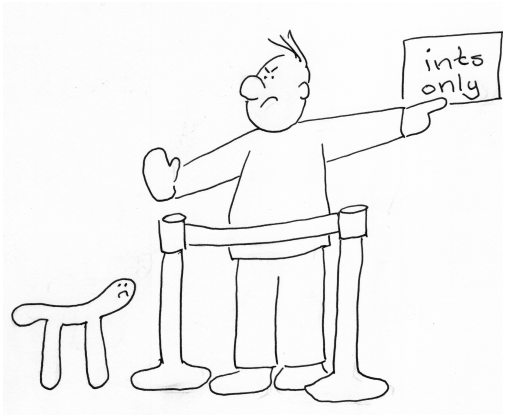
Chapter 2: Using `argparse`

Often getting the right data into your program is a real chore. The `argparse` module can really make your life much easier by validating and rejecting bad arguments from the user. It's like our program's "bouncer," only allowing the right kinds of values into our program. Often half or more of the program can be handled simply by defining the arguments properly with `argparse`!



In Chapter 1, we ended up writing a very flexible program that could extend warm salutations to an optionally named entity such as the "World" or "Universe":

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

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

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

Say hello

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

The `argparse` module helped us define a parser for the parameters and generate the usage, saving us loads of time and making our program 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 would recommend you look over the documentation (https://docs.python.org/3/library/argparse.html). Now let's dig further into what this module can do for us. In this chapter, we 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.

Types of arguments

As we saw in Chapter 1, 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 file name as the first argument and an output directory as the second. Typically positional arguments are always required. Making them optional is difficult — how would you write a program that accepts 2 or 3 arguments where the second and third ones are independent and optional? In all the versions of `hello.py` up until the last one, the argument (a name to greet) was positional.
- **Named options**: Standard Unix format allows for a "short" name like `-f` (one dash and a single character) or a "long" name like `--file` (two dashes and a string of characters) followed by some value like a file name or a number. Named options allow for arguments to be provided in any order, so their *position* is not relevant; hence they are 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. We changed the required, position `name` argument of `hello.py` to the optional `--name`. Note that some 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, `-d` or `--debug` to turn on debugging. Typically the presence of the flag indicates a `True` value for the argument; therefore, it's absence would mean `False`, so `--debug` turns *on* debugging while its absence means it is off. If you run `ls -s`, the `-s` is the flag to show that you want the files sorted by size.

Using `argparse`

Let's dig a bit deeper into the `get_args` function which is defined like this:

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

The `def` keyword defines a new function. 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.

Creating the `parser`

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

```
parser = argparse.ArgumentParser(                           1
    description='Argparse Python script',                   2
    formatter_class=argparse.ArgumentDefaultsHelpFormatter) 3
```

1. Call the `argparse.ArgumentParser` method to create a new `parser`.
2. A short summary of your program's purpose.
3. The `formatter_class` argument tells `argparse` to show the default values in usage. See the documentation for other values you can use.

A positional parameter

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

```
parser.add_argument('positional',                 1
                    metavar='str',                2
                    help='A positional argument') 3
```

1. The lack of leading dashes makes this a positional parameter, not the name "positional."
2. A hint to the user for the data type. By default, all arguments are strings.
3. 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. The `argparse` interprets the string `'positional'` as such because it is not preceeded with any dashes.

An optional string parameter

The following line creates an *optional* parameter with a short name of `-a` and a long name of `--arg` that will be a `str` with a default value of `''` (the empty string). Note that you can leave off either the short or long name in your own programs, but it's good form to provide both. Most of the tests for the exercises will use both short and long option names.

```
parser.add_argument('-a',                         1
                    '--arg',                      2
                    help='A named string argument', 3
                    metavar='str',                4
                    type=str,                     5
                    default='')                   6
```

1. The short name.
2. The long name.
3. Brief description for the usage.
4. Type hint for usage.
5. The actual Python data type (note the lack of quotes around `str`).
6. The default value.

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

An optional numeric parameter

The following line creates the 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:

```
parser.add_argument('-i',                         1
                    '--int',                      2
                    help='A named integer argument', 3
                    metavar='int',                4
                    type=int,                     5
                    default=0)                    6
```

1. The short name.
2. The long name.

3 Brief description for usage.

4 Type hint for usage.

5 Python data type that the string must be converted to. You can also use `float` for a floating point value (a number with a fractional component like `3.14`).

6 The default value.

One of the big reasons to define numeric arguments in this way is that `argparse` will convert the input to the correct `type`. That is, all values coming from the command are strings. It's the job of the program to convert the value to an actual numeric value. If you tell `argparse` that the option should be `type=int`, then when you ask the `parser` for the value, it will have already been converted to an actual `int` value. If the value provided by the user cannot be converted to an `int`, then the value will be rejected. That saves you a lot of time and effort!

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

```
parser.add_argument('-f',                          1
                    '--file',                      2
                    help='A readable file',        3
                    metavar='FILE',                4
                    type=argparse.FileType('r'),   5
                    default=None)                  6
```

1 The short name.

2 The long name.

3 Brief usage.

4 Type suggestion.

5 Says that the argument must name a readable (`'r'`) file.

6 Default value.

A flag

The flag option is slightly different in that it does not take a value like an string or integer. It's just the name part itself. Flags are either present or not. A common flag is `--debug` to turn *on* debugging statements or `--verbose` to print extra messages to the user. When `--debug` is not present, the default value is "off" (or `False`), which is why the `action` for this argument is `store_true`. It's not necessary to set a `default` value as it is automatically set to `False`.

```
parser.add_argument('-o',                  1
                    '--on',                2
                    help='A boolean flag', 3
                    action='store_true')   4
```

1 Short name.

2 Long name.

3 Brief usage.

4 What to do when this is present. The default value is `False`, so `True` will be stored if present.

Returning from `get_args`

The final statement in `get_args` is to `return` the result of having the `parser` object parse the arguments. That is, the code that calls `get_args` will receive this value back:

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

This could fail because `argparse` finds that the user provided invalid arguments, for example, a string value when it expected a `float` or perhaps a misspelled filename. If the parsing succeeds, then we will have a way in our code to access all the values the user provided. Additionally, those values will be of the *types* that we indicated. That is, if we indicate 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.

Manually checking arguments

It's also possible to manually validate arguments before you `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, and then exit with an error:

```
#!/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()  1
    if not 1 <= args.val <= 10:  2
        parser.error(f'--val "{args.val}" must be between 1 and 10')  3

    return args  4


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

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


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

1 Parse the arguments.

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

3 Call `parser.error` with an error message. The entire program will stop, the error message and the brief usage will be shown to the user.

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

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 Unix world, an exit status of `0` indicates "zero errors," so anything not `0` is considered an error. You may not realize just yet how wonderful that is, so just trust me. It is.

Examples using `argparse`

Many of the program tests 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 in. You should always expect and defend against every argument being wrong. [1] Our `hello.py` program was 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`.

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',  1
                        metavar='str',
                        type=str,
                        help='Color')

    parser.add_argument('size',   2
                        metavar='int',
                        type=int,
                        help='Size')

    return parser.parse_args()


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

    args = get_args()
    print('color =', args.color)  3
    print('size =', args.size)


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

1 This will be the first of the positional arguments because it is defined first.

2 This will be the second of the position arguments.

3 The argument is accessed via the name of the parameter `color`.

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


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

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

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

If we give 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 shell (here `bash`) doesn't require quotes around the `blue` or the `4`. To the shell, these are both strings, and they are passed to Python as strings. When we tell `argparse` that the second argument needs to be an `int`, then `argparse` will do the work to attempt the conversion of 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 have the problem that the user is required to remember the correct order. In the case of switching a `str` and `int`, `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 which 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.

Two of the same positional arguments

If you were writing a program that adds two numbers, you could define them as two positional arguments, like `number1` and `number2`. 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 some thing:

```
 1  #!/usr/bin/env python3
 2  """nargs=2"""
 3
 4  import argparse
 5
 6
 7  # --------------------------------------------------
 8  def get_args():
 9      """get args"""
10
11      parser = argparse.ArgumentParser(
12          description='nargs=2',
13          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
14
15      parser.add_argument('numbers',
16                          metavar='int',
17                          nargs=2,   1
18                          type=int,  2
19                          help='Numbers')
20
21      return parser.parse_args()
22
23
24  # --------------------------------------------------
25  def main():
26      """main"""
27
28      args = get_args()
29      n1, n2 = args.numbers              3
30      print(f'{n1} + {n2} = {n1 + n2}')  4
31
32
33  # --------------------------------------------------
34  if __name__ == '__main__':
35      main()
```

1. The `nargs=2` will require exactly 2 values.
2. Each value must be parsable as an integer value or the program will error out.
3. Since we defined that there are exactly two values for `numbers`, we can copy them into two variables.
4. 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
```

On line 29, you see we can unpack the two `numbers` into `n1` and `n2` and use them in the next line for addition:

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

It's completely safe to unpack `numbers` in this way because we would never get to line 29 if the user hadn't provided exactly two arguments, both of which can be converted to `int` values. Also, notice that the `n1` and `n2` values were actually integers. If they had been strings, then our program would print `35` instead of `8` for the arguments `3` and `5`

because the `+` operator in Python both adds numbers and concatenates strings!

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

One or more of the same positional arguments

You could expand your 2-number adder 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='+',  1
                        type=int,   2
                        help='Numbers')

    return parser.parse_args()


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

    args = get_args()
    numbers = args.numbers  3

    print('{} = {}'.format(' + '.join(map(str, numbers)), sum(numbers)))  4


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

1. The `+` will make `nargs` accept one or more values.
2. The `int` means that all the values must be integer values.
3. `numbers` will be a `list` with at least one element.
4. 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
```

Restricting values using `choices`

Sometimes you want to limit the values of an argument. Maybe you offer shirts in only primary colors. You can pass in a `list` of valid values using the `choices` option. Here we restrict the `color` to one of "red," "yellow," or "blue."

```
#!/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'])  1

    return parser.parse_args()


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

    args = get_args()
    print('color =', args.color)  2


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

1 The `choices` option takes a `list` of values. `argparse` will error out if the user fails to supply one of these.

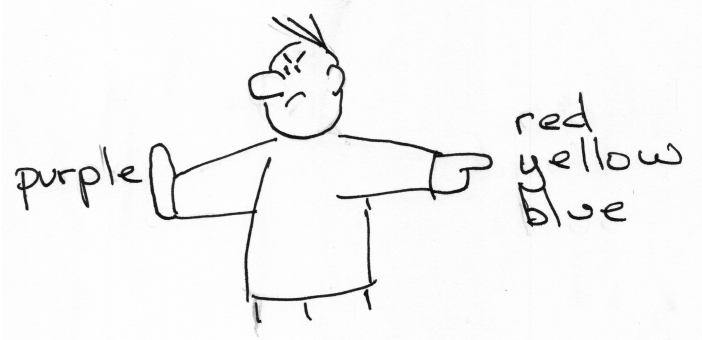
2 If we make it to this point, we know that `args.color` will definitely be one of those values. If the value was rejected, the program will never get to this point.

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] str
choices.py: error: the following arguments are required: str
```

If we provide "purple," it will be rejected because it is not in `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
usage: choices.py [-h] str
choices.py: error: argument str: invalid choice: 'purple'
(choose from 'red', 'yellow', 'blue')
```



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!

File arguments

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

Here is an example showing an implementation in Python of the command `cat -n` where `cat` will *concatenate* files and the `-n` says to *number* the lines of output (which is the command I use to create the following numbered line view of the program — how meta):

```
#!/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('r'),  1
                        help='Input file')

    return parser.parse_args()


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

    args = get_args()

    for i, line in enumerate(args.file, start=1):  2
        print(f'{i:6}  {line}', end='')


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

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

2 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 exercises!

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


```
file = args.file                        1
if not os.path.isfile(file):            2
    print(f'"{file}" is not a file')    3
    sys.exit(1)                         4

fh = open(file)                         5
```

1 Get whatever the user passed in for the `file`.

2 Check if this is *not* a file.

3 Print an error message.

4 Exit the program with a non-zero value.

5 Proceed to `open` the `file`.

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

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 like a door to 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 design to "pull"? The book *The Design of Everyday Things* by Don Norman uses the term "affordances" to describe the interfaces that objects present to us which 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 me 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 a new program with `new.py foo`, this is the help that will be generated:

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

Argparse Python script

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

1. an executable Python program
2. that accepts command line arguments
3. and generates a standard and useful help message

This is the "handle" to your program.

Summary

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

---

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

Last updated 2020-01-16 08:35:30 -0700