(argparse)`, or you could look up the docs on the internet at https://docs.python.org/3/library/argparse.html.

A.3.2 Creating a positional parameter

The following line will create a new *positional* parameter:

```
parser.add_argument('positional',
                    metavar='str',
                    help='A positional argument')
```

The lack of leading dashes makes this a positional parameter, not the name "positional."

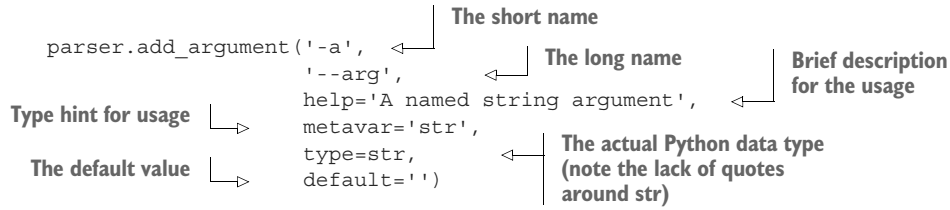
Provide a hint to the user about the data type. By default, all arguments are strings.

A brief description of the parameter for the usage

Remember that the parameter is not positional because the *name* is "positional." That's just there to remind you that it *is* a positional parameter. `argparse` interprets the string `'positional'` as a positional parameter *because the name does not start with any dashes.*

A.3.3 Creating an optional string parameter

The following line creates an *optional* parameter with a short name of `-a` and a long name of `--arg`. It will be a `str` with a default value of `''` (the empty string).

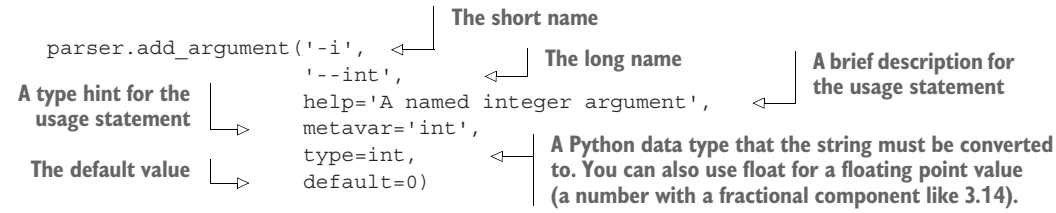


> **NOTE** You can leave out either the short or long name in your own programs, but it's good form to provide both. Most of the tests in this book will test your programs using both short and long option names.

If you wanted to make this a required, named parameter, you would remove the `default` and add `required=True`.

A.3.4 Creating an optional numeric parameter

The following line creates an option called `-i` or `--int` that accepts an `int` (integer) with a default value of `0`. If the user provides anything that cannot be interpreted as an integer, the `argparse` module will stop processing the arguments and will print an error message and a short usage statement.



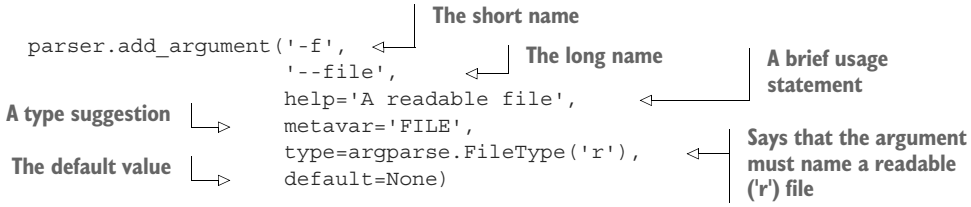
One of the big reasons to define numeric arguments in this way is that `argparse` will convert the input to the correct type. All values coming from the command are strings, and it's the job of the program to convert each value to an actual numeric value. If you tell `argparse` that the option should be `type=int`, it will have already been converted to an actual `int` value when you ask the `parser` for the value.

If the value provided by the user cannot be converted to an `int`, the value will be rejected. Note that you can also use `type=float` to accept and convert the input to a floating-point value. That saves you a lot of time and effort.

A.3.5 Creating an optional file parameter

The following line creates an option called `-f` or `--file` that will only accept a valid, readable file. This argument alone is worth the price of admission, as it will save you oodles of time validating the input from your user. Note that pretty much every exercise

that has a file as input will have tests that pass *invalid* file arguments to ensure that your program rejects them.



The person running the program is responsible for providing the location of the file. For instance, if you created the foo.py program in the top level of the repository, there will be a README.md file there. We could use that as the input to our program, and it would be accepted as a valid argument:

```
$ ./foo.py -f README.md foo
str_arg = ""
int_arg = "0"
file_arg = "README.md"
flag_arg = "False"
positional = "foo"
```

If we provide a bogus `--file` argument, like "blargh," we will get an error message:

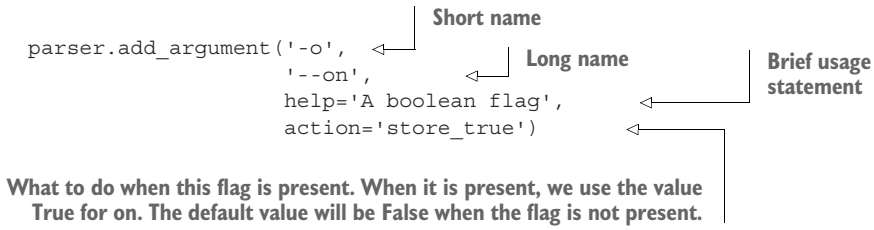
```
$ ./foo.py -f blargh foo
usage: foo.py [-h] [-a str] [-i int] [-f FILE] [-o] str
foo.py: error: argument -f/--file: can't open 'blargh': \
[Errno 2] No such file or directory: 'blargh'
```

A.3.6 Creating a flag option

The flag option is slightly different in that it does not take a value like a string or integer. Flags are either present or not, and they *usually* indicate that some idea is `True` or `False`.

You've already seen the `-h` and `--help` flags. They are not followed by any values. They either are present, in which case the program should print a "usage" statement, or they are absent, in which case the program should not. For all the exercises in this book, I use flags to indicate a `True` value when they are present and `False` otherwise, which we can represent using `action='store_true'`.

For instance, new.py shows an example of this kind of a flag called `-o` or `--on`:



It's not always the case that a "flag" like this should be interpreted as `True` when present. You could instead use `action='store_false'`, in which case `on` would be `False` when the flag is present, and the default value would be `True`. You could also store one or more constant values when the flag is present.

Read the `argparse` documentation for the various ways you can define this parameter. For the purposes of this book, we will only use a flag to turn "on" some behavior.

A.3.7 Returning from get_args

The final statement in `get_args()` is `return`, which returns the result of having the `parser` object parse the arguments. That is, the code that calls `get_args()` will receive the result of this expression:

```
return parser.parse_args()
```

This expression could fail because `argparse` finds that the user provided invalid arguments, such as a string value when it expected a `float` or perhaps a misspelled filename. If the parsing succeeds, we will be able to access all the values the user provided from inside our program.

Additionally, the values of the arguments will be of the *types* that we indicated. That is, if we indicated that the `--int` argument should be an `int`, then when we ask for `args.int`, it will already be an `int`. If we define a file argument, we'll get an *open file handle*. That may not seem impressive now, but it's really enormously helpful.

If you refer to the foo.py program we generated, you'll see that the `main()` function calls `get_args()`, so the `return` from `get_args()` goes back to `main()`. From there, we can access all the values we just defined using the names of the positional parameters or the long names of the optional parameters:

```
def main():
    args = get_args()
    str_arg = args.arg
    int_arg = args.int
    file_arg = args.file
    flag_arg = args.on
    pos_arg = args.positional
```

A.4 Examples using argparse

Many of the program tests in this book can be satisfied by learning how to use `argparse` effectively to validate the arguments to your programs. I think of the command line as the boundary of your program, and you need to be judicious about what you let into your program. You should always expect and defend against every argument being wrong.[1] Our hello.py program in chapter 1 is an example of a single, positional argument and then a single, optional argument. Let's look at some more examples of how you can use `argparse`.

[1] I always think of the kid who will type "fart" for every input.

A.4.1 A single positional argument

This is the first version of chapter 1's hello.py program, which requires a single argument specifying the name to greet:

```
#!/usr/bin/env python3
"""A single positional argument"""

import argparse


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

    parser = argparse.ArgumentParser(
        description='A single positional argument',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('name', metavar='name', help='The name to greet')

    return parser.parse_args()


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

    args = get_args()
    print('Hello, ' + args.name + '!')


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

The name parameter does not start with dashes, so this is a *positional* parameter. The metavar will show up in the help to let the user know what this argument is supposed to be.

Whatever is provided as the first positional argument to the program will be available in the args.name slot.

This program will not print the "Hello" line if it's not provided exactly one argument. If given nothing, it will print a brief usage statement about the proper way to invoke the program:

```
$ ./one_arg.py
usage: one_arg.py [-h] name
one_arg.py: error: the following arguments are required: name
```

If we provide more than one argument, it complains again. Here "Emily" and "Bronte" are two arguments because spaces separate arguments on the command line. The program complains about getting a second argument that has not been defined:

```
$ ./one_arg.py Emily Bronte
usage: one_arg.py [-h] name
one_arg.py: error: unrecognized arguments: Bronte
```

Only when we give the program exactly one argument will it run:

```
$ ./one_arg.py "Emily Bronte"
Hello, Emily Bronte!
```

While it may seem like overkill to use `argparse` for such a simple program, it shows that `argparse` can do quite a bit of error checking and validation of arguments for us.

A.4.2 Two different positional arguments

Imagine you want two *different* positional arguments, like the *color* and *size* of an item to order. The color should be a `str`, and the size should be an `int` value. When you define them positionally, the order in which you declare them is the order in which the user must supply the arguments. Here we define `color` first, and then `size`:

```
#!/usr/bin/env python3
"""Two positional arguments"""

import argparse


# --------------------------------------------------
def get_args():
    """get args"""

    parser = argparse.ArgumentParser(
        description='Two positional arguments',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('color',
                        metavar='color',
                        type=str,
                        help='The color of the garment')

    parser.add_argument('size',
                        metavar='size',
                        type=int,
                        help='The size of the garment')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """main"""

    args = get_args()
    print('color =', args.color)
    print('size =', args.size)


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

This will be the first of the positional arguments because it is defined first. Notice that metavar has been set to 'color' instead of 'str' as it's more descriptive of the *kind* of string we expect—one that describes the "color" of the garment.

This will be the second of the positional arguments. Here metavar='size', which could be a number like 4 or a string like 'small', so it's still ambiguous.

The "color" argument is accessed via the name of the color parameter.

The "size" argument is accessed via the name of the size parameter.

Again, the user must provide exactly two positional arguments. Entering no arguments triggers a short usage statement:

```
$ ./two_args.py
usage: two_args.py [-h] color size
two_args.py: error: the following arguments are required: color, size
```

Just entering one argument won't cut it either. We are told that "size" is missing:

```
$ ./two_args.py blue
usage: two_args.py [-h] color size
two_args.py: error: the following arguments are required: size
```

If we give it two strings, like "blue" for the color and "small" for the size, the size value will be rejected because it needs to be an integer value:

```
$ ./two_args.py blue small
usage: two_args.py [-h] color size
two_args.py: error: argument size: invalid int value: 'small'
```

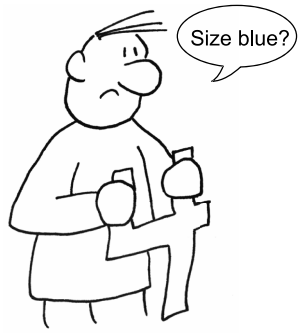
If we give it two arguments, the second of which can be interpreted as an `int`, all is well:


```
$ ./two_args.py blue 4
color = blue
size = 4
```

Remember that *all* the arguments coming from the command line are strings. The command line doesn't require quotes around `blue` or the `4` to make them strings the way that Python does. On the command line, everything is a string, and all arguments are passed to Python as strings.

When we tell `argparse` that the second argument needs to be an `int`, `argparse` will attempt to convert the string `'4'` to the integer `4`. If you provide `4.1`, that will be rejected too:

```
$ ./two_args.py blue 4.1
usage: two_args.py [-h] str int
two_args.py: error: argument int: invalid int value: '4.1'
```



Positional arguments require the user to remember the correct order of the arguments. If we mistakenly switch around `str` and `int` arguments, `argparse` will detect invalid values:

```
$ ./two_args.py 4 blue
usage: two_args.py [-h] COLOR SIZE
two_args.py: error: argument SIZE: invalid int
value: 'blue'
```

Imagine, however, a case of two strings or two numbers that represent two *different* values, like a car's make and model or a person's height and weight. How could you detect that the arguments are reversed?

Generally speaking, I only ever create programs that take exactly one positional argument or one or more *of the same thing*, like a list of files to process.

A.4.3 Restricting values using the choices option

In our previous example, there was nothing stopping the user from providing *two integer values*:

```
$ ./two_args.py 1 2
color = 1
size = 2
```

The `1` is a string. It may look like a number to you, but it is actually the *character* `'1'`. That is a valid string value, so our program accepts it.

Our program would also accept a "size" of `-4`, which clearly is not a valid size:

```
$ ./two_args.py blue -4
color = blue
size = -4
```

How can we ensure that the user provides both a valid `color` and `size`? Let's say we only offer shirts in primary colors. We can pass in a `list` of valid values using the `choices` option.

In the following example, we restrict the `color` to "red," "yellow," or "blue." Additionally, we can use `range(1, 11)` to generate a list of numbers from 1 to 10 (11 isn't included!) as the valid sizes for our shirts:

```
#!/usr/bin/env python3
"""Choices"""

import argparse


# --------------------------------------------------
def get_args():
    """get args"""

    parser = argparse.ArgumentParser(
        description='Choices',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('color',
                        metavar='str',
                        help='Color',
                        choices=['red', 'yellow', 'blue'])

    parser.add_argument('size',
                        metavar='size',
```

The choices option takes a list of values. argparse stops the program if the user fails to supply one of these.

```
                        type=int,
                        choices=range(1, 11),
                        help='The size of the garment')

    return parser.parse_args()
```

The user must choose from the numbers 1–10 or argparse will stop with an error.

```
# --------------------------------------------------
def main():
    """main"""

    args = get_args()
    print('color =', args.color)
    print('size =', args.size)

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

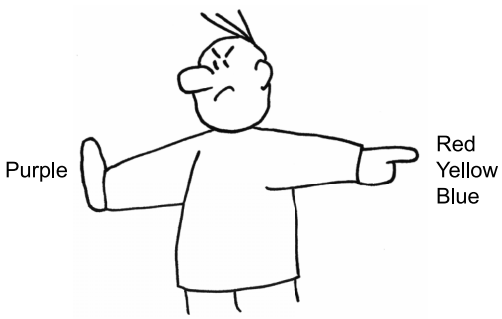
If our program makes it to this point, we know that args.color will definitely be one of those values and that args.size is an integer value in the range of 1–10. The program will never get to this point unless both arguments are valid.

Any value not present in the list will be rejected, and the user will be shown the valid choices. Again, no value is rejected:

```
$ ./choices.py
usage: choices.py [-h] color size
choices.py: error: the following arguments are required: color, size
```

If we provide "purple," it will be rejected because it is not in the `choices` we defined. The error message that `argparse` produces tells the user the problem ("invalid choice") and even lists the acceptable colors:

```
$ ./choices.py purple 1
usage: choices.py [-h] color size
choices.py: error: argument color: \
invalid choice: 'purple' (choose from 'red', 'yellow', 'blue')
```



Likewise with a negative `size` argument:

```
$ ./choices.py red -1
usage: choices.py [-h] color size
```

```
choices.py: error: argument size: \
invalid choice: -1 (choose from 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Only when both arguments are valid may we continue:

```
$ ./choices.py red 4
color = red
size = 4
```

That's really quite a bit of error checking and feedback that you never have to write. The best code is code you don't write!

A.4.4 Two of the same positional arguments

If we were writing a program that adds two numbers, we could define them as two positional arguments, like `number1` and `number2`. But since they are the same kinds of arguments (two numbers that we will add), it might make more sense to use the `nargs` option to tell `argparse` that you want exactly two of a thing:

```
#!/usr/bin/env python3
"""nargs=2"""

import argparse


# --------------------------------------------------
def get_args():
    """get args"""

    parser = argparse.ArgumentParser(
        description='nargs=2',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('numbers',
                        metavar='int',
                        nargs=2,
                        type=int,
                        help='Numbers')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """main"""

    args = get_args()
    n1, n2 = args.numbers
    print(f'{n1} + {n2} = {n1 + n2}')


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

The nargs=2 will require exactly two values.

Each value must be parsable as an integer value, or the program will error out.

Since we defined that there are exactly two values for numbers, we can copy them into two variables.

Because these are actual int values, the result of + will be numeric addition and not string concatenation.

The help indicates we want two numbers:

```
$ ./nargs2.py
usage: nargs2.py [-h] int int
nargs2.py: error: the following arguments are required: int
```

When we provide two good integer values, we get their sum:

```
$ ./nargs2.py 3 5
3 + 5 = 8
```

Notice that `argparse` converts the `n1` and `n2` values to actual integer values. If you change the `type=int` to `type=str`, you'll see that the program will print `35` instead of `8` because the `+` operator in Python both adds numbers and concatenates strings!

```
>>> 3 + 5
8
>>> '3' + '5'
'35'
```

A.4.5 One or more of the same positional arguments

You could expand your two-number adding program into one that sums as many numbers as you provide. When you want *one or more* of some argument, you can use `nargs='+'`:

```
#!/usr/bin/env python3
"""nargs=+"""

import argparse


# --------------------------------------------------
def get_args():
    """get args"""

    parser = argparse.ArgumentParser(
        description='nargs=+',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('numbers',
                        metavar='int',
                        nargs='+',
                        type=int,
                        help='Numbers')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """main"""
```

The + will make nargs accept one or more values.

The int means that all the values must be integer values.

```
    args = get_args()
    numbers = args.numbers

    print('{} = {}'.format(' + '.join(map(str, numbers)), sum(numbers)))

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

numbers will be a list with at least one element.

Don't worry if you don't understand this line. You will by the end of the book.

Note that this will mean `args.numbers` is always a `list`. Even if the user provides just one argument, `args.numbers` will be a `list` containing that one value:

```
$ ./nargs+.py 5
5 = 5
$ ./nargs+.py 1 2 3 4
1 + 2 + 3 + 4 = 10
```

You can also use `nargs='*'` to indicate *zero or more* of an argument, and `nargs='?'` means *zero or one* of the argument.

A.4.6 File arguments

So far you've seen how you can specify that an argument should be of a `type` like `str` (which is the default), `int`, or `float`. There are also many exercises that require a file as input, and for that you can use the `type` of `argparse.FileType('r')` to indicate that the argument must be a *file* that is *readable* (the `'r'` part).

If, additionally, you want to require that the file be *text* (as opposed to a *binary* file), you would add a `'t'`. These options will make more sense after you've read chapter 5.

Here is an implementation in Python of the command `cat -n`, where `cat` will *concatenate* a readable text file, and the `-n` says to *number* the lines of output:

```
#!/usr/bin/env python3
"""Python version of `cat -n`"""

import argparse


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

    parser = argparse.ArgumentParser(
        description='Python version of `cat -n`',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    return parser.parse_args()
```

The argument will be rejected if it does not name a valid, readable text file.

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

    args = get_args()

    for i, line in enumerate(args.file, start=1):
        print(f'{i:6}  {line}', end='')


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

The value of args.file is an open file handle that we can directly read. Again, don't worry if you don't understand this code. We'll talk all about file handles in the chapters.

When we define an argument as `type=int`, we get back an actual `int` value. Here, we define the `file` argument as a `FileType`, so we receive an *open file handle*. If we had defined the `file` argument as a string, we would have to manually check if it were a file and then use `open()` to get a file handle:

```
#!/usr/bin/env python3
"""Python version of `cat -n`, manually checking file argument"""

import argparse
import os


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

    parser = argparse.ArgumentParser(
        description='Python version of `cat -n`',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', metavar='str', type=str, help='Input file')

    args = parser.parse_args()

    if not os.path.isfile(args.file):
        parser.error(f'"{args.file}" is not a file')

    args.file = open(args.file)

    return args


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

    args = get_args()

    for i, line in enumerate(args.file, start=1):
        print(f'{i:6}  {line}', end='')
```

Intercept the arguments.

Check if the file argument is *not* a file.

Print an error message and exit the program with a non-zero value.

Replace the file with an open file handle.

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

With the `FileType` definition, you don't have to write any of this code.

You can also use `argparse.FileType('w')` to indicate that you want the name of a file that can be opened for *writing* (the `'w'`). You can pass additional arguments specifying how to open the file, like the encoding. See the documentation for more information.

A.4.7 Manually checking arguments

It's also possible to manually validate arguments before we `return` from `get_args()`. For instance, we can define that `--int` should be an `int`, but how can we require that it must be between 1 and 10?

One fairly simple way to do this is to manually check the value. If there is a problem, you can use the `parser.error()` function to halt execution of the program, print an error message along with the short usage statement, and then exit with an error value:

```
#!/usr/bin/env python3
"""Manually check an argument"""

import argparse


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

    parser = argparse.ArgumentParser(
        description='Manually check an argument',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-v',
                        '--val',
                        help='Integer value between 1 and 10',
                        metavar='int',
                        type=int,
                        default=5)

    args = parser.parse_args()
    if not 1 <= args.val <= 10:
        parser.error(f'--val "{args.val}" must be between 1 and 10')

    return args


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

Parse the arguments.

Check if the args.int value is *not* between 1 and 10.

Call parser.error() with an error message. The error message and the brief usage statement will be shown to the user, and the program will immediately exit with a non-zero value to indicate an error.

If we get here, everything was OK, and the program will continue as normal.

```
    args = get_args()
    print(f'val = "{args.val}"')


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

If we provide a good `--val`, all is well:

```
$ ./manual.py -v 7
val = "7"
```

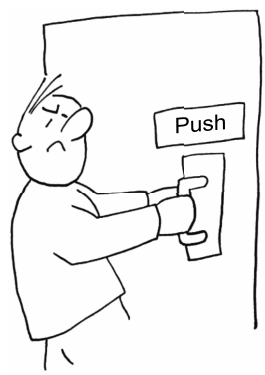
If we run this program with a value like `20`, we get an error message:

```
$ ./manual.py -v 20
usage: manual.py [-h] [-v int]
manual.py: error: --val "20" must be between 1 and 10
```

It's not possible to tell here, but the `parser.error()` also caused the program to exit with a non-zero status. In the command-line world, an exit status of `0` indicates "zero errors," so anything not `0` is considered an error. You may not realize yet just how wonderful that is, but trust me. It is.

A.4.8 Automatic help

When you define a program's parameters using `argparse`, the `-h` and `--help` flags will be reserved for generating help documentation. You do not need to add these, nor are you allowed to use these flags for other purposes.



I think of this documentation as being like a door into your program. Doors are how we get into buildings and cars and such. Have you ever come across a door that you can't figure out how to open? Or one that requires a "PUSH" sign when clearly the handle is designed to "pull"? The book *The Design of Everyday Things* by Don Norman (Basic Books, 2013) uses the term *affordances* to describe the interfaces that objects present to us that do or do not inherently describe how we should use them.

The usage statement of your program is like the handle of the door. It should let users know exactly how to use it. When I encounter a program I've never used, I either run it with no arguments or with `-h` or `--help`. I *expect* to see some sort of usage statement. The only alternative would be to open the source code itself and study how to make the program run and how I can alter it, and this is a truly unacceptable way to write and distribute software!

When you start creating a new program with `new.py foo.py`, this is the help that will be generated:

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

Rock the Casbah

positional arguments:
  str                   A positional argument

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

Without writing a single line of code, you have

- An executable Python program
- A variety of command-line arguments
- A standard and useful help message

This is the "handle" to your program, and you don't have to write a single line of code to get it!

Summary

- Positional parameters typically are required parameters. If you have two or more positional parameters representing different ideas, it would be better to make them named options.
- Optional parameters can be named, like `--file fox.txt` where `fox.txt` is the value for the `--file` option. It is recommended that you always define a default value for options.
- `argparse` can enforce many argument types, including numbers like `int` and `float`, or even files.
- Flags like `--help` do not have an associated value. They are (usually) considered `True` if present and `False` if not.
- The `-h` and `--help` flags are reserved for use by `argparse`. If you use `argparse`, your program will automatically respond to these flags with a usage statement.