Playful Python

Ken Youens-Clark

Version 1.0, 2019-12

Table of Contents

Introduction

////We'll update this front matter when you're finished with the book (and I'll ask you to add more), but I'm probably going to recommend pulling the GitHub stuff into an appendix and expanding it a bit for your MQR. They won't know a lot of the terms you're using)

> "Codes are a puzzle. A game, just like any other game." - Alan Turing

I believe you can learn serious things through playing. This is a book of programming exercises that explores programming ideas by coding puzzles and games. Each chapter includes a description of a program you should write with examples of how the program should work. Most importantly, each program includes tests so that you know if your program is working well enough.

When you are done with this books you be able to:

- Write command-line Python programs
- Process a variety of command-line arguments, options, and flags
- Write and run tests for your programs and functions
- Manipulate of Python data structures including strings, lists, tuples, sets, dictionaries
- Use higher-order functions like `map` and `filter`
- Write and use regular expressions
- Read, parse, and write various text formats
- Use and control randomness
- Create and use graphs, kmers, Markov chains, Hamming distance, the Soundex algorithm, and more

At the core, I want you to learn just how much you can do with Python's basic data types and structures like strings, numbers, lists, tuples, dictionaries, and sets, as well as how to write and test small, focused functions. At no point will I demonstrate or encourage you to write object-oriented code. We will use some modules like `random` (to get random values) and `itertools` (to do some complex list operations), but otherwise I stick to the core Python language and functions.

I also want you to think all the time about how you can test your code. How can you be sure that a function does what you think it does? I want you to learn to write small, testable functions and stick them together to make larger, testable programs.

The ideas we explore in each exercise are not specific to Python. In fact, I think when you ought to write every program in every other language you know and compare what parts of a different language make it easier or harder to write your programs.

Why Write Python?

Python is an excellent, general-purpose programming language. You can use it to write command-line programs, web servers, adventure games, and business applications. There are Python modules to help you wrangle complex scientific data, explore machine learning algorithms, and generate publication-ready graphics.

Many college-level computer science programs have moved away from languages like C and Java to Python as their introductory language because Python is a relatively easy language to learn. Often Python codes reads like English — the statement `for item in cart` is a way to visit each `item` in the collection called `cart`. While languages like C, Java, Perl, and PHP use semi-colons (`;`) to mark the ends of statements, Python uses a newline (when you hit the `Enter` key). These same languages also tend to group statements into "blocks" by enclosing them in curly brackets (`{}`), but Python groups statements by how many spaces they are indented (usually four). Many people feel that Python's syntax and lack of excessive punctuation makes it more readable than other languages.

Who Am I?

My name is Ken Youens-Clark. I work at the University of Arizona as a Sr. Scientific Programmer in the lab of Dr. Bonnie Hurwitz. We are a bioinformatics lab that works mostly in metagenomics. That means we study biology almost entirely from the perspective of sequence data, not microscopes and petri dishes, and we spend most of our time trying to figure out which miscrosopic organisms (bacteria, archaea, fungi, viruses) are present in different environments (like your gut or a wound or in the soil or ocean) and what they are doing.

I began college as a music major (Jazz Studies playing the drumset) at the University of North Texas in 1990. I changed my major a few times and eventually ended up with a BA in English literature in 1995. I didn't really have a plan for my career, but I did like computers. I stared tinkering with databases and HTML at at first job out of college, building the company's mailing list and first website. I was definitely hooked! After that, I managed to turn a temporary technical writing gig into a full-time coding position where I learned VisualBasic on Windows 3.1. Through the next few years, I worked in several languages and companies before landing in a bioinformatics group at Cold Spring Harbor Laboratory in 2001 led by Lincoln Stein, an early advocate for open-source software and data in biology. In 2015 I moved to Tucson, AZ, to join Dr. Hurwitz, and finished my MS in Biosystems Engineering in 2019.

I am, perhaps, a reluctant Python programmer. The largest portion of my coding career — something like 18 years from 1998 to 2016 or so — I wrote almost exclusive in Perl. This is because Perl ruled the early days of the Internet and the world of genomics and bioinformatics. Since then, I've been studying Python, JavaScript, Haskell, Rust, Elm, Prolog, Ruby, Lisp, Julia, and anything else that seems interesting, but Python has clearly taken the lead in scientific computing partly due to a relatively easy syntax that closely mimics English.

When I'm not coding, I like playing music, riding bikes, cooking, reading, and being with my wife and children.

Who Are You?

I think my ideal reader is someone who's been trying to learn to code well but isn't quite sure how to level up. Perhaps you are someone who's been playing with Python or some other language that has a similar syntax like Java(Script) or Perl. Or maybe you've cut your teeth on something like Haskell or Scheme and you're wondering how does it go in Python Land? You are looking for interesting challenges with enough structure to help you know when you're moving in the right direction. Maybe you're interested in "test-driven development" and want to know how to write and use tests? Maybe you've heard of computer science ideas like Markov chains or n-grams or graphs and wonder how they could be useful?

This is a book that will try to teach you formal, structured methods to write Python well. This book introduces techniques I use in my own daily activity, from how I start writing a new program to how I verify that it actually works properly. This book shows you why you'd want to learn about regular expressions and algorithm design and statistics and random events. This book uses simple puzzles and games you already understand and asks you to teach the rules to Python.

This is not an ideal book for the absolute beginning programmer. While I cover some basics of the Python language, I quickly move into programs that assume a familiarity with coding in general. I especially assume you can use the command-line, which might be a leap if you have come from the world of graphical user interfaces like Windows and Apple. I would say this is probably your third or fourth book at least on Python or programming generally.

Why Did I Write This Book?

> "The only way to learn a new programming language is by writing programs in it." - Dennis Ritchie

Over the years, I've had many opportunties to help people learn programming, and I always find it rewarding. The structure of this book comes from my own experience in the classroom where I think formal specifications and tests can be useful aids in learning how to break a program into smaller problems that need to be solved to create the whole program.

The biggest barrier to entry I've found when I'm learning a new language is that small concepts of the language are usually presented outside of any useful context. Yes, we all love to write "HELLO, WORLD!", but after that, I usually struggle to write a complete program that will accept some arguments and do something *useful*. For instance, how can I get the name of a file from the user, then read the file and do something with the contents, all the while handling the various errors that arise like not getting an argument from the user, the argument isn't actually a file, the file isn't readable, etc.? This book teaches you exactly how to write Python that accepts and validates arguments, presents usage messages, handles errors, and runs to completion.

More than anything, I think you need to practice. It's like the old joke: "What's the way to Carnegie Hall? Practice, practice, practice." These coding challenges are short enough that you could probably finish each in a few hours to days. This is more material than I could work through in a semester-long university-level class, so I imagine the whole book would take you several months, perhaps even a year or more. I hope you will solve the problems, then think about them and come back later to see if you can solve them differently, maybe using a more advanced technique or making them run faster. When you're done writing them in Python, I'd suggest you try to write them all again in some other language like Rust, Haskell, or Racket. Push yourself to compare what you know across languages to explore how they are the same and how they differ!

A Note about the lingo

Often in programming books you will see "foobar" used in examples. The word has no real meaning, but its origin probably comes from the military acronym "FUBAR" (Fouled Up Beyond All Recognition). If I use "foobar" in an example, it’s because I don’t want to talk about any specific thing in the universe, just the idea of a string of characters. If I need a list of items, usually the first item will be "foo," the next will be "bar." After that, convention uses "baz" and "quux," again because they mean nothing at all. Don’t get hung up on "foobar." It’s just a shorthand placeholder for something that could be more interesting later.

We also tend to call errors code "bugs." This comes from the days of computing before the invention of transitors. Early machines used vacuum tubes, and the heat from the machines would attract literal bugs who could cause short circuits. The "operators" (the people running the machines) would have to hunt through the tubes to find moths and such, hence the term to "debug."

Why Not Notebooks?

Notebooks are great for an interactive and visual exploration of data, but the downsides:

- Stored as JSON not line-oriented text, so no good `diff` tools
- Not easily shared
- Too easy to run cells out of order
- Hard to test
- No way to pass in arguments

I believe you can better learn how to create testable, *reproducible* software by writing command-line programs that always run from beginning to end and have a test suite. It's difficult to achieve that with Notebooks, but I do encourage you to explore Notebooks on your own.

Code examples, the REPL

I always love when a language has a good REPL (read-evaluate-print-loop) tool. Python excels in this respect. For simplicity's sake, I show the standard REPL when you execute `python3` on the command-line. Unfortunately, you can't copy and paste my code examples directly into the standard REPL or into IDLE. I suggest instead that you use iPython (`ipython`) where the magical `%paste` mode handles the leading `>>>` prompts. Even better, though, I suggest you manually type all the code yourself as this builds muscle memory and forces you to interact with the syntax of the language.

Code formatting and linting

Every program included has been automatically formatted with `yapf` (Yet Another Python Formatter), a tool from Google that can be customized with a configuration file. I encourage you to adopt and regularly use a formatter (see also `black`) *after every modification to your program*. Sometimes I even set up the formatter to format every time I save my program.

I would also encourage you to look at code "linters" like `pylint` and `flake8` to find potential errors in your code that the Python interpreter itself will not compalain about. The `mypy` tool will also be very helpful as we introduce type hints.

Organization

The exercises are roughly arranged from easier to harder. The first exercises deal mostly with strings and then move into lists and tuples. Eventually we move into more complicated data structures like dictionaries and sets. About 80% of my solutions are under 100 lines of code (LOC), about 10% are between 100 and 150 lines, and another 10% are between 150 and 180 lines, so these are not meant to be terribly long programs. Most took me a few hours to figure out, the harder ones a few days. You should work at your own pace and incorporate frequent walks and naps as well as discussions with colleagues (or a rubber duck) about where you're getting stuck.

The shorter, easier examples may seem trivial to solve, but there is quite a bit that you learn along the way. For instance, have you ever used a test suite? Are you familiar with the idea of writing tests first, verifying that your code fails, then writing the code to fix the test, and then verifying that your code now works? That's "test-driven development," and it's a fundamental aspect of the exercises. Have you ever used a code generation tool like `new.py`? Have you ever used `argparse`? Working methodically through each exercise, you will see how the concepts build upon each other, so I'd encourage you to start at the beginning.

Using test-driven development

"Test-driven development" is described by Kent Beck in his 2002 book by that title as a method to create more reliable programs. The basic idea is that we write tests even before we write code. We run the tests and verify that our code fails. Then we write the code to make each test pass. We always run all the tests so that, as we fix new tests, we ensure we don't break tests that were passing before. When all the tests pass, we have at least some assurances that the code we've written conforms to some manner of specification.

Each program you are asked to write comes with tests that will tell you when the code is working acceptably. Once you have started your new program with `new.py`, you should open it in your favorite editor and change the example arguments in `get_args` to suit the needs of your app, then add your code to accomplish the task described in the README. You can run the test suite using the GNU `make` program with `make test` in the same directory as the `test.py` and `article.py` program. If your system does not have `make` or you just don't want to use it, type `pytest -xv test.py`.

Your goal is to pass all the tests. The tests are written in an order designed to guide you in how break the problem down. For instance, if the program alters some input text (like `howler.py` wants you to uppercase the input), the first test will see if you can handle input from the command line. The next test will then pass in a file and ask you to read that for the text. I would suggest you solve the tests in order. If you use `make test`, the command in the `Makefile` executes `pytest -xv test.py` where the `-x` flag will have `pytest` halt testing after it finds one that fails. There's no point in running every test when one fails, nor should you attempt to solve harder problems before solving the simpler cases.

Every exercise has a `test.py` file that contains tests so that you know if your program is working properly. Some also contain a `unit.py` that contains tests for specific functions that you are encouraged (but not required) to write. I suggest you read over the tests to see how they are written and consider how you might expand them or add your own tests.

The first test in every exercise is whether the expected program exists. The second test checks if the program returns something like a "usage" statement when run with `-h|--help`. If you use `new.py` to create your program, you should pass the first two tests. If you create `article.py` as suggested and then run the tests, you should see something like this:

```
$ make test
pytest -xv test.py
==================== test session starts ====================
platform darwin -- Python 3.7.3, pytest-4.3.1, py-1.8.0, pluggy-0.9.0 --
/Users/kyclark/anaconda3/bin/python3
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/manning/playful_python/article, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.3.0, cov-2.7.1, arraydiff-
0.3
collected 6 items

test.py::test_exists PASSED                                          [ 16%]
test.py::test_usage PASSED                                           [ 33%]
test.py::test_01 FAILED                                              [ 50%]

======================== FAILURES ========================
_____________________________________ test_01 _____________________________________

    def test_01():
        """ bear -> a bear """

        for word in consonant_words:
            out = getoutput('{} {}'.format(prg, word))
>           assert out.strip() == 'a ' + word
E           assert 'str_arg = ""...bute \'name\'' == 'a bear'
E             + a bear
E             - str_arg = ""
E             - int_arg = "0"
E             - Traceback (most recent call last):
E             -   File "./article.py", line 74, in <module>
E             -     main()
E             -   File "./article.py", line 67, in main...
E
E             ...Full output truncated (3 lines hidden), use '-vv' to show

test.py:40: AssertionError
============== 1 failed, 2 passed in 0.21 seconds ==============
make: *** [test] Error 1
```

Leaning how to read test failures and figure out what to fix is a big part of what this book teaches. As expected, the first two tests pass, but the third one (`'test_01'`) fails. Can you see the error?

Honestly, it's a huge mess of output and very difficult to see! Try running the program according to the directions:

```
$ ./article.py bear
str_arg = ""
int_arg = "0"
Traceback (most recent call last):
  File "./article.py", line 74, in <module>
    main()
  File "./article.py", line 67, in main
    print('file_arg = "{}"'.format(file_arg.name))
AttributeError: 'NoneType' object has no attribute 'name'
```

Ah, it's still running with all that boilerplate code. Remove all the unneeded arguments and code and make it so the program just echos back the given argument like so:

```
$ ./article.py bear
bear
```

And then run the tests:

```
======================== FAILURES ========================
_____________________________________ test_01 _____________________________________

    def test_01():
        """ bear -> a bear """

        for word in consonant_words:
            out = getoutput('{} {}'.format(prg, word))
>           assert out.strip() == 'a ' + word
E           AssertionError: assert 'bear' == 'a bear'
E             - bear
E             + a bear
E             ? ++
```

The `assert` statement expects the string `'a bear'` but it got back `'bear'`. Fix that. Then run it again. You will probably fail the next test. Fix that test such that you don't break the previously passing one. Keep fixing each test until they all pass. Then you are done. It doesn't matter if you solved the problem the same way as in the `solution.py`. All that matters is that you solve it *on your own*!

Note that many of the tests use an input file that is a dictionary of English words. To be sure that you can reproduce my results, I include a copy of mine in `inputs/words.zip`.

Suggestions for writing

As a general rule, I try to write many very small functions and lots of tests. Every function should be as short as possible, but no shorter — generally 50 lines is my upper limit on function length. I have no lower bound, though. Even if a function is just one line of code, I like to give it a name and some tests to ensure it does what I think. This is often called "unit testing" in the world of software engineering, and it's a very good practice. As the challenges get harder, I start suggesting specific functions and tests you should write in your programs. If we know the smaller bits work individually, then we ought to have confidence that they will work in concert.

This is the basic idea of "compositionality" where we try to compose large systems from smaller ones, so I also write tests to check that my programs as a whole do what they should. This is called "integration testing" and is found in the `test.py` programs I've provided for you in each of these exercises where your programs are run with various options and checked that they produce the expected output.

So, in short, always start a program the same way. Always process the arguments the same way. Always have the same structure to your program (e.g., start with `main`). When you think you need a function in your program, write the tests first and then write the code that will pass the tests. Then integrate your function into the greater whole. Write a test suite outside your program that checks that it fails properly as well as works properly. Whenever you make a change to your program, run your test suite to see if you accidentally broke something that used to work!

Lastly, you should always use a source code manager. If you fork and clone the GitHub repo of the exercises, then you've made an important first step in learning how to keep track of your code. You don't have to use `git` or GitHub. Mercurial is fine, too. (It's also written in Python!) Find a tool that works for you, and commit to tracking your changes.

Chapter 1: How to write a Python program

In this chapter, I'm going to show how to write programs that are documented and tested using the following principles:

- Process and document command-line arguments using `argparse`
- Write and run tests for your code with `pytest`
- Use tools like `yapf` or `black` to format your code
- Annotate variables and functions with type hints and then use `mypy` to check correctness
- Use tools like `flake8` and `pylint` to find problems in your code

How to write the world's best "Hello, World!"

It's pretty common to write "Hello, World!" as your first program in any language, so let's start there. We're going go start as simply as possible and end with a program you'd be proud to submit for a code review.

Start off by creating a text file called `hello.py` and add this line:

```
print('Hello, World!')
```

We can run it with the command `python3 hello.py` to have Python version 3 execute the commands in the file called `hello.py`. (Note that my `python` is version 2, so I will be sure to run `python3`.) You should see this:

```
$ python3 hello.py
Hello, World!
```

Comment lines

In Python, the `#` character and anything following it is ignored by Python. This is useful to add comments to your code or to temporarily disable lines of code when testing and debugging. It's always a good idea to document your programs with the purpose of the program and/or the author's name and email address. We can use a comment for that:

```
# Purpose: Say hello
print('Hello, World!')
```

If you run it again, you should see the same output as before because the "Purpose" line is ignored. Note that any text to the left of the `#` is executed, so you can add a comment to the end of a line, if you like.

The shebang

We could write similar programs in other languages like bash, Ruby, and Perl, so it's common to document the language inside the program with a special comment character combination of `\#!`. The nickname for this is "shebang" (pronounced "shuh-bang" — I always think of the `\#` as "shuh" and the `!` as the "bang!") As usual, Python will ignore this, but the OS will use it to decide which program to use to interpret the rest of the file.

Here is the shebang you should add:

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

The `env` program will tell you about your "environment." If you run it like `env`, you should see lots of lines of output with lines like `USER=kyclark`. If you run it like `env python3`, it will search for and run the `python3` program. On my systems, I like to use the Anaconda Python distribution, and this is usually installed in my "home" directory. On my Mac laptop, this means that my `python3` is actually `/Users/kyclark/anaconda3/bin/python3`, but on another system like one of my Linux web servers will be an entirely different location. If I hardcoded the Mac location of `python3` in the shebang line, then my program won't work when I run it on Linux, so I always use the `env` program to find `python3`.

Now your program should look like this:

```
#!/usr/bin/env python3 ①
# Purpose: Say hello   ②
print('Hello, World!') ③
```

① The shebang line telling the operating system to use the `/usr/bin/env` program to find `python3` to interpret this program.

② A comment line documenting the purpose of the program.

③ A Python command to print some text to the screen.

Making a program executable

Now I'd like to "run" the script. On a Unix-like system, we can make any file "executable" with the command `chmod` (*change mode*). Think of it like turning your program "on." Run this command to make `hello.py` executable — to *add* (`+`) the *executable* (`x`) bit:

```
$ chmod +x hello.py
```

Now you can run the program like so:

```
$ ./hello.py
Hello, World!
```

That looks way cooler.

String diagrams

Throughout the book, I'll use "string diagrams" to visualize the inputs and outputs of the programs we'll write. If we create one for our program as it is, there are no inputs and the output is always "Hello, World!"

Adding a parameter

It's not terribly interesting for our program to always say "Hello, World!" Let's have it enthusiastically greet some name that we will pass as an *argument*. That is, we'd like our program to work like this:

```
$ ./hello.py Terra
Hello, Terra!
```

The arguments to a program are available through the Python `sys` (system) module's `argv` (argument vector). We have to add `import sys` to our program to use it. Change your program so it looks like this:

```
 1  #!/usr/bin/env python3
 2  # Purpose: Say hello
 3
 4  import sys                    ①
 5
 6  args = sys.argv[1:]           ②
 7  if args:                      ③
 8      name = args[0]            ④
 9      print(f'Hello, {name}!') ⑤
10  else:                         ⑥
11      print('Hello, World!')    ⑦
```

① Allows us to use code from the `sys` module.

② `argv` is a list, slice from index 1.

③ If there are any arguments,

④ Then get the `name` from index 0.

⑤ Print `name` using an f-string,

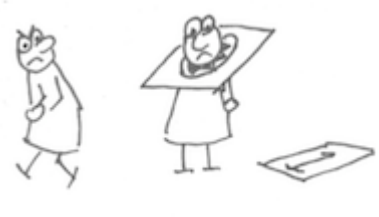
⑥ Otherwise,

⑦ Print the default greeting.

Line 6 probably looks a bit funny, so let's talk about that. The argument vector `argv` is a record of how the program was run starting with the path to the program itself and followed by any arguments. If you run it like `./hello.py Terra`, then `argv` will look like this:

Python, like so many other programming languages, uses the *index* `0` for the first element in a list. The element at `argv[0]` will always be the path to the program itself.

So on line 6, we use `sys.argv[1:]` to say we want a *slice* of that list starting from the index `1` and going to the end of the list and assign that to the variable called `args`. On line 7, we check if there is something in `args` — that is, there were some arguments to the program. If so, on line 8 we assign the variable `name` to the first element in `args`. If there are no `args`, we will print the default greeting on line 11. In chapter 3 (Picnic), we'll talk much more about lists and slices.



Now our program will say "Hello, World!" when there are no

arguments:

```
$ ./hello.py
Hello, World!
```

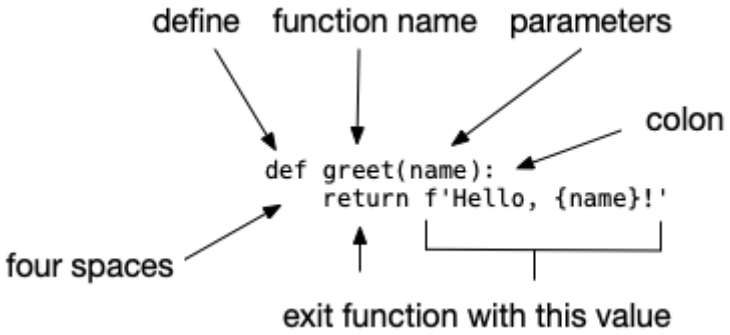
Or will extend warm salutations to whatever value is given as an argument:

```
$ ./hello.py Universe
Hello, Universe!
```

Writing and testing functions

Our `hello.py` program now behaves differently depending on how it is run. How can we *test* our program be sure it does the right thing? Let's find the part of our program that is variable and put that into a smaller unit of code that we can test. The unit will be a *function*, and the thing that changes is the greeting that we produce.

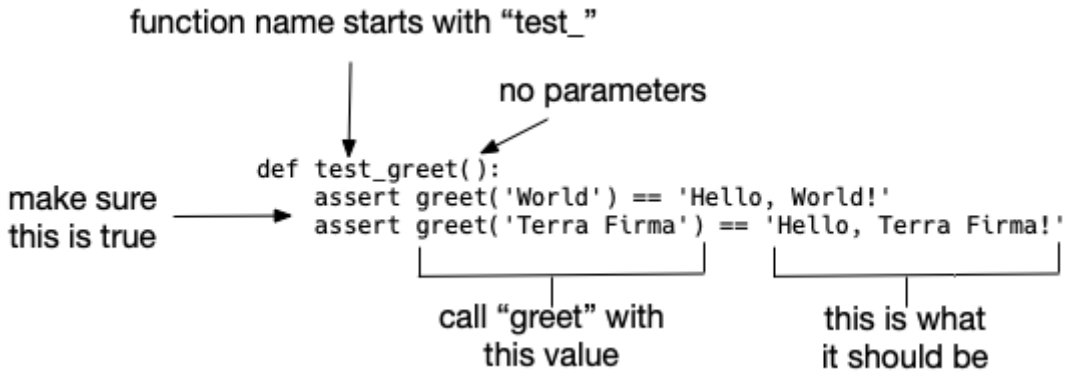
Create a function called `greet` with the `name` as a parameter.



The `def` is to *define* a function, and the function name follows after that. The function's parameters (the things that can change when it is run) go inside the parentheses. Here we have just one, and it's called `name`. The colon (`:`) lets Python know that a "block" of statements will follow, all of which need to be indented a consistent number of spaces (please indent with 4 spaces only, never tabs).

The `return` will leave the function, optionally returning any value that follows. If no value follows the `return`, then the special value `None` will be returned to the caller. Some languages will return the last value of the function, but not Python. You must explicitly `return` what you want. You may return multiple values by separating them with commas, and you are also allowed to have more than one `return` in a function.

Our new `greet` function does not `print` the greeting to the user, though. Its only job is to create the greeting so that we can write a test for it like so:



Here is what our whole program looks like now:

```
 1  #!/usr/bin/env python3
 2  # Purpose: Say hello
 3
 4  import sys
 5
 6  def greet(name):                    ①
 7      return f'Hello, {name}!'
 8
 9  def test_greet():                   ②
10      assert greet('World') == 'Hello, World!'
11      assert greet('Terra Firma') == 'Hello, Terra Firma!'
12
13  args = sys.argv[1:]                 ③
14  name = args[0] if args else 'World' ④
15  print(greet(name))                  ⑤
```

① A function to create a greeting.

② A function to test the `greet` function.

③ The program starts here.

④ An `if` expression is a shorthand way to write `if`/`else` that fits on one line. The `name` will be set to `args[0]` if is something in `args`, otherwise use the value `'World'`. We'll talk more about these in chapter 2 (Crow's Nest) when we are dealing with *binary* (two) choices.

⑤ The `print` and `greet` functions are *two separate ideas*. I know that I can count on Python to reliably `print`, but I need to test my `greet` function to ensure it works properly.

I personally like to use the `pytest` module to run my tests, but you may prefer to use the `unittest` module, especially if you have a background in Java. `pytest` will execute every function that has a name starting with `test_`, so I call my function `test_greet`. This function will call the `greet` function two times, once with the argument `'World'` and once with the argument `'Terra Firma'` to `assert` that the value returned from the function is what I would expect.

To see the tests in action, run `pytest -xv hello.py`:

```
$ pytest -xv hello.py
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-4.6.5, py-1.8.0, pluggy-0.12.0 --
/Users/kyclark/anaconda3/bin/python
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/manning/playful_python/hello
plugins: openfiles-0.3.2, arraydiff-0.3, doctestplus-0.3.0, remotedata-0.3.1, cov-
2.7.1
collected 1 item

hello.py::test_greet PASSED                                              [100%]

============================== 1 passed in 0.03 seconds ========================
```

The `-v` flag is to have `pytest` run in the "verbose" mode, and the `-x` flag to `pytest` tells it to stop at the first failing test. It's common to combine short flags in this way (`-xv` or `-vx`, the order doesn't matter). Here the `-x` flag causes `pytest` to stop when it sees that `crowsnest.py` isn't there. There's no need to run any further tests if the program doesn't even exist. I include a `Makefile` with each exercise so that you can run `make test` to execute this for you. If you don't have `make` or don't want to use it, just manually run `pytest -xv test.py`.

Adding a `main` function

Our program is pretty good, but it's not quite idiomatic Python. Do you notice how we have two functions and then some other code just sort of hanging out (lines 13-15) flush left? It's good practice to put *all* your code inside functions, and it's very common to create a function called `main` where your program starts:

```
def main():
    args = sys.argv[1:]
    name = args[0] if args else 'World'
    print(greet(name))
```

We still have to tell Python to run this function, though, and the idiom for that in Python is to put these as the *last* two lines in your program:

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

Python reads your whole program from top to bottom. When it gets to these last lines, it will see if the special variable called `{dbl_}name{dbl_}` (one of the many special "double-under" or "dunder" variables) shows that we are in the "main" namespace. If you use `import` to bring your code into another piece of code, then it would *not* be in the "main" namespace. When you execute the code like a program, then then the namespace will be "main" and so the `main` function will be run.

Here is our whole program now:

```
 1  #!/usr/bin/env python3
 2  # Purpose: Say hello
 3
 4  import sys
 5
 6  def greet(name):
 7      return f'Hello, {name}!'
 8
 9  def test_greet():
10      assert greet('World') == 'Hello, World!'
11      assert greet('Terra Firma') == 'Hello, Terra Firma!'
12
13  def main():                    ①
14      args = sys.argv[1:]
15      name = args[0] if args else 'World'
16      print(greet(name))
17
18  if __name__ == '__main__':     ②
19      main()                     ③
```

① A new `main` entry function.

② Program starts here.

③ Call "main" function if in "main" namespace

Note that the call to run `main()` must occur in the file *after* the definition of the `main` function. Python will throw an exception if you try to call a function that hasn't yet been defined. Try moving these to the *top* of your program and running it again, and you'll see an error:

```
NameError: name 'main' is not defined
```

Why is testing important?

Would you fly on a plane or ride on an elevator if you knew it had never been tested? While lives may not depend so directly on the software you write, you still want to write code that is free from errors.

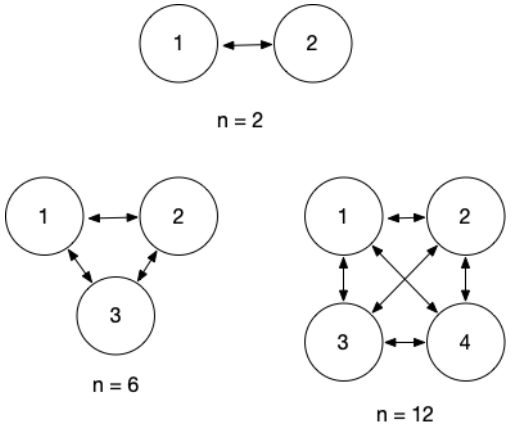


Figure 1. Complexity increases quickly!

As programs get longer, the tendency is that they get much harder to maintain. It's like adding a new team member. When there are just two people, there are only two ways to talk. Every new member increases the number of channels exponentially. Each new line of code or function you write interacts with all the other code in ways that become far too complicated for you to remember.

Tests help us check that our code still does what we think it does. Imagine we want to translate our `hello.py` to Spanish. We need to change the "Hello" to "Hola," but we misspell it "Halo":

```
#!/usr/bin/env python3
# Purpose: Say hello

import sys

def greet(name):
    return f'Halo, {name}!' ①

def test_greet():
    assert greet('World') == 'Hola, World!'             ②
    assert greet('Terra Firma') == 'Hola, Terra Firma!' ③

def main():
    args = sys.argv[1:]
    name = args[0] if args else 'World'
    print(greet(name))

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

① "Hola" is misspelled.

② "Hola" is correct.

③ Here, too.

If you run `pytest` on this, you will see this failure:

```
================================== FAILURES ===================================
_________________________________ test_greet __________________________________

    def test_greet():
>       assert greet('World') == 'Hola, World!'
E       AssertionError: assert 'Halo, World!' == 'Hola, World!'
E         - Halo, World!
E         ?   --
E         + Hola, World!
E         ?  ++

hello07.py:10: AssertionError
============================ 1 failed in 0.07 seconds =============================
```

I would encourage you to write many functions, each of which does a few things as possible. Each function should have a `test_`, either in the same source file (I usually place it immediately after the function it's testing) or in a separate file (which I usually call `unit.py` for "unit" tests).

Often I will even write my `test_` function *before* I write the function itself, and I will imagine how I would want the function to work under a variety of conditions. As a general rule, I usually will test a function with:

- No arguments
- One argument
- Several arguments
- Bad arguments

I will then test if the function works or fails as I expect it. For instance, what would you have `greet` do if given no arguments or given a *list* of arguments instead of one?

Adding type hints

Each variable in Python has a `type`. For instance, we execute `python3` on the command line to interact directly with the Python interpreter. You will see a `>>>` prompt that is waiting for you to type some Python code: Create a `name` variable set to "World":

```
>>> name = 'World'
```

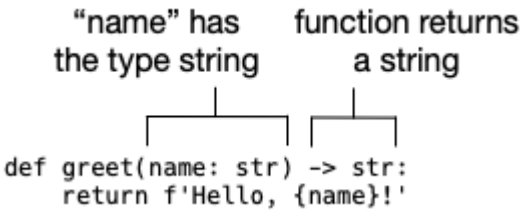
Now use the `type` command to see how Python represents this data:

```
>>> type(name)
<class 'str'>
```

Anything in quotes (single or double) is a "string," which in Python is represented by the class `str`. Python has many other types, for instance, the number `10` is an `int` or "integer":

```
>>> type(10)
<class 'int'>
```

We can add "type hints" to our code to ensure that we use the right types of data, like not trying to divide an `int` by a `str` which would cause an *exception* and crash our code.



Shown is how `greet` would look with type hints. You can read `name: str` as "name has the type string." Additionally, the `greet` function itself has been annotated with `\-> str` to indicate that it will return a `str` value. If a function (like `main`) returns nothing, add the special `None` as the return type.

Here is what the entire program looks like with type hints. ***Can you find the error?***

```
 1  #!/usr/bin/env python3
 2  # Purpose: Say hello
 3
 4  import sys
 5
 6  def greet(name: str) -> str:  ①
 7      return f'Hello, {name}!'
 8
 9  def test_greet() -> None:     ②
10      assert greet('World') == 'Hello, World!'
11      assert greet('Terra Firma') == 'Hello, Terra Firma!'
12
13  def main() -> None:           ③
14      args = sys.argv[1:]
15      print(greet(args))
16
17  if __name__ == '__main__':
18      main()
```

① The `name` argument is a `str`, and the function returns a `str`.

② The function returns `None`.

③ The function returns `None`.

Verification with `mypy`

Python itself completely ignores the type hints, but we can use the `mypy` program to find a problem in the code:

```
$ mypy hello.py
hello.py:15: error: Argument 1 to "greet" has incompatible type "List[str]"; expected
"str"
```

Can you see that we tried to pass `args` (which is a list of strings) to `greet` instead of a single `str` value? Here is the correct `main` function:

```
def main() -> None:
    args = sys.argv[1:]
    name = args[0] if args else 'World'
    print(greet(name))
```

Manually validating and documenting arguments

We're now going to change our program to require exactly one argument. If we don't get that, we need to print a "usage" statement that explains how to use the program:

```
$ ./hello.py
usage: hello.py NAME
```

And likewise if run with more than one argument:

```
$ ./hello.py Terra Firma
usage: hello.py NAME
```

When given one argument, it should work as expected:

```
$ ./hello.py "Terra Firma"
Hello, Terra Firma!
```

Here is the new version to make that work:

```
 1  #!/usr/bin/env python3
 2  # Purpose: Say hello
 3
 4  import os, sys
 5
 6  def greet(name: str) -> str:
 7      return f'Hello, {name}!'
 8
 9  def test_greet() -> None:
10      assert greet('World') == 'Hello, World!'
11      assert greet('Terra Firma') == 'Hello, Terra Firma!'
12
13  def main() -> None:
14      args = sys.argv[1:]
15      if len(args) != 1:                              ①
16          prg_name = os.path.basename(sys.argv[0]) ②
17          print(f'usage: {prg_name} NAME')          ③
18          sys.exit(1)                                 ④
19      else:
20          print(greet(args[0]))
21
22  if __name__ == '__main__':
23      main()
```

① `args` must have 1 element.

② `sys.argv[0]` has program name. Use `basename` to remove the path.

③ Print "usage" statement.

④ Exit with error (not a 0 value).

On line 4, note that I can `import` more than one module by separating them with commas. Here I `import os` so that I can use the `os.path.basename` function to get the "basename" of `sys.argv[0]` which, you'll recall, is the path of the program itself. The "basename" of a path is the filename itself:

```
>>> import os
>>> os.path.basename('/path/to/hello.py')
'hello.py'
```

On line 15, I check if the `len` (length) of the `args` is *not* 1. If so, I'll print a "usage" statement on how to run the program. Note that I call `sys.exit(1)` to indicate a *non-zero exit value* because `0` (which is the default) indicates "zero errors." Any exit value that is not `0` indicates a failure.

Most Unix tools will also respond to `-h` or `--help` to show a "usage" statement. What happens when we try that?

```
$ ./hello.py --help
Hello, --help!
```

Well, that didn't work out. Just because we gave `--help` as an argument didn't mean that our program interpreted it correctly. Our program has been written to process *positional* arguments only, and the `--help` argument is a "flag," meaning a value that is `True` when present and `False` when absent. Here is a pretty ugly way to code this:

```
def main() -> None:
    args = sys.argv[1:]
    val = args[0] if args else ''
    if len(args) != 1 or val == '-h' or val == '--help':
        prg_name = os.path.basename(sys.argv[0])
        print(f'usage: {prg_name} NAME')
    else:
        print(greet(val))
```

Now we are setting a `val` ("value") variable equal to the first argument if one is present. Then we can check `len(args)` or if the `val` is equal to either the string `'-h'` or `'--help'`. I can verify that it works:

```
$ ./hello.py -h
usage: hello.py NAME
$ ./hello.py --help
usage: hello.py NAME
```

That big compound `if` test is pretty ugly. It's what I would call a "code smell" — it's trying to do too many things at once. We can write something much more elegant.

Documenting and validating arguments using `argparse`

The Python language has a standard module called `argparse` that will parse all the command line arguments, options, and flags. You have to invest a bit of time to learn it, but it will save you so much time in return. Here is a new version that will make our code much cleaner:

```
 1  #!/usr/bin/env python3
 2  # Purpose: Say hello
 3
 4  import argparse, os, sys
 5
 6  def greet(name: str) -> str:
 7      return f'Hello, {name}!'
 8
 9  def test_greet() -> None:
10      assert greet('World') == 'Hello, World!'
11      assert greet('Terra Firma') == 'Hello, Terra Firma!'
12
13  def main() -> None:
14      parser = argparse.ArgumentParser(description='Say hello') ①
15      parser.add_argument('name', help='The name to greet')     ②
16      args = parser.parse_args()                                ③
17      print(greet(args.name))                                   ④
18
19  if __name__ == '__main__':
20      main()
```

① Create argument parser.

② Add the `name` parameter.

③ Get the parsed `args`.

④ Print the greeting given the `args.name` value.

Now instead of directly dealing with `sys.argv`, we describe to `argparse` that we want a single argument called `name` and let it do the hard work of parsing and validating the arguments. This will be a *positional* argument because `name` does *not* start with a dash. We will set `args` equal to the result of our `parser` doing the work to `parse_args`.

Now if you run the program with no arguments, the "usage" statement will be generated by `argparse`. Though you can't see it directly, the exit value is also set to something other than `0` to indicate failure:

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

And both the `-h` and `--help` flags will trigger a longer help document that looks like a Unix `man` page:

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

Say hello

positional arguments:
  str         The name to greet

optional arguments:
  -h, --help  show this help message and exit
```

As a matter of personal taste, I like to put all the `argparse` code into its own function that I always call `get_args`. For some of my programs, this can get quite long, and it makes the `main` function stay much shorter if this is separated. Besides, it is a functional unit in my mind, and so it belongs by itself. I always put `get_args` as the first function so that I can see it immediately when I read the source code. I usually put `main` right after it.

Here is how the program looks now:

```
#!/usr/bin/env python3
# Purpose: Say hello

import argparse

def get_args() -> argparse.Namespace: ①
    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('name', metavar='str', help='The name to greet')
    return parser.parse_args()

def main() -> None:
    args = get_args()                ②
    print(greet(args.name))

def greet(name: str) -> str:
    return f'Hello, {name}!'

def test_greet() -> None:
    assert greet('World') == 'Hello, World!'
    assert greet('Terra Firma') == 'Hello, Terra Firma!'

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

① The `get_args` function dedicated to getting the command-line arguments

② Call `get_args` function to get parsed arguments.

Checking style and errors

We now have the bones of a pretty respectable program. I like to use the `flake8` and `pylint` tools to give me suggestions on how to improve my programs. This is a process called "linting," and the tools are called "linters." I find that `flake8` is unhappy with readability as it

tells me to put 2 lines after each function definition:

```
$ flake8 hello.py
hello.py:6:1: E302 expected 2 blank lines, found 1
hello.py:11:1: E302 expected 2 blank lines, found 1
hello.py:15:1: E302 expected 2 blank lines, found 1
hello.py:18:1: E302 expected 2 blank lines, found 1
hello.py:22:1: E305 expected 2 blank lines after class or function definition, found 1
```

The `pylint` program has other things to complain about, namely that my functions are missing documentation ("docstrings"):

```
$ pylint hello.py
************* Module hello
hello.py:1:0: C0111: Missing module docstring (missing-docstring)
hello.py:6:0: C0111: Missing function docstring (missing-docstring)
hello.py:11:0: C0111: Missing function docstring (missing-docstring)
hello.py:15:0: C0111: Missing function docstring (missing-docstring)
hello.py:18:0: C0111: Missing function docstring (missing-docstring)

-------------------------------------------------------------------
Your code has been rated at 6.67/10 (previous run: 6.67/10, +0.00)
```

A docstring is a string that occurs just after the `def` of the function. It can be a single line of text enclosed in single or double quotes. It's also common to use several lines, and Python allows you to use triple-quotes for strings that have line breaks: You can assign them to a variable:

```
>>> multi_line = """
... I should have been a pair of ragged claws.
... Scuttling across the floors of silent seas.
... """
```

Or you can use them in place of the `#` for multi-line comments. For instance, I usually document my whole program by putting a docstring just after the shebang. Inside I put my name, email address, the purpose of the script, and the date.

To fix the formatting issues, I ran the whole program through the formatting program called `yapf` (Yet Another Python Formatter). Another popular formatter is `black`. It really doesn't matter which one you choose. Just choose one and use it religiously.

Here is a version that will silence all of our critics:

```
#!/usr/bin/env python3
"""
Purpose: Say hello        ①
Author:  Ken Youens-Clark
"""

import argparse


# --------------------------------------------------
def get_args() -> argparse.Namespace:
    """Get command-line arguments""" ②

    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('name', metavar='str', help='The name to greet')
    return parser.parse_args()


# --------------------------------------------------
def main() -> None:
    """Start here"""

    args = get_args()
    print(greet(args.name))


# --------------------------------------------------
def greet(name: str) -> str:
    """Create a greeting"""

    return f'Hello, {name}!'


# --------------------------------------------------
def test_greet() -> None:
    """Test greet"""

    assert greet('World') == 'Hello, World!'
    assert greet('Terra Firma') == 'Hello, Terra Firma!'


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

① Triple-quoted, multi-line docstring for program/module.

② Triple-quotes can be used on a single line, too.

I also like to add the commented dashes to help me see the beginning of each function. I think it makes my code more readable, but this purely personal taste — you can omit these.

Making the argument optional

Let's return to making the argument to our program optional so that we can run with and without an argument. We'd like to run the program with no argument and have it default to using "World" for the `name` like so:

```
$ ./hello.py
Hello, World!
```

If we make `name` an *optional* argument, it can no longer be a *positional* argument. Therefore, we have to use either `-n` or `--name` as the name for the optional argument:

```
$ ./hello.py --name Cleveland
Hello, Cleveland!
```

The "short" name is a single dash and a single letter like `-n`, and the "long" name is two dashes followed by a longer name like `--name`. Here is what the usage looks like now:

```
$ ./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)
```

And here is the code. Note the use of the `formatter_class` to have `argparse` show the default values for arguments in the "usage" output:

```
 1  #!/usr/bin/env python3
 2  """
 3  Purpose: Say hello
 4  Author:  Ken Youens-Clark
 5  """
 6
 7  import argparse
 8
 9
10  # --------------------------------------------------
11  def get_args() -> argparse.Namespace:
12      """Get command-line arguments"""
13
14      parser = argparse.ArgumentParser(
15          description='Say hello',
16          formatter_class=argparse.ArgumentDefaultsHelpFormatter) ①
```

```
17
18      parser.add_argument('-n',             ②
19                          '--name',         ③
20                          default='World',  ④
21                          metavar='str',    ⑤
22                          help='The name to greet')
23
24      return parser.parse_args()
25
26
27  # --------------------------------------------------
28  def main() -> None:
29      """Start here"""
30
31      args = get_args()
32      print(greet(args.name))
33
34
35  # --------------------------------------------------
36  def greet(name: str) -> str:
37      """Create a greeting"""
38
39      return f'Hello, {name}!'
40
41
42  # --------------------------------------------------
43  def test_greet() -> None:
44      """Test greet"""
45
46      assert greet('World') == 'Hello, World!'
47      assert greet('Terra Firma') == 'Hello, Terra Firma!'
48
49
50  # --------------------------------------------------
51  if __name__ == '__main__':
52      main()
```

① Show default values in "usage."

② Short option name

③ Long option name

④ The default value.

⑤ Hint to user of the data type.

Starting a new program with `new.py`

In my own practice, I almost never start writing a Python program from an empty page. I created a Python program called `new.py` that helps me start writing new Python programs. As most of my programs need to take parameters, I always use the `argparse` to interpret the command-line options.

I have put my `new.py` program into the `bin` ("binaries" even though these are just text files) directory of the GitHub repo, and I recommend you start every new program with this program.

A central tenet of this book is to create *documented* and *testable* programs. Anything that *can* change about a program should be passed as an *argument* to the program when it is run. For instance, if a program will read an input file for data, the name of that file should be an argument like `-f input1.txt`. Then the fact that the program takes an input file is now visible through the `--help` and we can pass in different files and test if the program processes the files correctly. It is *not* a requirement that you use `new.py` and `argparse`, however. As long as your programs process command-line arguments in the same way as `argparse` and always produce a usage on `-h|--help`, your programs should pass the test suites. The template that `new.py` provides is meant only to make it faster and more convenient to create new programs.

Here is how I would use `new.py` to create a new `hello.py` program:

```
$ new.py hello.py
Done, see new script "hello.py."
```

This is what will be produced:

```
 1  #!/usr/bin/env python3
 2  """
 3  Author : Ken Youens-Clark <kyclark@gmail.com>
 4  Date   : 2019-10-21
 5  Purpose: Rock the Casbah
 6  """
 7
 8  import argparse
 9  import os
10  import sys
11
12
13  # --------------------------------------------------
14  def get_args():
15      """Get command-line arguments"""
16
17      parser = argparse.ArgumentParser(
18          description='Rock the Casbah',
19          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
20
21      parser.add_argument('positional',
```

```
22                          metavar='str',
23                          help='A positional argument')
24
25      parser.add_argument('-a',
26                          '--arg',
27                          help='A named string argument',
28                          metavar='str',
29                          type=str,
30                          default='')
31
32      parser.add_argument('-i',
33                          '--int',
34                          help='A named integer argument',
35                          metavar='int',
36                          type=int,
37                          default=0)
38
39      parser.add_argument('-f',
40                          '--file',
41                          help='A readable file',
42                          metavar='FILE',
43                          type=argparse.FileType('r'),
44                          default=None)
45
46      parser.add_argument('-o',
47                          '--on',
48                          help='A boolean flag',
49                          action='store_true')
50
51      return parser.parse_args()
52
53
54  # --------------------------------------------------
55  def main():
56      """Make a jazz noise here"""
57
58      args = get_args()
59      str_arg = args.arg
60      int_arg = args.int
61      file_arg = args.file
62      flag_arg = args.on
63      pos_arg = args.positional
64
65      print('str_arg = "{}"'.format(str_arg))
66      print('int_arg = "{}"'.format(int_arg))
67      print('file_arg = "{}"'.format(file_arg.name))
68      print('flag_arg = "{}"'.format(flag_arg))
69      print('positional = "{}"'.format(pos_arg))
70
71
72  # --------------------------------------------------
```

```
73  if __name__ == '__main__':
74      main()
```

The arguments that this program will accept are:

1. A single positional argument of the type `str`. *Positional* means it is not preceeded by a flag to name it but has meaning because of its position.
2. An automatic `-h` or `--help` flag that will cause `argparse` to print the usage.
3. A named string argument called either `-a` or `--arg`
4. A named interger argument called `-i` or `--int`
5. A named file argument called `-f` or `--file`
6. A boolean (off/on) flag called `-o` or `--on`

Each option here has both a "short" and "long" name. It is not a requirement to have both, but it is common and tends to make your program more readable.

After you use `new.py` to start your new program, you should open it with your editor and modify the argument names and types to suit the needs of your program. For instance, in the "Crow's Nest" chapter, you can delete everything but the positional argument which you should rename from `'positional'` to something like `'word'` (because the argument is going to be a word).

Note that you can control the `name` and `email` values that are used by `new.py` by creating a file called `.new.py` (note the leading dot!) in your home directory. Here is mine:

```
$ cat ~/.new.py
name=Ken Youens-Clark
email=kyclark@gmail.com
```

If you don't want to use `new.py`, then I have included a sample of the above program as `template/template.py` that you can copy. For instance, in the next chapter you should create the program `crowsnest/crowsnest.py`. (That is, from the root directory of the repository, go into the `crowsnest` directory and create a file called `crowsnest.py`.)

Either you can do this with `new.py`:

```
$ cd crowsnest
$ new.py crowsnest.py
```

Or the `cp` (copy) command:

```
$ cp template/template.py crowsnest/crowsnest.py
```

The main point is that I don't want you to have to start every program from scratch! I think it's much easier to start from a complete, working program and modify it.

Types of arguments

As you see above, 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, one-line 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 like the following options.

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.

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:

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

return args ④
```

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.

Examples using `argparse`

A fair number of the 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. 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`:

```
 1  #!/usr/bin/env python3
 2  """Two positional arguments"""
 3
 4  import argparse
 5
 6
 7  # --------------------------------------------------
 8  def get_args():
 9      """get args"""
10
11      parser = argparse.ArgumentParser(
12          description='Two positional arguments',
13          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
14
15      parser.add_argument('color',
16                          metavar='str',
17                          type=str,
18                          help='Color')
19
20      parser.add_argument('size',
21                          metavar='int',
22                          type=int,
23                          help='Size')
24
25      return parser.parse_args()
26
27
28  # --------------------------------------------------
29  def main():
30      """main"""
31
32      args = get_args()
33      print('color =', args.color)
34      print('size =', args.size)
35
36
37  # --------------------------------------------------
38  if __name__ == '__main__':
39      main()
```

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

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

Just one won't cut it. 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 two arguments, the second of which can be interpreted as an `int`, all is well:

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

It's important to note that *all* the arguments coming from the command line are strings because pretty much everything in `bash` (or whatever your shell) is a string. 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] COLOR SIZE
two_args.py: error: argument SIZE: 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:

```
#!/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='Two numbers')

    return parser.parse_args()


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

    args = get_args()
    n1, n2 = args.numbers
    print('n1 =', n1)
    print('n2 =', n2)


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

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:

```
$ ./nargs2.py 3 5
n1 = 3
n2 = 5
```

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='+',
                        type=int,
                        help='Numbers')

    return parser.parse_args()


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

    args = get_args()
    numbers = args.numbers

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


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

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. On line 18 in the following program, I restrict the `color` to one of "red," "yellow," or "blue."

```
 1  #!/usr/bin/env python3
 2  """Choices"""
 3
 4  import argparse
 5
 6
 7  # --------------------------------------------------
 8  def get_args():
 9      """get args"""
10
11      parser = argparse.ArgumentParser(
12          description='Choices',
13          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
14
15      parser.add_argument('color',
16                          metavar='str',
17                          help='Color',
18                          choices=['red', 'yellow', 'blue'])
19
20      return parser.parse_args()
21
22
23  # --------------------------------------------------
24  def main():
25      """main"""
26
27      args = get_args()
28      print('color =', args.color)
29
30
31  # --------------------------------------------------
32  if __name__ == '__main__':
33      main()
```

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')
```

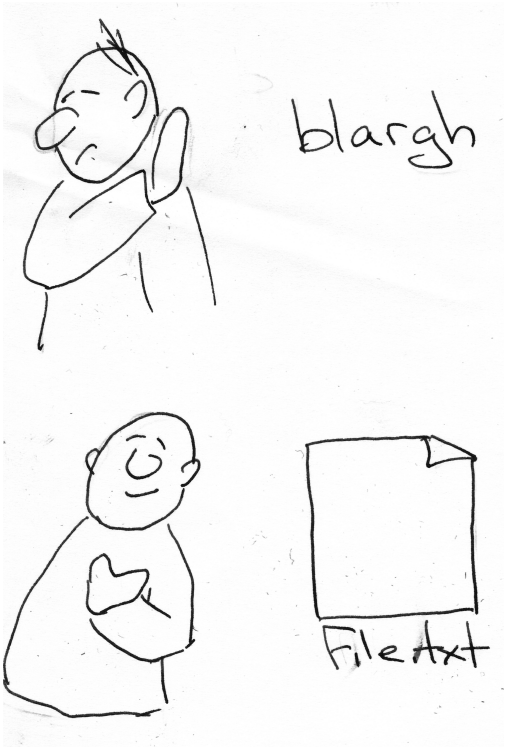
That's really quite a bit of error checking and feedback that you never have to write. The best code is code you don't write!

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):

```python
#!/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'),
                        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):
        print(f'{i:6}  {line}', end='')


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



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. Plus we can directly read line-by-line with a `for` loop on line 29!

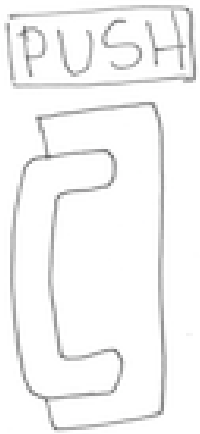
Automatic help

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

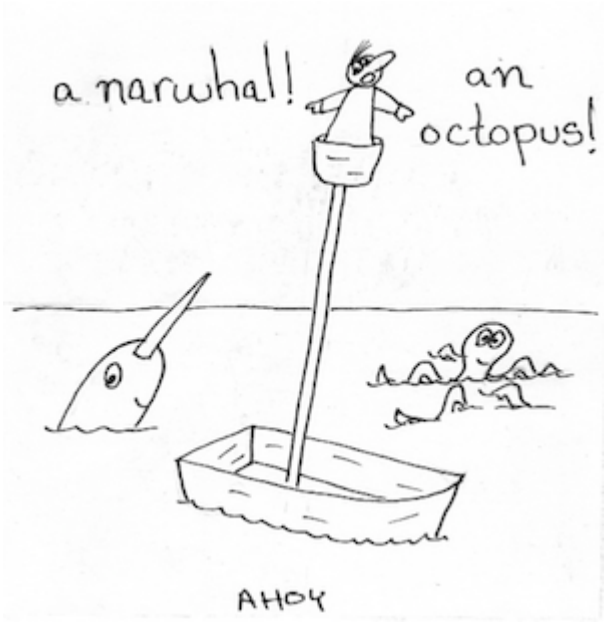
- The `argparse` module will help you document and parse all the parameters to your program. You can validate the types and numbers of arguments which can be positional, optional, or flags. The usage will be automatically generated.
- You should write many small, limited functions and `test_` functions that you can run with `pytest`. It is often best to write the tests before you write the functions to imagine how they ought to work.
- You should run your tests after any change to your program to ensure that everything still works.
- You should automatically format your code with `yapf` and `black` (and probably other programs).
- You should add type hints to your code and use `mypy` to check for errors.
- You should "lint" your code with `pylint` and/or `flake8` to find both programmatic and style problems.
- You can use the `new.py` program to generate new Python programs that use `argparse`.

Chapter 2: The Crow's Nest: Working with strings

From this point on, I want you to try your hand at writing programs that I will describe. In each chapter, I will describe a program that you should write and will discuss key ideas you'll need to solve the problems. In the Git repository, I've provided tests for each program to help you know when everything is correct. You should have a copy of the Git repo locally (see the setup instructions) where you should write your program and run the tests.

In this chapter, we're going to start off working with strings. By the end, you will be able to:

- Create a program that accepts a positional argument and produces usage documentation
- Create a new output string depending on the inputs to the program
- Run a test suite



For this program, you are the lookout in the crow's nest — the little bucket attached to the top of a mast of a sailing ship. Your job is to keep a lookout for interesting or dangerous things, like a ship to plunder or an iceberg to avoid. When you see something like a "narwhal," you are supposed to cry out, "Ahoy, Captain, ***a narwhal*** off the larboard bow!" If you see an octopus, you'll shout "Ahoy, Captain, ***an octopus*** off the larboard bow!" (We'll assume everything is "off the larboard bow" for this exercise. It's a great place for things to be.)

Your program should be called `crowsnest.py`. It will accept a single positional argument and will `print` the given argument inside the "Ahoy" bit along "a" or "an" depending on whether the argument starts with a consonant or a vowel.

That is, if given "narwhal," it should do this:

```
$ ./crowsnest.py narwhal
Ahoy, Captain, a narwhal off the larboard bow!
```

And if given "octopus":

```
$ ./crowsnest.py octopus
Ahoy, Captain, an octopus off the larboard bow!
```

This means we're going to need to write a program that accepts some input on the command line, decides on the proper article ("a" or "an") for the input, and prints out a new string that puts those two values into the "Ahoy" phrase.

Getting started

You're probably ready to start writing the program! Well, hold on just a minute longer. We need to discuss how we'll use the tests to know when our program is working and how we might get started programming.

How to use the tests

> "The greatest teacher, failure is." — Yoda

In the code repository, I've included tests that will guide you in the writing of your program. Before you even write the first line of code, I'd like you to run `make test` or `pytest -xv test.py` so you can see how the first test fails. Be sure you are in the `crowsnest` directory for this!

Among all the output, you'll notice this line:

```
test.py::test_exists FAILED                                          [ 16%]
```

If you read more, you'll see lots of other output all trying to convince you that the expected file, `crowsnest.py` does not exist. Learning to read the test output is a skill in itself! It takes quite a bit of practice to learn to read test output, so try not to feel overwhelmed. In my terminal (iTerm on a Mac), the output from `pytest` shows colors and bold print to highlight key failures. The text in bold, red letters is usually where I start, but your terminal may behave differently.

Creating programs with `new.py`

In order to pass this test, we need to create a file called `crowsnest.py` inside the `crowsnest` directory where `test.py` is located. While it's perfectly fine to start writing from scratch, I suggest you use the `new.py` program to print some useful boilerplate code that you'll need in every exercise:

```
$ new.py crowsnest.py
Done, see new script "crowsnest.py."
```

If you don't want to use `new.py`, you could copy the `template/template.py` program:

```
$ cp template/template.py crowsnest/crowsnest.py
```

At this point you should have the outline of a working `crowsnest.py` program that accepts command-line arguments. If you run your new `crowsnest.py` with no arguments, it will print a short usage statement like the following (notice how "usage" is the first word of the output):

```
$ ./crowsnest.py
usage: crowsnest.py [-h] [-a str] [-i int] [-f FILE] [-o] str
crowsnest.py: error: the following arguments are required: str
```

Those are not the correct parameters for our program, just the default examples given to you by `new.py`. You will need to modify them to suit this program.

If you run your tests again, you will pass the first *two* tests that check:

1. Does the program exist?
2. Does the program print something that looks like "usage."

And then you will fail the third test. There are more tests after this, but that's all your see if you run `pytest -xv test` because the `-x` flag tells `pytest` to stop at the first failing test:

```
test.py::test_exists PASSED                                              [ 16%]
test.py::test_usage PASSED                                               [ 33%]
test.py::test_consonant FAILED                                           [ 50%]
```

Now we have a working program that accepts some arguments (but not the right ones). Next we need to make our program accept the "narwhal" or "octopus" value that needs to be announced, and we'll use command-line arguments to do that.

Defining your arguments

Here is a diagram showing the inputs (or *arguments*) and output of the program. We'll use these throughout the book to imagine how code and data work together. In this program, some "word" is the input, and a phrase incorporating that word with the correct article is the ouptut.

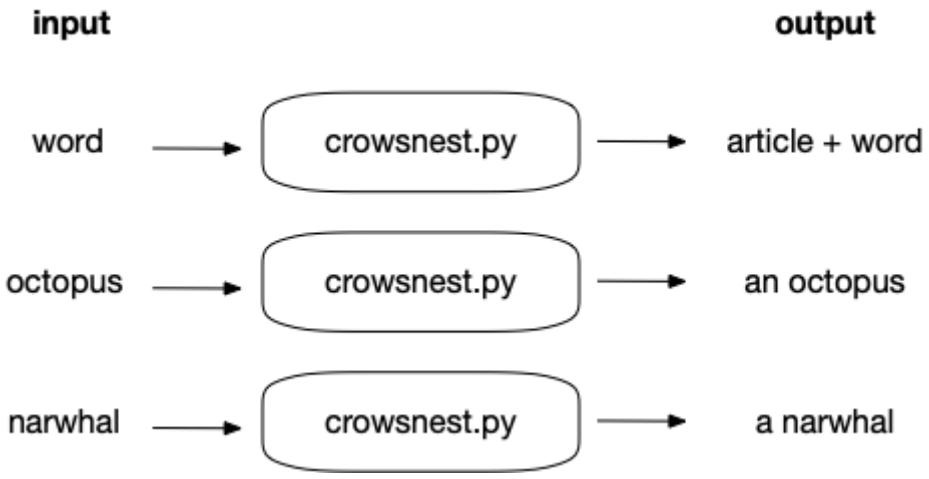


Figure 2. The input to the program is a word, and the output is that word plus it's proper article (and some other stuff).

We need to modify the part of the program that gets the arguments — the aptly named `get_args` function. This function uses the `argparse` module to parse the command-line arguments. Refer to the `argparse` discussion in the introduction, particularly the section "A single, positional argument."

The default `get_args` names the first argument `'positional'`, and that's the only one you need. Remember that positional arguments are defined by their position and so don't have names that start with dashes. You can delete all the arguments except for the positional `word`.

Modify the `get_args` part of your program until it will print this usage:

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

Likewise, it should print a longer usage for the `-h` or `--help` flag:

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

Crow's Nest -- choose the correct article

positional arguments:
  str         A word

optional arguments:
  -h, --help  show this help message and exit
```

When your program prints the correct usage, you can get the `word` argument inside the `main` function like so:

```
def main():
    args = get_args()
    word = args.word
```

Make your program print the given `word`:

```
$ ./crowsnest.py narwhal
narwhal
```

And now run your tests. You should still be passing two and failing the third. Let's read the test failure:

```
================================== FAILURES ===================================
_______________________________ test_consonant ________________________________

    def test_consonant():
        """brigatine -> a brigatine"""

        for word in consonant_words:
            out = getoutput('{} {}'.format(prg, word))
>           assert out.strip() == template.format('a', word)
E           AssertionError: assert 'brigatine' == 'Ahoy, Captain, a brigatine off the
larboard bow!'
E             - brigatine
E             + Ahoy, Captain, a brigatine off the larboard bow!
```

Those lines start with `E` are the "error" lines. The line starting with a `-` is what the test got when it ran with the argument `'brigatine'` — it got back just the word "brigatine." The line starting with the `+` is what the test expected, "Ahoy, Captain, a brigatine off the larboard bow!"

So, we need to get the `word` into the "Ahoy" phrase. How can we do that?

Concatenating strings

Putting strings together is usually called "concatenating," but you could also call it "joining" strings. To demonstrate, I'm going to enter some code directly into the Python interpreter. I want you to type along. No, really! Type everything you see, and try it for yourself.

Open a terminal and type `python3` or `ipython` to start a REPL, a "Read-Evaluate-Print-Loop" because Python will *read* each line of input, *evaluate* and *print* the results in a *loop*. (I pronounce this like "reh-pull" in a way that kind of, sort of rhymes with "pebble.") Here's what it looks like on my system:

```
$ python3
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You may also like to use Python's IDLE (integrated development environment) program, or you can use Jupyter notebooks to interact with the language. I'll stick to the simple `python3` REPL for showing code examples. To exit the REPL, either type `quit()` or `CTRL-d` (the `Control` key plus the `d`).

The `>>>` is a prompt where you can type code. Let's start off by assigning the variable `word` to the value "narwhal." In the REPL, type `word = 'narwhal'<Enter>`:

```
>>> word = 'narwhal'
```

Note that you can put as many (or no) spaces around the `=` as you like, but convention and readability (and tools like `pylint` or `flake8` that help you find errors in your code) would ask you to use exactly one space on either side. If you type `word<Enter>`, Python will print the current value of `word`:

```
>>> word
'narwhal'
```

Now type `werd<Enter>`:

```
>>> werd
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'werd' is not defined
```

There is no `werd` variable because we haven't set `werd` to be anything. Using an undefined variable causes an *exception* that will crash your program. Python will happily create a `werd` for you when you assign it a value.

In Python, there are many ways we can concatenating strings. The `+` operator can be used to join strings together:

```
>>> 'Ahoy, Captain, a ' + word + ' off the larboard bow!'
'Ahoy, Captain, a narwhal off the larboard bow!'
```

If you change your program to `print` that instead of just the `word`, you should be able to four tests:

```
test.py::test_exists PASSED                                          [ 16%]
test.py::test_usage PASSED                                           [ 33%]
test.py::test_consonant PASSED                                       [ 50%]
test.py::test_consonant_upper PASSED                                 [ 66%]
test.py::test_vowel FAILED                                           [ 83%]
```

If we look closely at the failure, you'll see this:

```
E            - Ahoy, Captain, a aviso off the larboard bow!
E            + Ahoy, Captain, an aviso off the larboard bow!
E            ?                +
```

So we hard-coded the "a" before the `word`, but we really need to figure out whether to put "a" or "an" depending on whether the `word` starts with a vowel. How can we do that?

Variable types

Before we go much further, I need to take a small step back and point out that our `word` variable is a "string." Every variable in Python has a "type" that describes the kind of data they hold. Because we put the value for `word` in quotes (`'narwhal'`), the `word` holds a "string" which Python represents with a class called `str`. (A "class" is a collection of code and functions that we can use.)

The `type` function will tell us what kind of data Python thinks this is:

```
>>> type(word)
<class 'str'>
```

Whenever you put a value in single (`''`) or double quotes (`""`), Python will interpret it as a `str`:

```
>>> type("submarine")
<class 'str'>
```

If you forget the quotes, then Python will look for some variable or function by that name. If there is no variable or function by that name, it will cause an exception.

```
>>> word = narwhal
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'narwhal' is not defined
```

Exceptions are bad, and we will try to write code that avoids them or at least knows how to handle them gracefully.

Getting just part of a string

Back to our problem! We need to put either "a" or "an" in front of the `word` we're given based on whether the first character of `word` is a vowel or a consonant. In Python, we use square brackets and an *index* to get an individual character from a string. The index is the numeric position of an element in a sequence, and we must remember that indexing starts at `0`. You can use this with a variable:

```
>>> word[0]
'n'
```

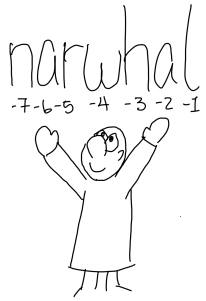
Or directly on a string:

```
>>> 'narwhal'[0]
'n'
```

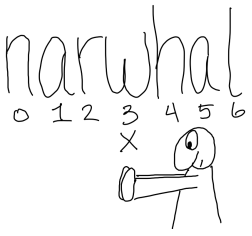
This means that the last index is *one less than the length*, which is often confusing. The length of "narwhal" is 7, but the last character is found at index `6`:

```
>>> word[6]
'l'
```

You can also use negative numbers to count backwards from the end, so the last index is also `-1`:




```
>>> word[-1]
'l'
```



You can use the "slice" notation `[start:stop]` to get a range of characters. Both `start` and `stop` are optional. The default value for start is `0` (the beginning of the string), and the `stop` value is *not inclusive*:

```
>>> word[:3]
'nar'
```

And the default value for `stop` is the end of the string:

```
>>> word[3:]
'whal'
```

In the next chapter, we'll see that this is the same syntax for slicing lists. A string is (sort of) a list of characters, so this isn't too strange.

Finding help in the REPL

The class `str` has a ton of functions we can use to handle strings, but what are they? A large part of programming is knowing how to ask questions and where to look for answers. A common refrain you may hear is "RTFM" — Read the Fine Manual. The Python community has created reams of documentation which are all available at https://docs.python.org/3/. You will need to refer to the documentation constantly to remind yourself how to use certain functions.

The docs for the string class are here:

https://docs.python.org/3/library/string.html

I prefer to read the docs directly inside the REPL by typing `help(str)`:

```
>>> help(str)
```

Inside the `help`, you move up and down in the text using the up and down cursor arrows on your keyboard. You can also press the `<Space>` bar or `CTRL-f` to jump forward to the next page, and `CTRL-b` to jump backward. You can search through the documentation by pressing `/` and then the text you want to find. If you press `n` (for "next") after a search, you will jump to the next place that string is found. To leave the help, press `q` (for "quit").

String methods

Now that we know `word` is a string (`str`), we have all these incredibly useful *methods* we can call on the variable. (A "method" is a function that belongs to a variable like `word`.) For instance, if I wanted to shout about the fact that we have a "narwhal," I could print it in UPPERCASE LETTERS. If I search through the help, I see there is a function called `upper`. Here is how to call it:

```
>>> word.upper()
'NARWHAL'
```

You must include the parentheses `()` or else you're talking about the *function itself*:

```
>>> word.upper
<built-in method upper of str object at 0x10559e500>
```

That will actually come in handy later when we use functions like `map` and `filter`, but for now we want Python to *execute* or *call* the `upper` function on the varible `word`, so we add the parens. Note that the function returns an uppercase version of the word but *does not* change the value of `word` itself:

```
>>> word
'narwhal'
```

There is another `str` function with "upper" in the name called `isupper`. The name helps you know that this will return a True/False type answer. Let's try it:

```
>>> word.isupper()
False
```

We can chain methods together like so:

```
>>> word.upper().isupper()
True
```

That makes sense. If I convert the `word` to uppercase, then `isupper` is `True`.

I find it odd that the `str` class does not include a method to get the length of a string. For that, we

use a separate function called `len`, short for "length":


```
>>> len('narwhal')
7
```

Are you typing all this into Python yourself? I recommend you do! Find other methods in the `str` help and try them out.

String comparisons

So now you know how to get the first letter of `word` by using `word[0]`. Let's assign it to the variable `char`:

```
>>> word = 'octopus'
>>> char = word[0]
>>> char
'o'
```

Now we need to figure out if `char` is a vowel or a consonant. We'll say that letters "a," "e," "i," "o," and "u" make up our set of "vowels." You can use `==` to compare strings:

```
>>> char == 'a'
False
>>> char == 'o'
True
```

Be careful to always use one equal sign (`=`) when *assigning a value* to a variable, like `word = 'narwhal'` and two equal signs (`==`, which, in my head, I say "equal-equal") when you *compare two values* like `word == 'narwhal'`. The first is a statement that changes the value of `word`, and the second is an *expression* that returns `True` or `False`.

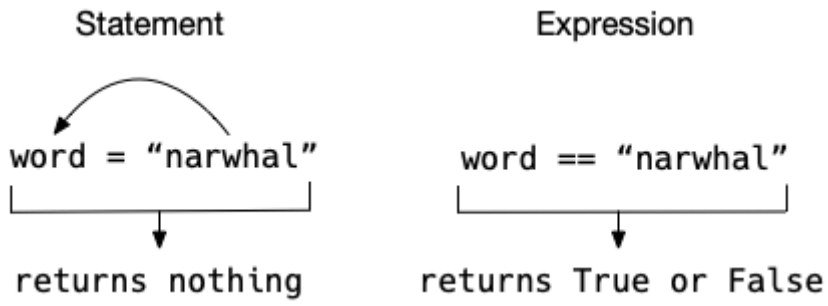


Figure 3. An expression returns a value. A statement does not.

We need to compare our `char` to *all* the vowels. You can use `and` and `or` in such comparisons and they will be combined according to standard Boolean algebra:

```
>>> char == 'a' or char == 'e' or char == 'i' or char == 'o' or char == 'u'
True
```

What if the `word` is "Octopus" or "OCTOPUS"?

```
>>> word = 'OCTOPUS'
>>> char = word[0]
>>> char == 'a' or char == 'e' or char == 'i' or char == 'o' or char == 'u'
False
```

Do we have to make 10 comparisons in order to check the uppercase versions, too? What if we were to lowercase the `word`?

```
>>> word = 'OCTOPUS'
>>> char = word.lower()[0]
>>> char == 'a' or char == 'e' or char == 'i' or char == 'o' or char == 'u'
True
```

An easier way to determine if `char` is a vowel would be to use Python's `x in y` construct where we want to know if the value `x` is in the collection `y`. We can ask if the letter `'a'` is in the longer string `'aeiou'`:

```
>>> 'a' in 'aeiou'
True
```

But the letter `'b'` is not:

```
>>> 'b' in 'aeiou'
False
```

Now use it to test on a lowercased `char` (which is `'o'`):

```
>>> char.lower() in 'aeiou'
True
```

Conditional branching

Once you have figured out if the first letter is a vowel, you will need to select an article. We'll use a very simple rule that, if the word starts with a vowel, choose "an," otherwise choose "a." This misses exceptions like when the initial "h" in a word is silent. For instance, we say both "a hat" and "an honor". Nor will we consider when an initial vowel has a consonant sound as in "union" where the "u" sounds like a "y."

We can create a new variable called `article` that we will set to the empty string and then use an `if`/`else` statement to figure out what to put in it:

```
>>> article = ''        ①
>>> if char in 'aeiou': ②
...     article = 'an'  ③
... else:               ④
...     article = 'a'   ⑤
...
```

① Initialize `article` to the empty string.

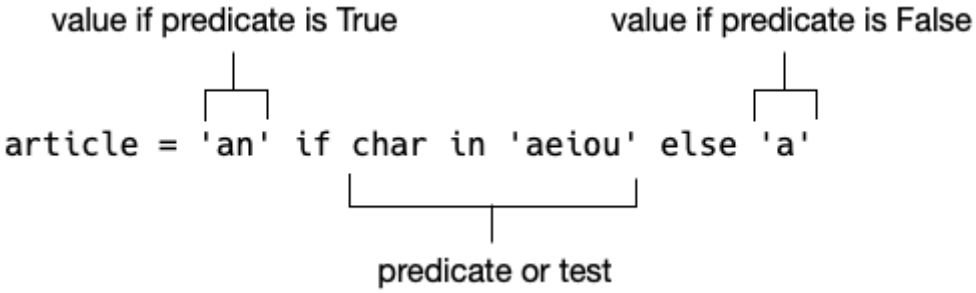
② Check if `char` is a vowel.

③ If `char` is a vowel, set `article` to `'an'`

④ Otherwise...

⑤ Set `article` to `'an'`.

Here is a much shorter way to write that with an `if` *expression* (expressions return values, statements do not). The `if` expression is written a little backwards. First comes the value if the test (or "predicate") is `True`, then the predicate, then the value if the predicate is `False`:



This way is also safer because the `if` expression is *required* to have the `else`. There's no chance that we could forget to handle both cases:

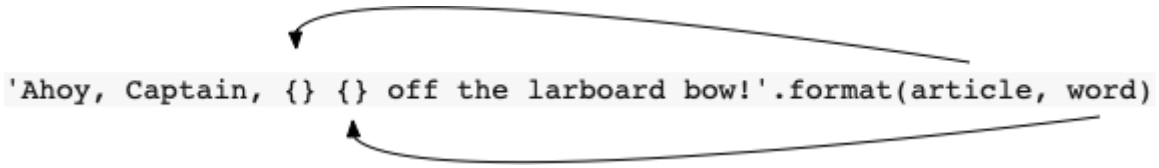
```
>>> article = 'an' if char in 'aeiou' else 'a'
```

And we can verify that we have the correct `article`:

```
>>> article
'an'
```

String formatting

Now we have two variables, `article` and `word` that need to be incorporated into our "Ahoy!" phrase. We saw earlier that we can use the plus sign (`+`) to concatenate strings. Another method to create new strings from other strings is to use the `str.format` method. To do so, you create a string template with curly brackets `{}` that indicate placeholders for values. The values that will be substituted go as arguments to the `format`, and they are substituted in the same order that the `{}` appear:



Here it is in code:

```
>>> 'Ahoy, Captain, {} {} off the larboard bow!'.format(article, word)
'Ahoy, Captain, an octopus off the larboard bow!'
```

Another method uses the special "f-string" where you can put the variables directly into the `{}` brackets. It's a matter of taste which one you choose.

```
>>> f'Ahoy, Captain, {article} {word} off the larboard bow!'
'Ahoy, Captain, an octopus off the larboard bow!'
```

Python variables are very variable

A note that in some programming languages, you have to declare the variable's name and what *type* of data it will hold. If a variable is declared to be a number, then it can never hold a value of a different type like a string. This is called *static typing* because the type of the variable can never change. Python is a *dynamically typed* language because you do not have to declare a variable or what kind of data the variable will hold. You can change the value and type of data at any time. This could be either great or terrible news. As Hamlet says, "There is nothing either good or bad, but thinking makes it so."

It's business time

At this point, you should probably be able to pass all the tests. Can you do it? Go write the program!

Hints:

- Start your program with `new.py` and fill in the `get_args` with a single position argument called `word`.
- You can get the first character of the word by indexing it like a list, `word[0]`.
- Unless you want to check both upper- and lowercase letters, you can use either the `str.lower` or `str.upper` method to force the input to one case for checking if the first character is a vowel or consonant.
- There are fewer vowels (five, if you recall) than consonants, so it's probably easier to check if the first character is one of those.
- You can use the `x in y` syntax to see if the element `x` is `in` the collection `y` where "collection" here is a `list`.
- Use the the `str.format` or f-strings to insert the correct article for the given word into the longer phrase.
- Run `make test` (or `pytest -xv test.py`) *after every change to your program* to ensure your program compiles and is on the right track.

Now go write the program before you turn the page and study a solution!

Solution

```
 1  #!/usr/bin/env python3
 2  """Crow's Nest"""
 3
 4  import argparse
 5
 6
 7  # --------------------------------------------------
 8  def get_args(): ①
 9      """Get command-line arguments"""
10
11      parser = argparse.ArgumentParser( ②
12          description="Crow's Nest -- choose the correct article", ③
13          formatter_class=argparse.ArgumentDefaultsHelpFormatter)  ④
14
15      parser.add_argument('word', metavar='str', help='A word')    ⑤
16
17      return parser.parse_args() ⑥
18
19
20  # --------------------------------------------------
21  def main(): ⑦
22      """Make a jazz noise here"""
23
24      args = get_args() ⑧
25      word = args.word  ⑨
26      article = 'an' if word[0].lower() in 'aeiou' else 'a' ⑩
27
28      print(f'Ahoy, Captain, {article} {word} off the larboard bow!') ⑪
29
30
31  # --------------------------------------------------
32  if __name__ == '__main__': ⑫
33      main() ⑬
```

① Defines the function `get_args` to handle the command-line arguments. I like put this first so I can see it right away when I'm reading the code.

② The `parser` will do the work of parsing the arguments.

③ The `description` shows in the usage to describe what the program does.

④ Show the default values for each parameter in the usage.

⑤ Define a positional argument called `word`.

⑥ The result of parsing the arguments will be returned to line 24.

⑦ Defines the `main` function where the program will start.

⑧ `args` contains the return value from the `get_args` function.

⑨ Put the `args.word` value from the arguments into the variable `word`.

⑩ Choose the correct `article` using an `if` expression to see if the lowercased, first character of `word` is or is not in the set of vowels.

⑪ Print the output string using an f-string to interpolate the `article` and `word` variables inside the string.

⑫ Check if we are in the "main" namespace, which means the program is *running*.

⑬ If so, call the `main()` function to make the program start.

Discussion

I'd like to stress that the preceeding is *a* solution, not *the* solution. There are many ways to express the same idea in Python. As long as your code passes the test suite, it is correct.

That said, I created my program with `new.py` which automatically gives me two functions:

1. `get_args` where I define the arguments to the program
2. `main` where the program starts

Let's talk about these two functions.

(I imagine the above code could be annotated with big numbers that could be referenced by the following sections. E.g., a big "1" next to the `get_args` function and a big "2" next to `main`, etc. Or at least there could be some simple call-outs on the above code to generally explain what's happening, although I think the function names themselves are pretty explanatory.)

Defining the arguments with `get_args`

I prefer to put the `get_args` function first so that I can see right away what the program expects as input. You don't have to define this as a separate function. You could put all this code inside `main`, if you prefer. Eventually our programs are going to get longer, though, and I think it's nice to keep this as a separate idea. Every program I present will have a `get_args` function that will handle defining and validating the input.

Our program specifications (the "specs") say that the program should accept one positional argument. I changed the `'positional'` argument name to `'word'` because I'm expecting a single word:

```
parser.add_argument('word', metavar='str', help='Word')
```

I would really recommend you never leave the "positional" argument named `'positional'` because it is an entirely undescriptive term. Naming your variables *what they are* will make your code more readable. Since the program doesn't need any of the other options created by `new.py`, you can delete the rest of the `parser.add_argument` calls. The `get_args` function will `return` the result of parsing the command line arguments which I put into the variable `args`:

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

If `argparse` is not able to parse the arguments — for example, there are none — it will never `return` from `get_args` but will instead print the "usage" for the user and exit with an error code to let the operating system know that the program exited without success. (In the Unix world, an exit value of `0` means there were 0 errors. Anything other than `0` is considered an error.)

The `main` thing

Many programming languages will automatically start from the `main` function, so I always define a `main` function and start my programs there. This is not a requirement, just how I like to write programs. Every program I present will start with the `main` function which will first call `get_args` to get the program's inputs:

```
args = get_args()
```

I can now access the `word` by call `args.word`. Note the lack of parentheses. It's not `args.word()` because is not a function call. Think of `args.word` like a slot where the value of the "word" lives:

```
word = args.word
```

I like to work through my ideas using the REPL, so I'm going to pretend that `word` has been set to "octopus":

```
>>> word = 'octopus'
```

Classifying the first character of a word

To figure out whether the article I choose should be `a` or `an`, I need to look at the first character of the `word` which we can get like so. In the introduction, we used this:

```
>>> word[0]
'o'
```

I can check if the first character is `in` the string of vowels, both lower- and uppercase:

```
>>> word[0] in 'aeiouAEIOU'
True
```

I can make this shorter, however, if I use `word.lower` function so I'd only have to check the lowercase vowels:

```
>>> word[0].lower() in  'aeiou'
True
```

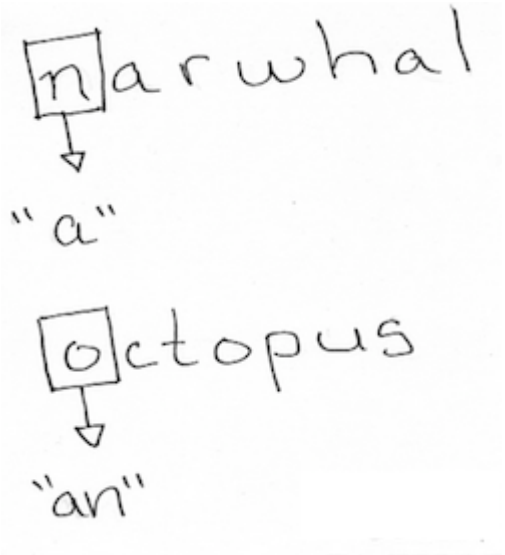
Remember that the `x in y` form is a way to ask if element `x` is in the collection `y`. You can use it for letters in a longer string (like the vowels):

```
>>> 'a' in 'aeiou'
True
```

Or for a string in list of other strings:

```
>>> 'tanker' in ['yatch', 'tanker', 'vessel']
True
```

We can use membership in the "vowels" as a condition to choose "an," otherwise we choose "a":



As mentioned in the introduction, the `if` expression is the shortest and safest for a "binary" choice (where there are only two possibilities):

```
>>> article = 'an' if word[0].lower() in  'aeiou' else 'a'
>>> article
'an'
```

The safety comes from the fact that Python will not even run this program if you forget the `else`. We can change the `word` to "galleon" and check that it still works:

```
>>> word = 'galleon'
>>> article = 'an' if word[0].lower() in  'aeiou' else 'a'
>>> article
'a'
```

Printing the results

Finally we need to `print` out the phrase with our `article` and `word`. As noted in the introduction, you can use `str.format`:

```
>>> article = 'a'
>>> word = 'ketch'
>>> print('Ahoy, Captain, {} {} off the larboard bow!'.format(article, word))
Ahoy, Captain, a ketch off the larboard bow!
```

Python's f-strings will *interpolate* any code inside the `{}` placeholders, so variables get turned into their contents:

```
>>> print(f'Ahoy, Captain, {article} {word} off the larboard bow!')
Ahoy, Captain, a ketch off the larboard bow!
```

However you chose to print out the article and word is fine as long as it passes the tests.

Running the test suite

"A computer is like a mischievous genie. It will give you exactly what you ask for, but not always what you want. - Joe Sondow"

Computers are a bit like bad genies. They will do exactly what you tell them but not necessarily what you *want*. In an episode of *The X-Files*, the character Mulder wishes for peace on Earth and a genie removes all humans but him.

Tests are what we can use to verify that our programs are doing what we actually want them to do. Tests they can never prove that our program is actually, completely free of errors, only that the bugs we imagined or found while writing the program no longer exist. Still, we write and run tests because they are really quite effective and much better than not doing so.

This is the idea behind "test-driven development":

- We can write tests *even before* we write the software.
- We run the tests to verify that our as-yet-unwritten software definitely fails to deliver on some task.
- Then we write the software to fulfill the request.
- Then we run the test to check that it now *does* work.
- We keep running all the tests to ensure that, when we add some new code, we do not break existing code.

Passing Tests

I would encourage you to look at the `test.py` program to see how it is testing your program. Eventually I'll recommend you write your own tests, but for now just see what's being expected of your code. I use the `pytest` module to write tests. There are other testing frameworks in Python, but I find `pytest` to be relatively easy to use. The `pytest` module will run any functions that begin with `test_` in the order they are found in the source code.

The first `test_` function is `test_exists` that uses the `assert` function to check if the `crowsnest.py` program exists. This is why your program must be named `'crowsnest.py'`. It must exist as this name so that we can run it and check the output:

```
prg = './crowsnest.py'

def test_exists():
    """exists"""

    assert os.path.isfile(prg)
```

The next is `test_usage` to check if the program will print something that looks like "usage" when run with `-h` and `--help` flags:

```
def test_usage():
    """usage"""

    for flag in ['-h', '--help']:
        rv, out = getstatusoutput('{} {}'.format(prg, flag))
        assert rv == 0
        assert out.lower().startswith('usage')
```

Inside `test.py`, there are two lists of words, one starting with consonants and one starting with vowels.

```
consonant_words = [
    'brigatine', 'clipper', 'dreadnought', 'frigate', 'galleon', 'haddock',
    'junk', 'ketch', 'longboat', 'mullet', 'narwhal', 'porpoise', 'quay',
    'regatta', 'submarine', 'tanker', 'vessel', 'whale', 'xebec', 'yatch',
    'zebrafish'
]
vowel_words = ['aviso', 'eel', 'iceberg', 'octopus', 'upbound']
```

There is also a string `template` for what the program should print:

```
template = 'Ahoy, Captain, {} {} off the larboard bow!'
```

The `test_consonant` test runs through each of the `consonant_words` and checks if the program puts an "a" in front of the word.

```
# ------------------------------------------------
def test_consonant():
    """brigatine -> a brigatine"""

    for word in consonant_words:
        out = getoutput('{} {}'.format(prg, word))
        assert out.strip() == template.format('a', word)
```

The next function does the same thing but uses a capitalized version of the consonant word. The next two tests then use the `vowel_words`, checking both lower- and uppercase versions.

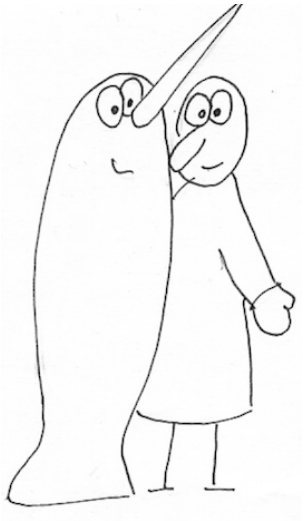
When all tests are passing, this is the output you should see:

```
$ make test
pytest -xv test.py
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 --
/Users/kyclark/anaconda3/bin/python3
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/manning/playful_python/crowsnest
plugins: openfiles-0.3.2, arraydiff-0.3, doctestplus-0.3.0, remotedata-0.3.1, cov-
2.7.1
collected 6 items

test.py::test_exists PASSED                                              [ 16%]
test.py::test_usage PASSED                                               [ 33%]
test.py::test_consonant PASSED                                           [ 50%]
test.py::test_consonant_upper PASSED                                     [ 66%]
test.py::test_vowel PASSED                                               [ 83%]
test.py::test_vowel_upper PASSED                                         [100%]

========================== 6 passed in 2.28 seconds ===========================
```

Summary

- All Python's documentation is available on https://docs.python.org/3/ and with the `help` command in the REPL.
- Variables in Python are dynamically typed according to whatever value you assign them and the come into existence when you assign a value to them.
- Strings have methods like `upper` and `isupper` that you can call to alter them or get information.
- You can get parts of a string by using square brackets and indexes like `[0]` for the first letter or `[-1]` for the last.
- You can concatenate strings with the `+` operator.
- The `str.format` method allows you to create a template with `{}` placeholders that get filled in with the arguments.
- F-strings like `f'{article} {word}'` allow variables and code to go directly inside the brackets.
- The `x in y` expression will report if the value `x` is present in the collection `y`.
- Statements like `if`/`else` do not return a value while expressions like `x if y else z` do return a value.
- Test-driven development is a way to ensure programs meet some minimum critieria of correctness. Every feature of a program should have tests, and writing and running test suites should be an integral part of writing programs.

Going Further



- Have your program match the case of the incoming word, e.g., "an octopus" and "An Octopus." Copy an existing `test_` function in the `test.py` to verify that your program works correctly while still passing all the other tests. Try writing the test first, then make your program pass the test. That's *test-driven development*!
- Accept a new parameter that changes "larboard" (the left side of the boat) to "starboard" (the right side.[1]). You could either make an option called `--side` that defaults to "larboard," or you could make a `--starboard` flag that, if present, changes the side to "starboard."
- The provided tests only give you words that start with an actual alphabetic character. Expand your code to handle words that start with numbers or punctuation. Should your program reject these? Add more tests to ensure that your program does what you intend.

[1] "Starboard" has nothing to do with stars but with the "steering board" or a rudder which typically would be on the right-side of the boat for right-handed sailors!

Chapter 3: Going on a picnic: Working with lists

Writing code makes me hungry! Let's write a program to consider some tasty foods we'd like to eat. So far we've just handled *one* of something like a name to say "hello" to or a nautical-themed object to point out. We want to eat one or more food things which we will store in a `list`, a variable that can hold any number of items. We use lists all the time in life. Maybe it's your top-five favorite songs, your birthday wish-list, or a bucket list of the best types of buckets.

In this exercise, we're going on a picnic, and we want to print a list of items to bring. You will learn to:

- Write a program that accepts multiple positional arguments
- Use `if`, `elif`, and `else` to handle conditional branching
- How to find and alter items in a list
- How to sort and reverse lists
- Format a list into a new string

The items will be passed as positional arguments. When there is only one item, you'll just print that:

```
$ ./picnic.py salad
You are bringing salad.
```

When there are two items, you'll put "and" in between them:

```
$ ./picnic.py salad chips
You are bringing salad and chips.
```

Hmm, chips. This is sounding like a pretty good picnic.

When there are three or more items, you will separate the items with commas:

```
$ ./picnic.py salad chips cupcakes
You are bringing salad, chips, and cupcakes.
```

There's one other twist. You will also need to accept a `--sorted` argument that will require you to sort the items before you print them, but we'll deal with that in a bit. So, your Python program must:

- Store one or more positional arguments in a `list`
- Count the number of arguments
- Possibly modify the list if there are three or more arguments
- Use the list to print a new a string that formats the arguments according to how many items there are.

Let's get started!

Starting the program

I will always recommend you start programming either by running `new.py` or by copying `template/template.py` to the program name. This time the program should be called `picnic.py`, and we need to create it in the `picnic` directory:

```
$ cd picnic
$ new.py picnic.py
Done, see new script "picnic.py."
```

Now run `make test` or `pytest -xv test.py`. You should pass the first two tests (program exists, program creates usage), and fail the third:

```
test.py::test_exists PASSED                                          [ 14%]
test.py::test_usage PASSED                                           [ 28%]
test.py::test_one FAILED                                             [ 42%]
```

The rest of the output is complaining about the fact that the test expected "You are bringing chips." but got something else. Run your program with the argument "chips":

```
$ ./picnic.py chips
str_arg = ""
int_arg = "0"
file_arg = ""
flag_arg = "False"
positional = "chips"
```

Remember, the template doesn't have the *correct* arguments, just some examples, so the first thing we need to do is to fix the `get_args` function. Here is what your program should print a usage statement if given *no arguments*:

```
$ ./picnic.py
usage: picnic.py [-h] [-s] str [str ...]
picnic.py: error: the following arguments are required: str
```

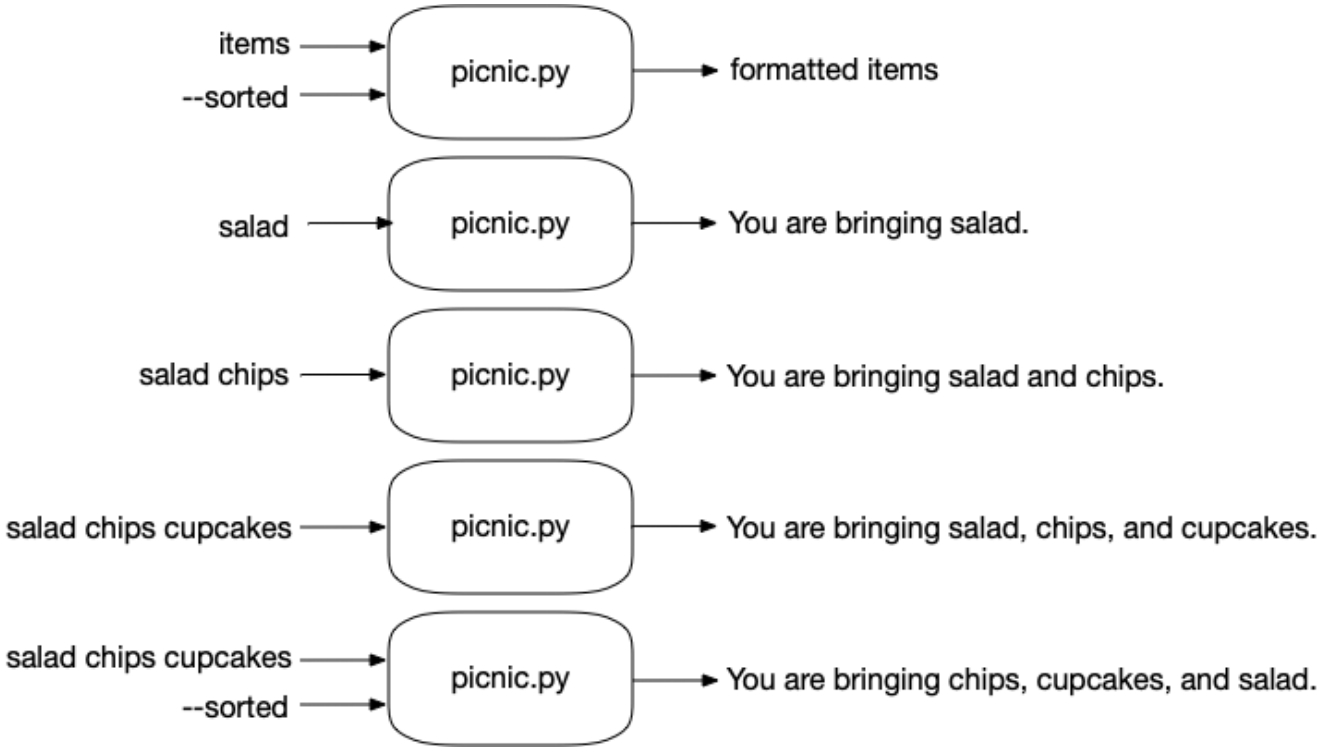
And here is the usage for the `-h` or `--help` flags:

```
$ ./picnic.py -h
usage: picnic.py [-h] [-s] str [str ...]

Picnic game

positional arguments:
  str           Item(s) to bring

optional arguments:
  -h, --help    show this help message and exit
  -s, --sorted  Sort the items (default: False)
```

We need a single positional argument and an optional flag called `--sorted`. Modify your `get_args` until it produces the above output. Note that there should be one or more of the `item` parameter, so you should define it with `nargs='+'`. Refer to the section "One or more of the same positional arguments" in chapter 1.

Writing `picnic.py`

Here is a diagram of the inputs and ouputs for the `picnic.py` program we'll write:



The program should accept one or more "positional" arguments as the items to bring on a picnic as well as a `-s` or `--sorted` *flag* to indicate whether or not to sort the items. The output will be "You are bringing" and then the list of items formatted according the following rules:

1. If one item, just state the item:

```
$ ./picnic.py chips
You are bringing chips.
```

2. If two items, put "and" in between the items. Note that "potato chips" is just one string that happens to contain two words. If we leave out the quotes, then there would be three arguments to the program. Also, it doesn't matter here if we use single or double quotes:

```
$ ./picnic.py "potato chips" salad
You are bringing potato chips and salad.
```

3. If three or more items, place a comma and space between each item and the work "and" before the final element. Don't forget the comma before the "and" (sometimes called the "Oxford comma") because your author was an English lit major and, while I may have finally stopped using two spaces after the end of a sentence, you can pry the Oxford comma from my cold, dead hands:

```
$ ./picnic.py "potato chips" salad soda cupcakes
You are bringing potato chips, salad, soda, and cupcakes.
```

Be sure to sort if given the `-s` or `--sorted` flag:

```
$ ./picnic.py --sorted salad soda cupcakes
You are bringing cupcakes, salad, and soda.
```

In order to figure out how many items we have, how to sort and slice them, and how to format the output string, we need to talk about the `list` type in Python in order to solve this problem.

Introduction to Lists

We briefly touched on the idea of a `list` in the `hello.py` program when we looked at `sys.argv`, the "argument vector" for a program. If we run this:

```
$ ./picnic.py salad chips cupcakes
```

Then the arguments `salad chips cupcakes` would be available in `sys.argv` as the `list` of the strings `['salad', 'chips', 'cupcakes']`. Note that they would be in the same order as they were provided on the command line. Lists always keep their order!

Let's go into the REPL and create a variable called `items` to hold some tasty foods we plan to bring on our picnic. I really want you to type these commands yourself, too, whether in the `python3` REPL or `ipython` or a Jupyter Notebook. It's very important to interact in real time with the language!

To create a new, empty list, we can either use the `list()` function:

```
>>> items = list()
```

Or use empty square brackets:

```
>>> items = []
```

Check what Python says for the `type`. Yep, it's a `list`:

```
>>> type(items)
<class 'list'>
```

One of the first things we need to know is how many `items` we have for our picnic. Like a `str`, we can use `len` (length) to get the number of elements in `items`.

The length of an empty `list` is `0`:

```
>>> len(items)
0
```

Adding one element to a list

An empty list is not very useful. Let's see how we can add new items. We use `help(str)` in the last chapter to read the documentation about the string *methods*, the functions that belong to every `str` in Python. Here I want you to use `help(list)` to find the `list` methods:

```
>>> help(list)
```

You'll see lots of "double-under" methods like `{dbl_}len{dbl_}`. Skip over those, and the first method we can find is `append`, which we can use to add items to the end of the list. If we evaluate our `items`, we see that the empty brackets tell us that it's empty:

```
>>> items
[]
```

Let's add "sammiches" to the end:

```
>>> items.append('sammiches')
```

Nothing happened, so how do we know that worked? Let's check the length. It should be `1`:

```
>>> len(items)
1
```

Hooray! That worked. In the spirit of testing, use the `assert` method to verify that the length is `1`. The fact that nothing happens is good. When as assertion fails, it triggers an exception that results in a lot of messages. Here, no news is good news:

```
>>> assert len(items) == 1
```

If you type `items<Enter>` in the REPL, Python will show you a the contents:

```
>>> items
['sammiches']
```

Cool, we added one element.

Adding many elements to a list

Let's try to add "chips" and "ice cream" to the `items`:

```
>>> items.append('chips', 'ice cream')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: append() takes exactly one argument (2 given)
```

Here is one of those pesky exceptions, and these will cause your programs to *crash*, something we

want to avoid at all costs. We see that `append() takes exactly one argument`, and we gave it two. If you look at the `items`, you'll see that nothing was added:

```
>>> items
['sammiches']
```

OK, so maybe we were supposed to give it a `list` of items to add? Let's try that:

```
>>> items.append(['chips', 'ice cream'])
```

Well, that didn't cause an exception, so maybe it worked? We would expect there to be 3 `items`, so let's use an assertion to check that:

```
>>> assert len(items) == 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

What is the length?

```
>>> len(items)
2
```

Only 2? Let's look at `items`:

```
>>> items
['sammiches', ['chips', 'ice cream']]
```

Check that out! Lists can hold any type of data like strings and numbers and even other lists. We asked `append` to add `['chips', 'ice cream']`, which is a `list`, and that's just what it did. Of course, it's not quite what we *wanted*.

Let's reset `items` so we can fix this. We can either use the `del` command to delete the element at index `1`:

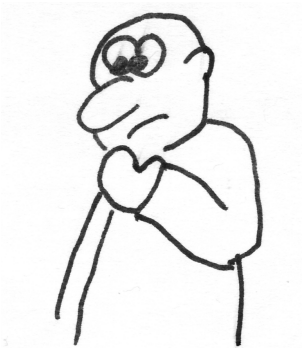
```
>>> del items[1]
```

Or reassign it a new value:

```
>>> items = ['sammiches']
```

If you read further into the `help`, you will find the `extend` method:

```
|  extend(self, iterable, /)
|      Extend list by appending elements from the iterable.
```

Let's try that: 

```
>>> items.extend('chips', 'ice cream')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: extend() takes exactly one argument (2 given)
```

Well that's frustrating! Now Python is telling us that `extend() takes exactly one argument` which, if you refer to the `help`, should be an `iterable`. A `list` is something you can iterate (travel over from beginning to end), so that will work:

```
>>> items.extend(['chips', 'ice cream'])
```

Nothing happened. No exception, so maybe that worked? Let's check the length. It *should* be 3:

```
>>> assert len(items) == 3
```

Yes! Let's look at the `items` we've added:

```
>>> items
['sammiches', 'chips', 'ice cream']
```

Great! This is sounding like a pretty delicious outing.

If you know everything that will go into the `list`, you can create it like so:

```
>>> items = ['sammiches', 'chips', 'ice cream']
```

The `append` and `extend` methods add new elements to the *end* of a given `list`. The `insert` method allows you to place new items at any position by specifying the index. I can use the index `0` to put a new element at the beginning of `items`:

```
>>> items.insert(0, 'soda')
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

In addition to `help(list)`, you can also find lots of great documentation here:

https://docs.python.org/3/tutorial/datastructures.html

I recommend you read over all the `list` functions so you get an idea of just how powerful this data structure is!

Indexing lists

So now we have a list of `items`. We know how to use `len` to find how many `items` there are, and now we need to know how to get parts of the `list` to format. Indexing a `list` in Python looks exactly the same as indexing a `str`. (This actually makes me a bit uncomfortable, so I tend to imagine a `str` as a `list` of characters and then I feel somewhat better.)

All indexing in Python is zero-offset, so the first element of `items` is at index `items[0]`:

```
>>> items[0]
'soda'
```

If the index is negative, Python starts counting backwards from the end of the list. The index `-1` is the last element of the list:

```
>>> items[-1]
'ice cream'
```

Referencing an index that is not present will cause an exception.

You should be very careful when using indexes to reference elements in a list. This is unsafe code:

```
>>> items[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

We'll soon learn how to safely *iterate* or travel through a `list` so that we don't have to use indexes to get at elements.

Slicing lists

You can extract "slices" (sub-lists) of a `list` by using `list[start:stop]`.

```
  0         1          2          3
['soda', 'sammiches', 'chips', 'ice cream']
 -4        -3         -2         -1
```

To get the first two items elements, you use `[0:2]`. Remember that the `2` is actually the index of the *third* element but *it's not inclusive*:

```
>>> items[0:2]
['soda', 'sammiches']
```

If you leave out `start`, it will be `0`, so this does the same thing:

```
>>> items[:2]
['soda', 'sammiches']
```

If you leave out `stop`, it will go to the end of the list:

```
>>> items[2:]
['chips', 'ice cream']
```

Oddly, it is completely *safe* for slices to use list indexes that don't exist. Here I can ask for all the elements from index `10` to the end even though there is nothing at index `10`. Instead of an exception, we get an empty list:

```
>>> items[10:]
[]
```

For our exercise, you're going to need to get the word "and" into the list if there are three or more elements. Could you use a list index to do that?

Finding elements in a list

Did we remember to pack the chips?! Often we want to know if some items is in a `list`. The `index` method will return the location of an element in a `list`:

```
>>> items.index('chips')
2
```

`list.index` is unsafe code because it will cause an exception if the argument is not present in the list!

See what happens if we check for the fog machine:

```
>>> items.index('fog machine')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'fog machine' is not in list
```

You should never use `index` unless you have first verified that an element is present. The `x in y` that we used in "Crow's Nest" to see if a letter was in the list of vowels can also be used for lists. We get back a `True` value if `x` is `in` the collection of `y`:

```
>>> 'chips' in items
True
```

I hope they're salt and vinegar chips.

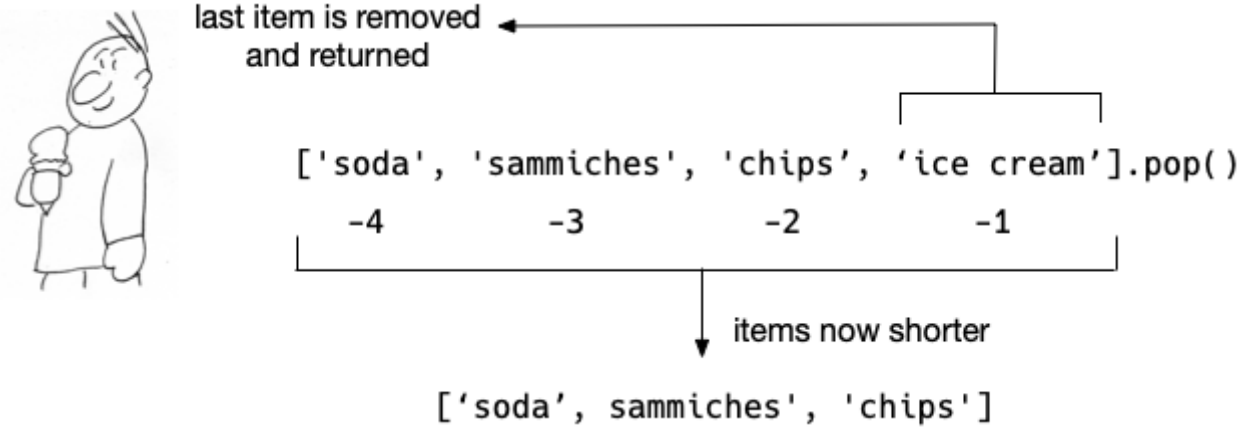
The same returns `False` if it is not present:

```
>>> 'fog machine' in items
False
```

We're going to need to talk to the planning committee. What's a picnic without a fog machine?

Removing elements from a list

We've seen that we can use the `del` function to delete a `list` element by index. The `list.pop` method will remove *and return* the element at the index. By default it will remove the *last* item (`-1`):

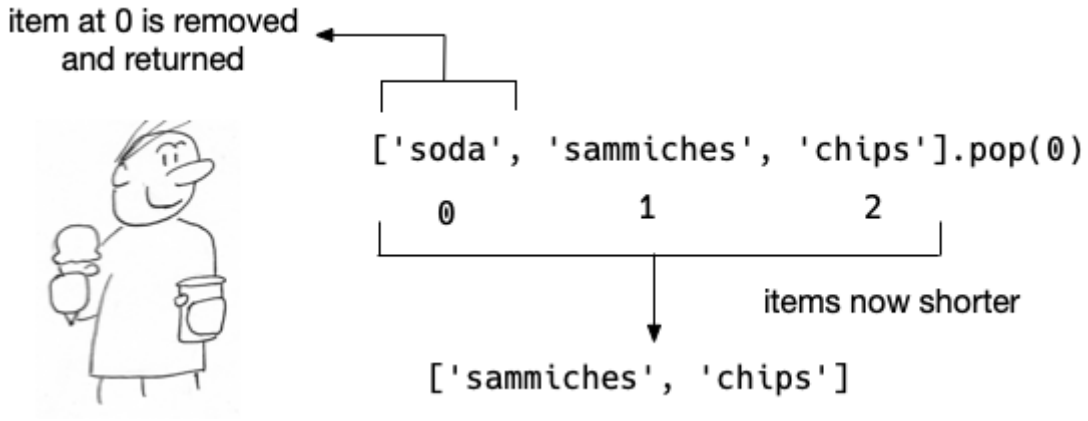


```
>>> items.pop()
'ice cream'
```

If we look at the list, we will see it's shorter by one:

```
>>> items
['soda', 'sammiches', 'chips']
```

You can use an item's index to remove an element at a particular location. For instance, we can use `0` to remove the first element:

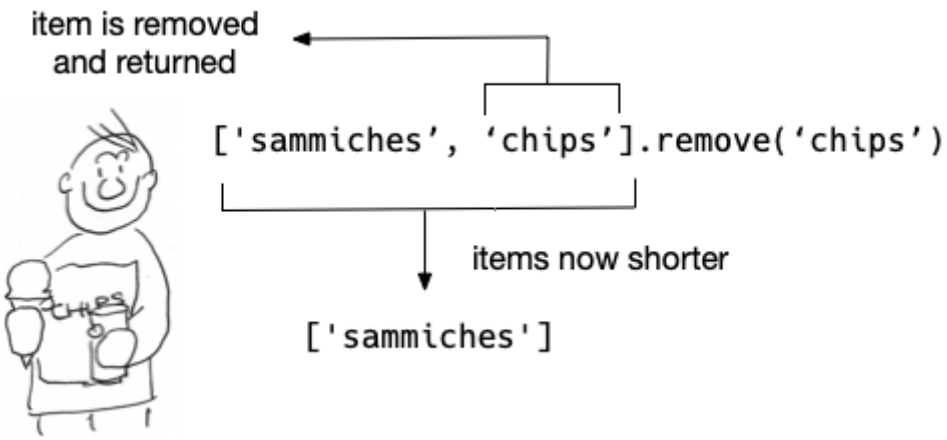



```
>>> items.pop(0)
'soda'
```

And now our `list` is shorter still:

```
>>> items
['sammiches', 'chips']
```

You can also use the `list.remove` method to remove the first occurrence of a given item:



```
>>> items.remove('chips')
>>> items
['sammiches']
```

The `list.remove` will cause an exception if the element is not present.

If we try to `remove` the chips again, we get an exception:

```
>>> items.remove('chips')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

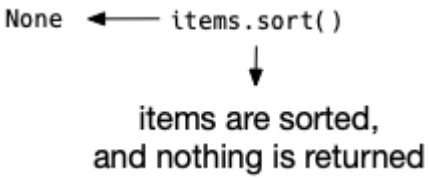
So don't use this code unless you've verified that a given element is in the list:

```
>>> if 'chips' in items:
...     items.remove('chips')
...
```

Sorting and reversing a list

If the `--sorted` flag is present, we're going to need to sort the `items`. You might notice in the help documentation that two methods, `list.reverse` and `list.sort` stress that they work *IN PLACE*. That means that the `list` itself will be either reversed or sorted and *nothing will be returned*. So, given this list:

```
>>> items = ['soda', 'sammiches', 'chips', 'ice cream']
```



The `sort` method will return nothing:

```
>>> items.sort()
```

But if you inspect `items`, you will see they have been sorted alphabetically:

```
>>> items
['chips', 'ice cream', 'sammiches', 'soda']
```

Note that Python will sort a `list` of numbers *numerically*, so we've got that going for us, which is nice:

```
>>> sorted([4, 2, 10, 3, 1])
[1, 2, 3, 4, 10]
```

As with `list.sort`, we see nothing on the `list.reverse` call:

```
>>> items.reverse()
```

But the `items` are now in the opposite order:

```
>>> items
['soda', 'sammiches', 'ice cream', 'chips']
```

The `list.sort` and `list.reverse` *methods* are easily confused with the the `sorted` and `reversed` *functions*. The `sorted` *function* accepts a `list` as an argument and *returns* a new `list`:

```
>>> items = ['soda', 'sammiches', 'chips', 'ice cream']
>>> sorted(items)
['chips', 'ice cream', 'sammiches', 'soda']
```

It's crucial to note that the `sorted` function *does not alter* the `list`:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

The `list.sort` *method* is a function that belongs to the `list`. It can take arguments that affect the way the sorting happens. Let's look at the `help(list.sort)`:

```
sort(self, /, *, key=None, reverse=False)
    Stable sort *IN PLACE*.
```

So we could also `sort` the `items` in `reverse` like so:

```
>>> items.sort(reverse=True)
```

And now they look like this:

```
>>> items
['soda', 'sammiches', 'ice cream', 'chips']
```

The `reversed` function works a bit differently:

```
>>> reversed(items)
<list_reverseiterator object at 0x10e012ef0>
```

I bet you expecting to see a new `list` with the `items` in reverse? This is an example of a *lazy* function in Python. The process of reversing a `list` might take a while, so Python is showing that it has generated an "iterator object" that will provide the reversed list just as soon as we actually ask for the elements. We can do that in the REPL by using the `list` function to evaluate the iterator:

```
>>> list(reversed(items))
['ice cream', 'chips', 'sammiches', 'soda']
```

As with the `sorted` function, the original `items` itself remain unchanged:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

If you use the `list.sort` method instead of the `sorted` function, you might end up deleting your data. Imagine you wanted to set your `items` equal to the sorted list of `items` like so:

```
>>> items = items.sort()
```

What is in `items` now? If you print the `items` in the REPL, you won't see anything useful, so inspect the `type`:

```
>>> type(items)
<class 'NoneType'>
```

It's no longer a `list`. We just set it equal to the result of called `items.sort()` method that works on the `list` *in-place* and returns `None`. I would note that I tend to not use the `sort`/`reverse` methods because I don't generally like to mutate my data. I would tend to do something like this:

```
>>> items = ['soda', 'sammiches', 'chips', 'ice cream']
>>> sorted_items = sorted(items)
>>> sorted_items
['chips', 'ice cream', 'sammiches', 'soda']
```

Now I have explictly named a `sorted_items` list, and the original `items` has not been altered.

If the `--sorted` flag is given to your program, you will need to sort your items in order to pass the test. Will you use `list.sort` or the `sorted` function?

Lists are mutable

As we've seen, we can change a `list` quite easily. The `list.sort` and `list.reverse` methods change the whole list, but you can also change any single element by referencing it by index. Maybe we make our picnic slightly healthier by changing out the chips for apples:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
>>> if 'chips' in items:            ①
...     idx = items.index('chips')  ②
...     items[idx] = 'apples'       ③
...
```

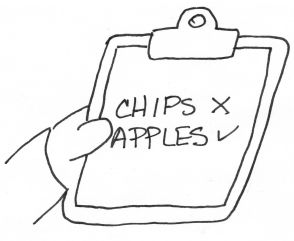
① See if the string `'chips'` is in the list of `items`.

② Assign the index of `'chips'` to the variable `idx`.

③ Use the index `idx` to change the element to `'apples'`.

Let's look at `items` to verify:

```
>>> items
['soda', 'sammiches', 'apples', 'ice cream']
```



We can also write a couple of tests:

```
>>> assert 'chips' not in items ①
>>> assert 'apples' in items    ②
```

① Make sure "chips" are no longer on the menu.

② Check that we now have some "apples."

Joining a `list`

In our exercise, you'll need to print a string based on the number of elements in the given list. The string will intersperse some other string like `', '` in between all the elements of the list. Oddly, this the syntax to join a list on the string made of the comma and a space:

```
>>> ', '.join(items)
'soda, sammiches, chips, ice cream'
```

Here we use the `str.join` method and pass the `list` as an argument. It always feels backwards to me, but that's the way it goes. The result of `str.join` is a *new string*:

```
>>> type(', '.join(items))
<class 'str'>
```

The original `list` remains unchanged:

```
>>> items
['soda', 'sammiches', 'chips', 'apples']
```

There is quite a bit more that we can do with Python's `list`, but that should be enough for you to solve this problem.

Conditional branching with if/elif/else

You need to use the conditional branching based on the number of items to correctly format the output. In the Crow's Nest exercise, there were two conditions (a "binary" choice) — either a vowel or not — so we used `if`/`else` statements. Here we have three options to consider, so you will have to use `elif` (else-if). For instance, we want to classify someone by their age by three options:

1. If their age is greater than `0`, it is valid.
2. If their age is less than `18`, they are a minor.
3. Otherwise they are 18 year or older, which means they can vote:

Here is how I could write that code:

```
>>> age = 15
>>> if age < 0:
...     print('You are impossible.')
... elif age < 18:
...     print('You are a minor.')
... else:
...     print('You can vote.')
...
You are a minor.
```

See if you can use that to figure out how to write the three options for `picnic.py`. That is, first write the branch that handles one item. Then write the branch that handles two items. Then write the last branch for three or more items. Run the tests *after every change to your program*.

Now go write the program yourself before you continue to look at my solution.

Hints:

- Go into your `picnic` directory and run `new.py picnic.py` to start your program. Then run `make test` (or `pytest -xv test.py`) and you should pass the first two tests.
- Next work on getting your `--help` usage looking like the above. It's very important to define your arguments correctly. For the `items` argument, look at `nargs` in `argparse` as discussed in chapter 1's "One or more of the same positional arguments" section.
- If you use `new.py` to start your program, be sure to leave the "boolean flag" and modify it for your `sorted` flag.
- Solve the tests in order! First handle just one item, then handle two items, then handle three. Then handle the sorted items.

You'll get the best benefit from this book if you try writing the program and passing the tests before turning the page to read the solution!

Solution

```
#!/usr/bin/env python3
"""Picnic game"""

import argparse


# --------------------------------------------------
def get_args(): ①
    """Get command-line arguments"""

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

    parser.add_argument('item',
                        metavar='str',
                        nargs='+',
                        help='Item(s) to bring')

    parser.add_argument('-s',
                        '--sorted',
                        action='store_true',
                        help='Sort the items')

    return parser.parse_args()


# --------------------------------------------------
def main(): ②
    """Make a jazz noise here"""

    args = get_args()
    items = args.item
    num = len(items)

    if args.sorted:
        items.sort()

    bringing = ''
    if num == 1:
        bringing = items[0]
    elif num == 2:
        bringing = ' and '.join(items)
    else:
        bringing = ', '.join(items[:-1] + ['and ' + items[-1]])

    print('You are bringing {}.'.format(bringing))
```

```
48
49
50  # --------------------------------------------------
51  if __name__ == '__main__':
52      main() ③
```

① The `get_args` function is placed first so that we can easily see what the program accepts when we read it. Note that the function order here is not important to Python, only to us, the reader.

② The `main` function is where the program will start.

③ This calls the `main` function when the program is executing.

Discussion

How did it go? Did it take you long to write your version? How different was it from mine? Let's talk about my solution. It's perfectly fine if yours is really different from mine, just as long as you pass the tests!

Defining the arguments

This program can accept a variable number of arguments which are all the same thing (strings). In my `get_args`, I define an `item` like so:

```
parser.add_argument('item',                 ①
                    metavar='str',          ②
                    nargs='+',              ③
                    help='Item(s) to bring') ④
```

① A single, required, positional (because no dashes in name) argument called `item`.

② An indicator to the user in the usage that this should be a string.

③ The number of arguments where `'+'` means *one or more*.

④ A longer help description that appears for the `-h` or `--help` options.

This program also accepts `-s` and `--sorted` arguments. Remember that the leading dashes makes them optional. They are "flags," which typically means that they are `True` if they are present and `False` if absent.

```
parser.add_argument('-s',                   ①
                    '--sorted',             ②
                    action='store_true',    ③
                    help='Sort the items') ④
```

① The short flag name.

② The long flag name.

③ If the flag is present, store a `True` value. The default value will be `False`.

④ The longer help description.

Assigning and sorting the items

To get the arguments, in `main` I call `get_args` and assign them to the `args` variable. Then I create the `items` variable to hold the `args.item` value(s):

```
args = get_args()
items = args.item
```

If `args.sorted` is `True`, then I need to sort my `items`. I chose the *in-place* `sort` method here:

```
if args.sorted:
    items.sort()
```

Now I have the items, sorted if needed, and I need to format them for the output.

Formatting the items

I suggested you solve the tests in order. There are 4 conditions we need to solve:

1. Zero items
2. One item
3. Two items
4. Three or more items

The first test is actually handled by `argparse` — if the user fails to provide any arguments, they get a usage:

```
$ ./picnic.py
usage: picnic.py [-h] [-s] str [str ...]
picnic.py: error: the following arguments are required: str
```

Since `argparse` handles the case of no arguments, we have to handle the other three conditions. Here's one way to do that:

```
bringing = ''                       ①
if num == 1:                        ②
    bringing = items[0]             ③
elif num == 2:                      ④
    bringing = ' and '.join(items) ⑤
else:                               ⑥
    items[-1] = 'and ' + items[-1] ⑦
    bringing = ', '.join(items)    ⑧
```

① Initialize a variable for what we are `bringing`.

② Check if the number of items is one.

③ If there is one item, then `bringing` is the one item.

④ Check if the number of items is two.

⑤ If two items, join the `items` on the string `' and '`.

⑥ Otherwise...

⑦ Insert the string `'and '` before the last item

⑧ Join all the `items` on the string `', '`.

Printing the items

Finally to `print` the output, I can use a format string where the `{}` indicates a placeholder for some value like so:

```
>>> print('You are bringing {}.'.format(bringing))
You are bringing salad, soda, and cupcakes.
```

Or, if you prefer, you can use an `f''`-string:

```
>>> print(f'You are bringing {bringing}.')
You are bringing salad, soda, and cupcakes.
```

They both get the job done, so whichever you prefer.

Testing

For this exercise, I've written the tests for you. Can you see how this is similar to the `hello.py` where we created a `greet` function inside the program and added a `test_greet` function to test it? If I were writing this code for myself, I would do the same by creating a function to format the items and placing the unit test for that inside the program itself. As it is, the `test.py` I'm providing conflates such a unit test with an overall integration test where I check that the program responds to the command line options properly.

I believe there is an art to testing. It's up to you to figure out how best to test your own code. Try copying some of the relevant `test_` functions from `test.py` into your `picnic.py` and then running `pytest picnic.py` to see how it works. What do you prefer?

Summary

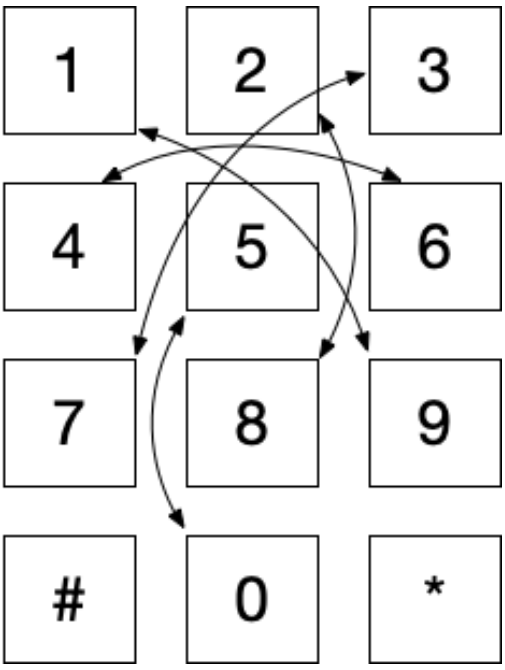
- Python lists are ordered sequences of other Python data types such as strings and numbers.
- There are methods like `append` and `extend` to add elements to a `list` and `pop` and `remove` to remove them.
- You can use `x in y` to ask if element `x` is in the list `y`. You could also use `list.index` to find the index of an element, but this will cause an exception if the element is not present.
- Lists can be sorted and reversed, and elements within lists can be modified. Lists are useful when the order of the elements is important.
- Strings and lists share many features such as using `len` to find their lengths, using zero-based indexing where `0` is the first element and `-1` is the last, and using slices to extract smaller pieces from the whole.
- The `str.join` method can be used to make a new `str` from a `list`.
- `if`/`elif`/`else` can be used to branch code depending on conditions.

Going Further

- Add an option so the user can choose not to print with the Oxford comma.
- Add an option to separate items with some character passed in by the user (like a semicolon if the list of items needed to contain commas).

Chapter 4: Jump the Five (Working with dictionaries)

> "When I get up, nothing gets me down." - D. L. Roth



In an episode of the television show *The Wire*, drug dealers encode telephone numbers that they text in order to obscure them from the police who they assume are intercepting their messages. They use an algorithm we'll call "Jump The Five" where a number is changed to the one that is opposite on a US telephone pad if you jump over the 5.

If we start with "1" and jump across the 5, we get to "9," then "6" jumps the 5 to become "4," and so forth. The numbers "5" and "0" will swap with each other. In this exercise, we're going to write a Python program called `jump.py` that will take in some text as a positional argument. Each number in the text will be encoded using this algorithm. All non-number will pass through unchanged, for example:

```
$ ./jump.py 867-5309
243-0751
$ ./jump.py 'Call 1-800-329-8044 today!'
Call 9-255-781-2566 today!
```

We will need some way to inspect each character in the input text and identify the numbers. We will learn how to use a `for` loop for this and how that relates to a "list comprehension." Then we will need some way to associate a number like "1" with the number "9," and so on for all the numbers. We'll learn about a data structure in Python called a "dictionary" type that allows us to do exactly that.

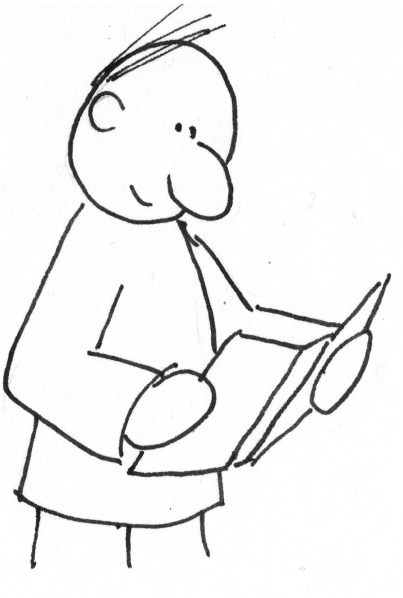
In this chapter, you will learn to:

- Create a dictionary.

- Use a `for` loop to process text character-by-character.
- Check if items exist in a dictionary.
- Retrieve values from a dictionary.
- Print a new string with the numbers substituted for their encoded values.

Before we get started with the coding, let's spend some time learning about dictionaries.

Dictionaries



A Python `dict` allows us to relate some *thing* (a "key") to some other *thing* (a "value"). An actual dictionary does this. If we look up a word like "quirky" in a dictionary (https://www.merriam-webster.com/dictionary/quirky), we can find a definition. We can think of the word itself as the "key" and the definition as the "value."

quirky ➪ unusual, esp. in an interesting or appealing way

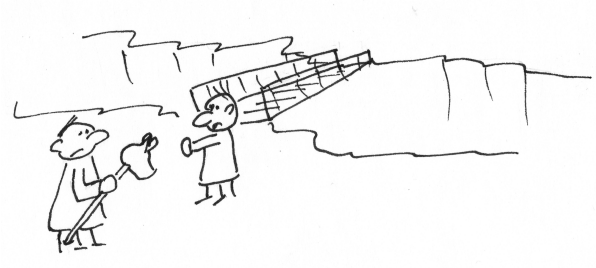
Dictionaries actually provide quite a bit more information such as pronunciation, part of speech, derived words, history, synonyms, alternate spellings, etymology, first known use, etc. (I really love dictionaries.) Each of those attributes has a value, so we could also think of the lookup itself as a dictionary:

quirky

definition	➪ unusual, esp. in an interesting or appealing way
pronunciation	➪ ˈkwər-kē
part of speech	➪ adjective

Let's see how we can use Python's dictionaries to go beyond word definitions.

Creating a dictionary, setting values



In the film *Monty Python and the Holy Grail*, King Arthur and his knights must cross The Bridge of Death. Anyone who wishes to cross must correctly answer three questions from the Keeper. Those who fail are cast into the Gorge of Eternal Peril.

Let's create and use a `dict` to keep track of the questions and answers as key/value pairs.

Lancelot goes first. We can either use the `dict()` function to create an empty dictionary for his answers. Once again, I want you to fire up your `python3`/`ipython` REPL or Jupyter Notebook and type these out for yourself!

```
>>> answers = dict()
```

Or we can use empty curly brackets:

```
>>> answers = {}
```

The Keeper's first question: "What is your name?" Lancelot answers "My name is Sir Lancelot of Camelot." We can add the key "name" to our `answers` by using square brackets (`[]` — not curlies!) and the literal string `'name'`:

```
>>> answers['name'] = 'Sir Lancelot'
```

If you type `answers<Enter>` in the REPL, Python will show you a structure in curlies to indicate this is a `dict`:

```
>>> answers
{'name': 'Sir Lancelot'}
```

You can verify with the `type` function:

```
>>> type(answers)
<class 'dict'>
```

Next the Keeper asks, "What is your quest?" to which Lancelot answers "To seek the Holy Grail." Let's add "quest" to the `answers`:

```
>>> answers['quest'] = 'To seek the Holy Grail'
```

There's no return value to let us know something happened, so we can inspect the variable again to ensure our new key/value was added:

```
>>> answers
{'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail'}
```

Finally the Keeper asks "What is your favorite color?," and Lancelot answers "blue."

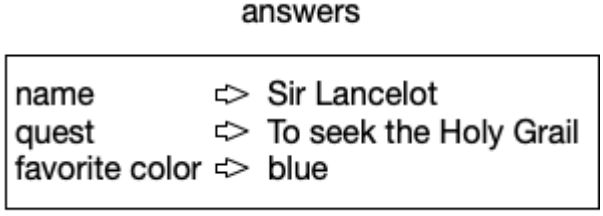
```
>>> answers['favorite_color'] = 'blue'
>>> answers
{'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail', 'favorite_color': 'blue'}
```

If you knew all the answers beforehand, you could create `answers` using the `dict()` function with this syntax where you do *not* have to quote the keys and the keys are separate from the values with equal signs:

```
>>> answers = dict(name='Sir Lancelot', quest='To seek the Holy Grail',
favorite_color='blue')
```

Or this syntax using curlies `{}` where the keys must be quoted and are followed by a colon (`:`):

```
>>> answers = {'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail',
'favorite_color': 'blue'}
```



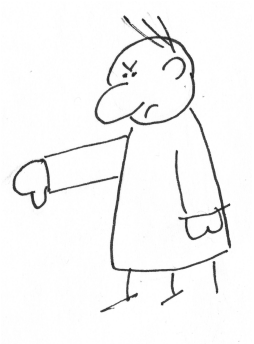
It might be helpful to think of the dictionary `answers` as a box that inside holds the key/value pairs that describe Lancelot's answers.

Accessing dictionary values

To retrieve the values, you use the key name inside square brackets (`[]`). For instance, I can get the `name` like so:

```
>>> answers['name']
'Sir Lancelot'
```

You will cause an exception if you ask for a dictionary key that doesn't exist!



```
>>> answers['age']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
```

Just as with lists, you can use the `x in y` to first see if a key exists in the `dict`:

```
>>> 'quest' in answers
True
>>> 'age' in answers
False
```

The `dict.get` method is a *safe* way to ask for a value:

```
>>> answers.get('quest')
'To seek the Holy Grail'
```

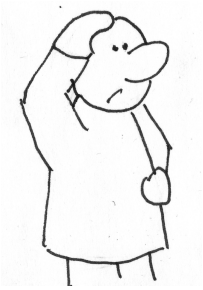
When the requested key does not exist in the `dict`, it will return the special value `None`:

```
>>> answers.get('age')
```

That doesn't print anything because the REPL won't print a `None`, but we can check the `type`:

```
>>> type(answers.get('age'))
<class 'NoneType'>
```

There is an optional second argument you can pass to `dict.get` which is the value to return *if the key does not exist*:



```
>>> answers.get('age', 'NA')
'NA'
```

Other dictionary methods

If you want to know how "big" a dictionary is, the `len` (length) function on a `dict` will tell you how many key/value pairs are present:

```
>>> len(answers)
3
```

The `dict.keys` method will give you just the keys:

```
>>> answers.keys()
dict_keys(['name', 'quest', 'favorite_color'])
```

And `dict.values` will give you the values:

```
>>> answers.values()
dict_values(['Sir Lancelot', 'To seek the Holy Grail', 'blue'])
```

Often we want both together, so you might see code like this:

```
>>> for key in answers.keys():
...     print(key, answers[key])
...
name Sir Lancelot
quest To seek the Holy Grail
favorite_color blue
```

An easier way to write this would be to use the `dict.items` method which will return a `list` of the key/value pairs:

```
>>> answers.items()
dict_items([('name', 'Sir Lancelot'), ('quest', 'To seek the Holy Grail'),
('favorite_color', 'blue')])
```

The above `for` loop could also be written using `items` and also a bit of string formatting to make the `key` be printed in a column 15 characters wide:

```
>>> for key, value in answers.items():
...     print(f'{key:15} {value}')
...
name            Sir Lancelot
quest           To seek the Holy Grail
favorite_color  blue
```

In the REPL you can execute `help(dict)` to see all the methods available to you like `pop` to remove a key/value or `update` to merge with another `dict`.

Each key in the `dict` is unique. That means if you set a value for a given key twice:

```
>>> answers = {}
>>> answers['favorite_color'] = 'blue'
>>> answers
{'favorite_color': 'blue'}
```

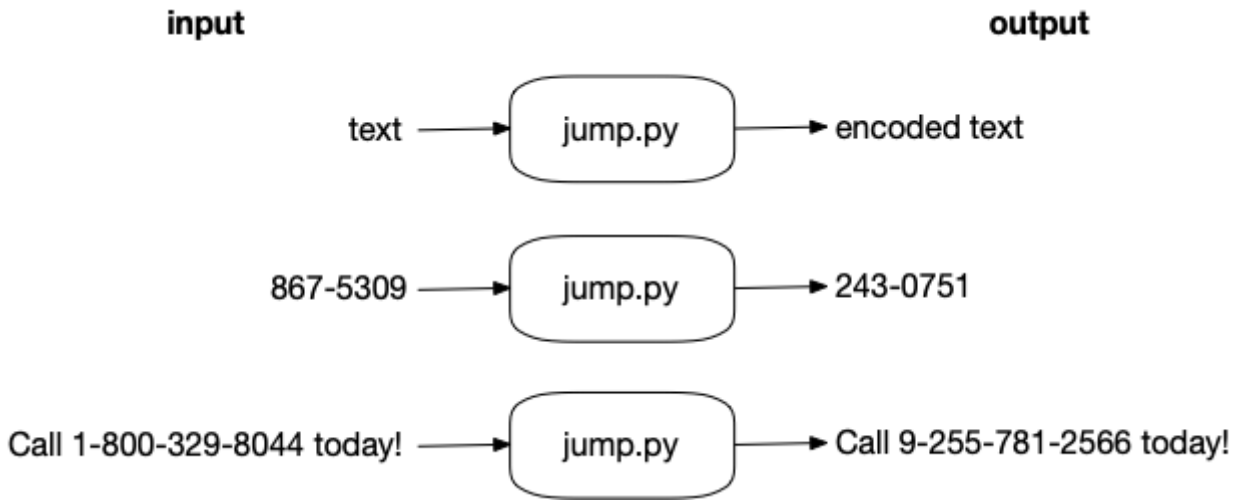
You will not have two entries but one entry with the *second* value:

```
>>> answers['favorite_color'] = 'red'
>>> answers
{'favorite_color': 'red'}
```

Keys don't have to be strings — you can also use numbers like `int` and `float`. Whatever value you use must be immutable. A `list` could not be a key because it is mutable, but a `tuple` (which is an immutable `list`) can be a key.

Writing `jump.py`

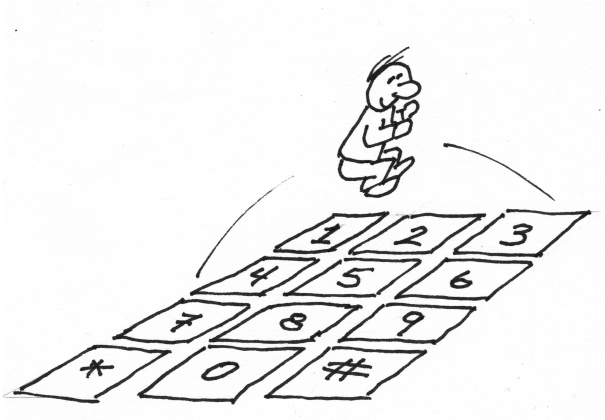
Now let's get started with writing our program. Here is a diagram of the inputs and outputs. Note that your program will only affect the numbers in the text. Anything that is *not* a number is unchanged:



When run with no arguments or `-h|--help`, your program should print a usage:

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

Jump the Five

positional arguments:
  str         Input text

optional arguments:
  -h, --help  show this help message and exit
```

Here is the substitution table for the numbers:



```
1 => 9
2 => 8
3 => 7
4 => 6
5 => 0
6 => 4
7 => 3
8 => 2
9 => 1
0 => 5
```

How would you represent this using a `dict`? Try creating a `dict` called `jump` in the REPL and then using a test. Remember that `assert` will return nothing if the statement is `True`:

```
>>> assert jumper['1'] == '9'
>>> assert jumper['5'] == '0'
```

Next, you will need a way to visit each character in the given text. I suggest you use a `for` loop like so:

```
>>> for char in 'ABC123':
...     print(char)
...
A
B
C
1
2
3
```

Now, rather printing the `char`, print the value of `char` in the `jumper` table or print the `char` itself. Look at the `dict.get` method! Also, if you read `help(print)`, you'll see there is an `end` option to change the newline that gets stuck onto the end to something else.

Hints:

- The numbers can occur anywhere in the text, so I recommend you process the input character-by-character with a `for` loop.
- Given any one character, how can you look it up in your table?
- If the character is in your table, how can you get the value (the translation)?
- If how can you `print` the translation or the value without printing a newline? Look at `help(print)` in the REPL to read about the options to `print`.
- If you read `help(str)` on Python's `str` class, you'll see that there is a `replace` method. Could you use that?

Now spend the time to write the program on your own before you look at the solutions! Use the tests to guide you.

Solution

```
#!/usr/bin/env python3
"""Jump the Five"""

import argparse


# --------------------------------------------------
def get_args(): ①
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Jump the Five',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input text') ②

    return parser.parse_args()


# --------------------------------------------------
def main(): ③
    """Make a jazz noise here"""

    args = get_args() ④
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0', ⑤
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}

    # Method 1: for loop with print
    for char in args.text: ⑥
        print(jumper.get(char, char), end='') ⑦
    print() ⑧


# --------------------------------------------------
if __name__ == '__main__':
    main() ⑨
```

① Define the `get_args` function first so it's easy to see when we read the program.

② We define one positional argument called `text`.

③ Define a `main` function where the program starts.

④ Get the command-line `args`.

⑤ Create a `dict` for the lookup table.

⑥ Process each character in the text.

⑦ Print either the value of the `char` in the `jumper` table or the `char` if it's not present, making sure

not to print a newline by adding `end=''`.

⑧ Print a newline.

⑨ Call the `main()` function if the program is in the "main" namespace.

Discussion

Defining the arguments

If you look at the solution, you'll see that the `get_args` function is defined first. Our program needs to define one positional argument. Since it's some "text" I expect, I call the argument `'text'` and then assign that to a variable called `text`.

```
parser.add_argument('text', metavar='str', help='Input text')
```

While all that seems rather obvious, I think it's very important to name things *what they are*. That is, please don't leave the name of the argument as `'positional'` — that does not describe what it *is*. It may seem like overkill to use `argparse` for such a simple program, but it handles the validation of the correct *number* and *type* of arguments as well as the generation of help documentation, so it's well worth the effort.

Using a `dict` for encoding

I suggested you could represent the substitution table as a `dict` where each number `key` has its substitute as the `value` in the `dict`. For instance, I know that if I jump from `1` over the `5` I should land on `9`:

```
>>> jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
...           '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
>>> jumper['1']
'9'
```

Since there are only 10 numbers to encode, this is probably the easiest way to write this. Note that the numbers are written with quotes around them, so they are actually of the type `str` and not `int` (integers). I do this because I will be reading characters from a `str`. If we stored them as actual numbers, I would have to coerce the `str` types using the `int` function:

```
>>> type('4')
<class 'str'>
>>> type(4)
<class 'int'>
>>> type(int('4'))
<class 'int'>
```

Method 1: Using a `for` loop to `print` each character

As suggested in the introduction, I can process each character of the `text` using a `for` loop. To start, I might first just see if each character of the text is in the `jumper` table using the `x in y` construct.

```
>>> text = 'ABC123'
>>> for char in text:
...     print(char, char in jumper)
...
A False
B False
C False
1 True
2 True
3 True
```

When `print` is given more than one argument, it will put a space in between each of bit of text. You can change that with the `sep` argument. Read `help(print)` to learn more.

Now let's try to translate the numbers. I could use an `if` expression where I print the value from the `jumper` table if `char` is present, otherwise print the `char`:

```
>>> for char in text:
...     print(char, jumper[char] if char in jumper else char)
...
A A
B B
C C
1 9
2 8
3 7
```

It's a bit laborious to check for every character. The `dict.get` method allows us to safely ask for a value if it is present. For instance, the letter "A" is not in `jumper`. If we try to retrieve that value, we'll get an exception:

```
>>> jumper['A']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'A'
```

But if we use `jumper.get`, there is no exception:

```
>>> jumper.get('A')
```

When a key doesn't exist in the dictionary, the special `None` value is returned:

```
>>> for char in text:
...     print(char, jumper.get(char))
...
A None
B None
C None
1 9
2 8
3 7
```

We can provide a second, optional argument to `get` that is the default value to return when the key does not exist. In our case, if a character does not exist in the `jumper`, we want to print the character itself. If we had "A," then we'd want to print "A":

```
>>> jumper.get('A', 'A')
'A'
```

But if we have, "5" then we want to print "0":

```
>>> jumper.get('5', '5')
'0'
```

So we can use that to process all the characters:

```
>>> for char in text:
...     print(jumper.get(char, char))
...
A
B
C
9
8
7
```

I don't want that newline printing after every character, so I can use `end=''` to tell Python to put the empty string at the `end` instead of a newline. When I run this in the REPL, the output is going to look funny because I have to hit <Enter> after the `for` loop for it to run, then I'll be left with `ABC987` with no newline and then the `>>>` prompt:

```
>>> for char in text:
...     print(jumper.get(char, char), end='')
...
ABC987>>>
```

And so in your code you have to add another `print()`. Mostly I wanted to point out a couple things you maybe didn't know about `print`. One is that you can have it put whatever you like for the `end`, and that you can `print()` with no arguments to print a newline. There are several other really cool things `print` can do, so I'd encourage you to read `help(print)` and try them out!

Method 2: Using a `for` loop to build a new string

There are several other ways we could solve this. I don't like all the `print` statements in the first solution, so here's another take:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}

    # Method 2: for loop to build new string
    new_text = ''                            ①
    for char in args.text:                   ②
        new_text += jumper.get(char, char) ③
    print(new_text)                          ④
```

① In this alternate solution, you create an empty `new_text` variable.

② Same `for` loop…

③ Append either the encoded number or the original `char` to the `new_text`

④ Print the `new_text`.

While it's fun to explore all the things we can do with `print`, that code is a bit ugly. I think it's cleaner to create a `new_text` variable and call `print` just once with that. To do this, we start off by setting a `new_text` equal to the empty string:

```
>>> new_text = ''
```

And we use our same `for` loop to process each character in the `text`. Each time through the loop, we use `+=` to append the right-hand side of the equation to the left-hand side. The `+=` adds the value on the right to the variable on the left:

```
>>> new_text += 'a'
>>> assert new_text == 'a'
>>> new_text += 'b'
>>> assert new_text == 'ab'
```

On the right, we're using the `jumper.get` method. Each character will be appended to the `new_text`:

```
>>> new_text = ''
>>> for char in text:
...     new_text += jumper.get(char, char)
...
```

Now we can call `print` one time with our new value:

```
>>> print(new_text)
ABC987
```

Method 3: Using a `for` loop to build a new `list`

This method is the same as above, but, rather than `new_text` being a `str`, it's a `list`:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}

    # Method 3: for loop to build new list
    new_text = []                                ①
    for char in args.text:                       ②
        new_text.append(jumper.get(char, char)) ③
    print(''.join(new_text))                     ④
```

① Initialize `new_text` as an empty `list`.

② Iterate through each character of the `text`.

③ Append the results of the `jumper.get` call to the `new_text` variable.

④ Join the `new_text` on the empty string to create a new `str` to print.

As we go through the book, I'll keep reminding you how Python treats strings and lists similarly. It's easy to go back and forth between those two types. Here I'm using `new_text` exactly the same as above, starting off with it empty and then making it longer for each character. We could actually use the exact same `+=` syntax instead of the `list.append` method:

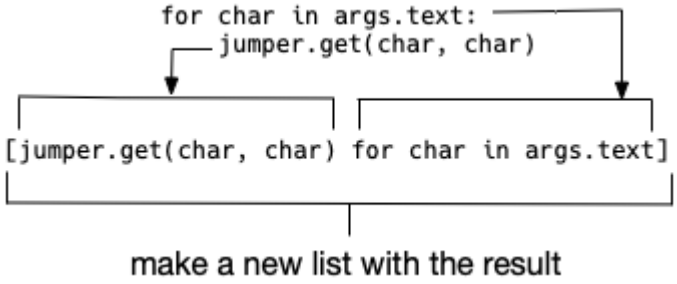
```
for char in args.text:
    new_text += jumper.get(char, char)
```

After the `for` loop is done, we have all the new characters that need to be put back together into a new string to print.

Method 4: List comprehension

A shorter solution uses a "list comprehension" which is basically a one-line `for` loop inside square brackets (`[]`) which results in a new `list`:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}

    # Method 4: list comprehension
    print(''.join([jumper.get(char, char) for char in args.text]))
```



A list comprension is read backwards from a `for` loop, but it's all there. It's just one line of code instead four!

```
>>> text = '867-5309'
>>> [jumper.get(char, char) for char in text]
['2', '4', '3', '-', '0', '7', '5', '1']
```

You can `join` that new `list` on the empty string to create a new string you can `print`:

```
>>> print(''.join([jumper.get(char, char) for char in text]))
243-0751
```

Method 5: str.translate

This last method uses a really powerful method from the `str` class to change all the characters in one step:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}

    # Method 5: str.translate
    print(args.text.translate(str.maketrans(jumper)))
```

The argument to `str.translate` is a translation table that defines how each character should be translated. That's exactly what our `jumper` does!

```
>>> text = 'Jenny = 867-5309'
>>> text.translate(str.maketrans(jumper))
'Jenny = 243-0751'
```

We'll revisit this function in the "Apples and Bananas" exercise where I'll explain it in greater detail.

(Not) using `str.replace`

Note that you could *not* use `str.replace` to change each number 0-9. Watch how we start off with this string:

```
>>> text = '1234567890'
```

When you change "1" to "9," now you have two 9s:

```
>>> text = text.replace('1', '9')
>>> text
'9234567890'
```

Which means when you try to change all the 9s to 1s, you end up with two 1s:

```
>>> text = text.replace('9', '1')
>>> text
'1234567810'
```

You might try to write it like so:

```
>>> text = '1234567890'
>>> for n in jumper.keys():
...     text = text.replace(n, jumper[n])
...
>>> text
'1234543215'
```

But the correctly encoded string is "9876043215", which is exactly why `str.translate` exists!

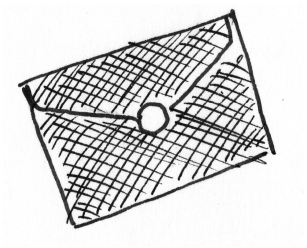
Summary

- You create a new dictionary using the `dict()` function or with empty curly brackets (`{}`).
- Dictionary values are retrieved using their keys inside square brackets or by using the `dict.get` method.
- For a `dict` called `x`, you can use `'key' in x` to determine if a key exists.
- You can use a `for` loop to iterate the characters of a `str` just like you can iterate through the elements of a `list`. You can think of strings as lists of characters.
- The `print` function takes optional keyword arguments like `end=''` which we can use to print a value to the screen without a newline.

Going Further

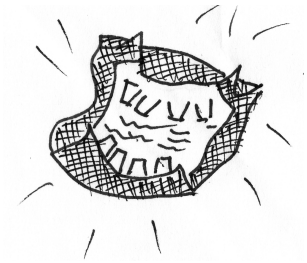
- Try creating a similar program that encodes the numbers with strings, e.g., "5" becomes "five", "7" becomes "seven."
- What happens if you feed the output of the program back into itself. For example, if you run `./jump.py 12345`, you should get `98760`. If you run `./jump.py 98760`, do you recover the original numbers? This is called "round-tripping," and it's a common operation with algorithms that encode and decode text.

Chapter 5: Howler: Working with files and STDOUT



In Harry Potter, a "Howler" is a nasty-gram that arrives by owl at Hogwarts. It will tear itself open, shout a blistering message at the recipient, and then combust. In this exercise, we're going to write a program that will transform text into a rather mild-mannered version of a Howler by MAKING ALL THE LETTERS UPPERCASE. (I had originally wanted to call this exercise OWEN MEANY but thought more people would be familiar with JK Rowling over John Irving.)

The text that we'll process will be given as a single, positional argument. (Remember spaces on the command line delimit arguments, so multiple words need to be enclosed in quotes to be considered one argument.) For instance, if our program is given the input, "How dare you steal that car!", it should scream:

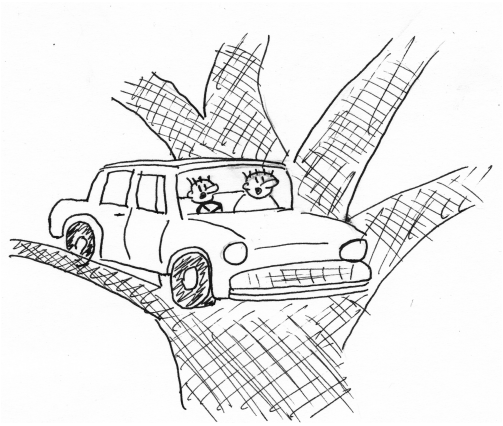


```
$ ./howler.py 'How dare you steal that car!'
HOW DARE YOU STEAL THAT CAR!
```

The argument to the program may also name a file, in which case we need to read the file for the input:

```
$ ./howler.py ../inputs/fox.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Our program will also accept an `-o` or `--outfile` option that names an output file into which the output text should be written. In that case, nothing will be printed on the command line:



```
$ ./howler.py -o out.txt 'How dare you steal that car!'
```

And there should now be a file called `out.txt` that has the output:

```
$ cat out.txt
HOW DARE YOU STEAL THAT CAR!
```

In this exercise, you will learn to:

- Accept text input from the command line or from a file.
- Change strings to uppercase.
- Print output either to the command line or to a file that needs to be created.

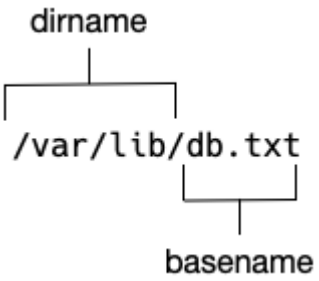
Reading files

This will be our first exercise that will involve reading files. The argument to the program will be some text. That text might name an input file in which case you will open and read the file. If it's not the name of a file, then you'll use the text itself.

The built-in `os` (operating system) module has a method for detecting if a string is the name of a file. To use it, you must import the `os` module. For instance, there's probably not a file called "blargh" on your system:

```
>>> import os
>>> os.path.isfile('blargh')
False
```

The `os` module contains loads of useful submodules and functions. Consult the documentation at https://docs.python.org/3/library/os.html or use `help(os)` in the REPL. For instance, the `os.path` module has functions like `basename` and `dirname` for getting a file's name or directory from a path, for example:




```
>>> path = '/var/lib/db.txt'
>>> os.path.dirname(path)
'/var/lib'
>>> os.path.basename(path)
'db.txt'
```

In the `inputs` directory of the source code repo, there are several files. I'll use a file called `inputs/fox.txt`. Note you will need to be in the main directory of the repo for this to work:

```
>>> file = 'inputs/fox.txt'
>>> os.path.isfile(file)
True
```

Once you've determined that the argument is the name of a file, you must `open` it to `read` it. The return from `open` is a *file handle*, the thing we use to `read` the file. I usually call this variable `fh` to remind me that it's a file handle. If I have more than one open file handle like both input and output handles, I may call them `in_fh` and `out_fh`.

```
>>> fh = open(file)
```

If you try to `open` a file that does not exist, you'll get an exception.

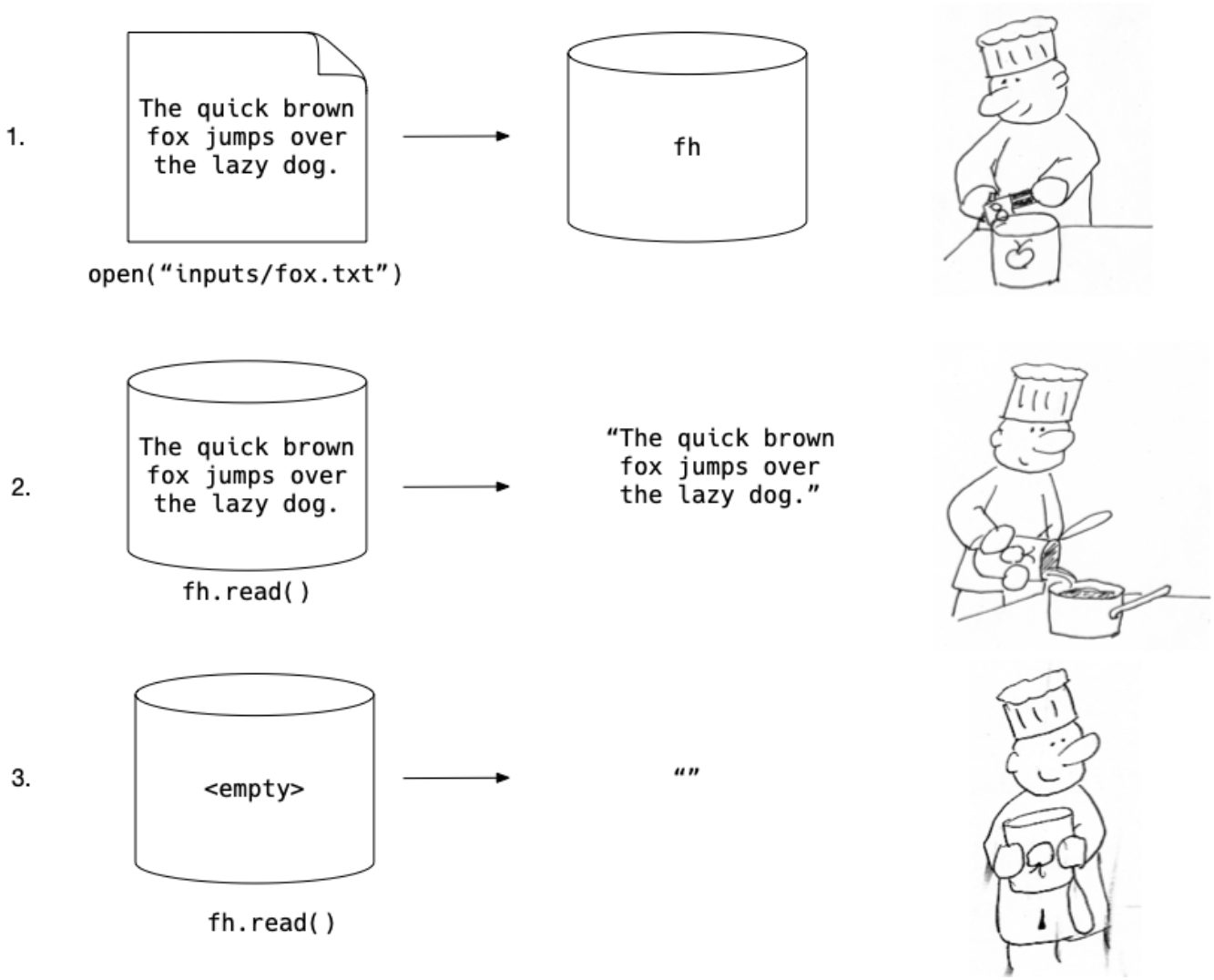
This is unsafe code:

```
>>> file = 'blargh'
>>> open(file)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'blargh'
```

Always check that the file exists!

```
>>> file = 'inputs/fox.txt'
>>> if os.path.isfile(file):
...     fh = open(file)
```

I think of the `file` here ("inputs/fox.txt") as the *name of the file* on disk. It's a bit like a can of tomatoes.



1. The file handle (`fh`) is a mechanism I can use to get at the contents of the file. To get at the tomatoes, we need to `open` the can.
2. The `fh.read` method returns what is inside the `file`. With the can opened, we can get at the contents.
3. Once the file handle has been read, there's nothing left.

Let's see what `type` the `fh` is:

```
>>> type(fh)
<class '_io.TextIOWrapper'>
```

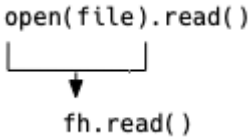
In computer lingo, "io" means "input/output." This is some sort of object that wraps I/O operations around text. You can use `help(fh)` (using the name of the variable itself) to read the docs on the `class TextIOWrapper`. The two methods you'll use quite often are `read` and `write`. Right now, we just care about `read`. Let's see what that gives us:

```
>>> fh.read()
'The quick brown fox jumps over the lazy dog.\n'
```

Do me a favor and execute that line one more time. What do you see?

```
>>> fh.read()
''
```

A file handle is different from something like a `str`. Once you `read` a file handle, it's empty! It's like pouring the tomatoes out of the can. Now the can is empty, and you can't empty it again.



We can actually compress the `open` and `read` into one line of code by *chaining* those methods together. The `open` returns a file handle that can be used for the call to `read`. Run this:

```
>>> open(file).read()
'The quick brown fox jumps over the lazy dog.\n'
```

And now run it again:

```
>>> open(file).read()
'The quick brown fox jumps over the lazy dog.\n'
```

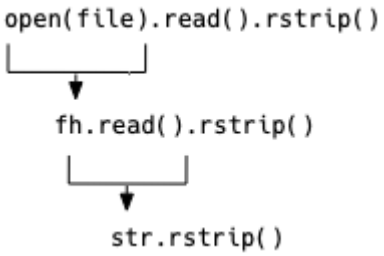
Each time you `open` the `file`, you get a fresh file handle to `read`.

If you want to preserve the contents, you need to copy them into a variable.

```
>>> text = open(file).read()
>>> text
'The quick brown fox jumps over the lazy dog.\n'
```

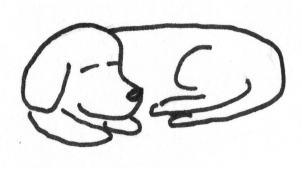
The `type` of the result is a `str`:

```
>>> type(text)
<class 'str'>
```



If I want, I can chain a `str` method onto the end of that. For instance, maybe I want to remove the trailing newline. The `str.rstrip` method will remove any whitespace (which includes newlines) from the *right* end of a string.

```
>>> text = open(file).read().rstrip()
>>> text
'The quick brown fox jumps over the lazy dog.'
```



Once you have your input text, whether it is from the command line or from a file, you need to UPPERCASE it. The `str.upper()` is probably what you want.

At this point, you have read your input, whether it is from the command-line or from a file.

Writing files

The output of the program should either appear on the command line or be written to a file. Command-line output is also called called "standard out" or `STDOUT`. It's the *standard* or normal place for *output* to occur.

There's also an option to the program to write the output to a file, so let's look at how to do that. You still need to `open` a file handle, but we have to use an optional second argument, the string `'w'`, that instructs Python to open it for *writing*. Other modes include `'r'` for *reading* (the default) and `'a'` for _appending.

Table 1. File writing modes

Mode	Meaning
w	write
r	read
a	append

You can additionally describe the kind of content, whether `'t'` for *text* (the default) or `'b'` for _binary:

Table 2. File content modes

Mode	Meaning
t	text
b	bytes

You can combine these two tables like `'rb'` to read a binary file or `'at'` to append to a text file. Here we will use `'wt'` to write a text file. I'll call my variable `out_fh` to remind me that this is the "output file handle":

```
>>> out_fh = open('out.txt', 'wt')
```

If the file does not exist, it will be created. If the file does exist, then it will be *overwritten*. It's possible to `open` in a mode that will append text to an existing file. For this exercise, `'wt'` will suffice.

You can use the `write` method of the file handle to put text into the file. Whereas the `print` method will append a newline (`\n`) unless you instruct it not to, the `write` method will *not* add a newline, so you have to explictly add one.

If you use the `fh.write` method in the REPL, you will see that it return the number of bytes written. Here each character is a byte, and remember that the newline (`\n`) is included:

```
>>> out_fh.write('this is some text\n')
18
```

You can check that this is correct:

```
>>> len('this is some text\n')
18
```

Most code tends to ignore this return value; that is, we don't bother to capture the results into a variable or check that we got a non-zero return. If `write` fails, there's usually some much bigger problem with your system. It's also possible to use the `print` function with the optional `file` argument. Notice that I don't include a newline with `print` because it will add one:

```
>>> print('this is some more text', file=out_fh)
```

When you are done writing to a file handle, you should `close` it so that Python can clean up the file and release the memory associate with it. This function returns no value:

```
>>> out_fh.close()
```

We can verify that our text made it:

```
>>> open('out.txt').read()
'this is some text\nthis is some more text\n'
```

When you `print` on an open file handle, the text will be appended to any previously written data. Look at this code, though:

```
>>> print("I am what I am an' I'm not ashamed", file=open('hagrid.txt', 'wt'))
```

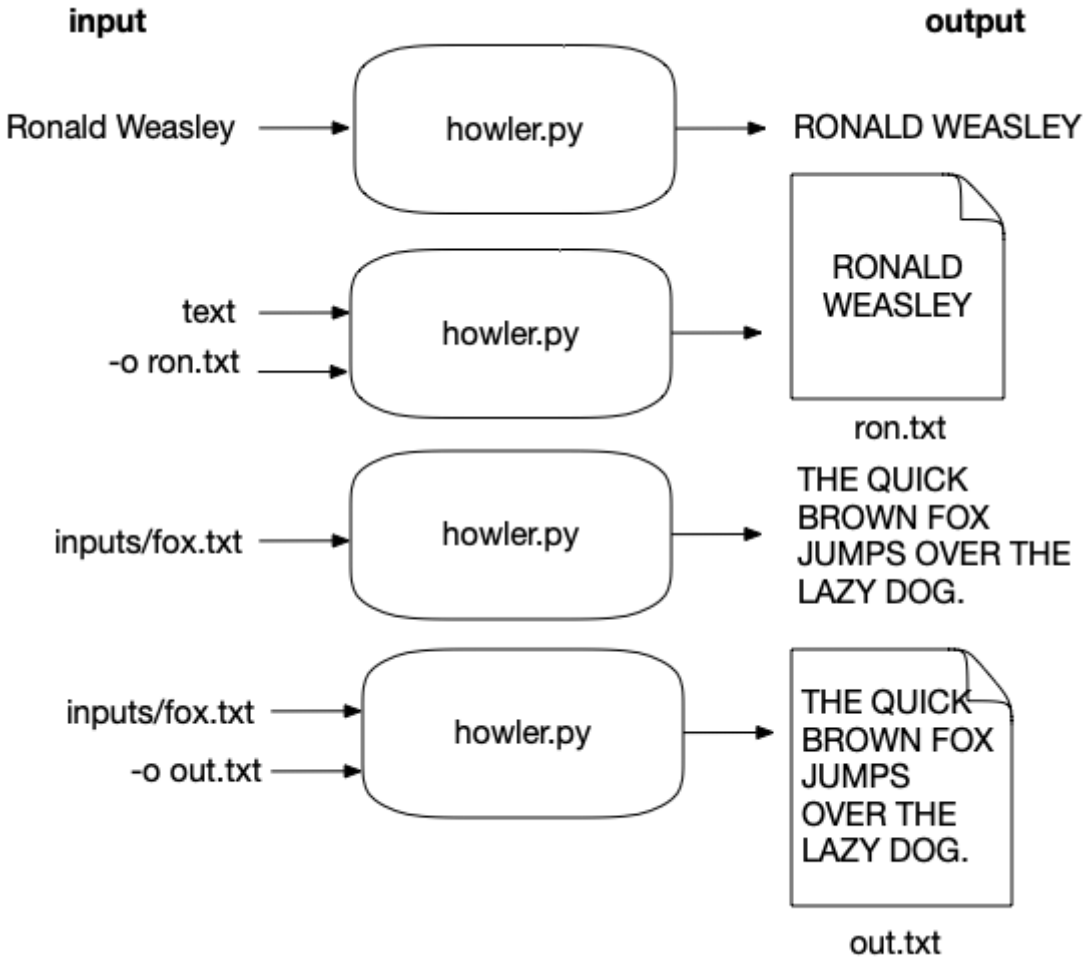
If you run that line twice, will the file called "hagrid.txt" have the line once or twice? Let's find out:

```
>>> open('hagrid.txt').read()
"I am what I am an' I'm not ashamed\n"
```

Just once! Why is that? Remember, each called `open` gives us a new file handle, so calling `open` twice results in new file handles. Both are opened in *write* mode, so that existing data is *overwritten*. It's important to understand how to `open` files properly or you may end up erasing important data files!

Writing `howler.py`

Here is a string diagram showing the overview of the program and some example inputs and outputs:



When run with no arguments, it should print a short usage:

```
$ ./howler.py
usage: howler.py [-h] [-o str] STR
howler.py: error: the following arguments are required: STR
```

When run with `-h` or `--help`, the program should print a longer usage statement:

```
$ ./howler.py -h
usage: howler.py [-h] [-o str] str

Howler (upper-cases input)

positional arguments:
  str                   Input string or file

optional arguments:
  -h, --help            show this help message and exit
  -o str, --outfile str
                        Output filename (default: )
```

If the argument is a regular string, it should uppercase that:

```
$ ./howler.py 'How dare you steal that car!'
HOW DARE YOU STEAL THAT CAR!
```

If the argument is the name of a file, it should uppercase the *contents of the file*:

```
$ ./howler.py ../inputs/fox.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

If given an `--outfile` filename, the uppercased text should be written to the indicated file and nothing should be printed to STDOUT:

```
$ ./howler.py -o out.txt ../inputs/fox.txt
$ cat out.txt
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

Hints:

- Start with `new.py` and alter the `get_args` section until your usage statements match the ones above.
- Run the test suite and try to pass just the first test that handles text on the command line and output to `STDOUT`.
- The next test is to see if you can write the output to a given file. Figure out how to do that.
- The next test is for reading input from a file. Then work on that. Don't try to do all the things at once!
- There is a special file handle that always exists called "standard out" (often `STDOUT`). If you `print` without a `file` argument, then it defaults to `sys.stdout`. You will need to `import sys` in order to use it.

Be sure you really try to write the program and pass all the tests before moving on to read the

solution! Fear is the mind-killer. You can do this.

Solution

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

import argparse
import os
import sys


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

    parser = argparse.ArgumentParser(
        description='Howler (upper-case input)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input string or file') ①

    parser.add_argument('-o', ②
                        '--outfile',
                        help='Output filename',
                        metavar='str',
                        type=str,
                        default='')

    args = parser.parse_args()

    if os.path.isfile(args.text): ③
        args.text = open(args.text).read().rstrip() ④

    return args


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

    args = get_args()
    text = args.text ⑤
    out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout ⑥
    out_fh.write(text.upper()) ⑦
    out_fh.close() ⑧


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

① The `text` argument is a string that may be the name of a file.

② The `--outfile` option is also a string that names a file.

③ Check if `args.text` names an existing file.

④ If so, overwrite the value of `args.text` with the results of reading the file.

⑤ At this point, the `text` input has been handled, whether from the command line or a file.

⑥ An `if` expression to choose either `sys.stdout` or a newly opened file handle for the output.

⑦ Use the opened file handle to write the output converted to `upper`.

⑧ Close the file handle.

Discussion

Defining the arguments

The `get_args` function, as always, is the first. Here I define two arguments. The first is a positional `text` argument. Since it may or may not name a file, all I can know is that it will be a string.

```
parser.add_argument('text', metavar='str', help='Input string or file')
```

The other argument is an option, so I give it a short name of `-o` and a long name of `--outfile`. The default `type` for all arguments is `str`, but I go ahead and state that explictly. The `default` value is the empty string. I could just as easily use the special `None` type which is also the default value, but I prefer to use a defined argument like the empty string.

```
parser.add_argument('-o',
                    '--outfile',
                    help='Output filename',
                    metavar='str',
                    type=str,
                    default='')
```

Reading input from a file or the command line

This is a deceptively simple program that demonstrates a couple of very important elements of file input and output. The `text` input might be a plain string or it might be the name of a file. This pattern will come up repeatedly in this book:

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```

The function `os.path.isfile` will tell us if there is a file with the name in `text`. If that returns `True`, then we can safely `open(file)` to get a *file handle* which has a method called `read` which will return *all* the contents of the file. This is usually safe, but be careful if you write a program that could potentially read gigantic files. For instance, in my day job we regularly deal with files with sizes in the 10s to 100s of gigabytes, so I would need to ensure my system has more memory than the size of the file!

The result of `open(file).read()` is a `str` which itself has a method called `rstrip` that will return a copy of the string *stripped* of the whitespace off the *right* side of the string (hence the name `rstrip`). The longer way to write the above would be:

```
if os.path.isfile(text):
    fh = open(text)
    text = fh.read()
    text = text.rstrip()
```

In my version, I choose to handle this inside the `get_args` function. This is the first time I'm showing you that you can intercept and alter the arguments before passing them on to `main`. I'll use this idea quite a bit, like in the "99 Bottle of Beer," we'll need to ensure that the starting "bottle" is between 0 and 99. I like to do all the work to validate the user's arguments inside `get_args`. I could just as easily do this in `main` after the call to `get_args`, so this is entirely a style issue.

Choosing the output file handle

The line decides where to put the output of our program

```
out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout
```

The `if` expression will open `args.outfile` for writing text (`wt`) if the user provided that argument; otherwise, we will use `sys.stdout` which is a file handle to STDOUT. Note that we don't have to call `open` on `sys.stdout` because it is always there and always open for business.

Printing the output

To get uppercase, we can use the `text.upper` method, then we need to find a way to print it to the output file handle. I chose to do:

```
out_fh.write(text.upper())
```

But you could also do:

```
print(text.upper(), file=out_fh)
```

Finally I need to close the file handle with `out_fh.close()`.

Review

- To `read` or `write` to files, you must `open` them.
- The default mode for `open` is for reading a file.
- To write a text file, you must use `'wt'` as the second argument to `open`.
- By default, you `write` text to a file handle. You must use the `'b'` flag to indicate that you want to write binary data.
- The `os.path` module contains many useful functions such as `isfile` that will tell you if a file exists by the given name.
- `STDOUT` (standard output) is always available via the special `sys.stdout` file handle which is always open.
- The `print` function takes an optional `file` argument of where to put the output. That argument must be an open file handle such as `sys.stdout` (the default) or the result of `open`.

Going Further

- Add a flag that will lowercase the input instead.
- Alter the program handle multiple input files
- Change `--outfile` to `--outdir` and write each input file to the same file name in the output directory.

Chapter 6: Words Count (Reading files/STDIN, iterating lists, formatting strings)

> "I love to count!" — Count von Count

Counting things is a surprisingly important programming skill. Maybe you're trying to find how many pizzas were sold each quarter or how many times you see certain words in a set of documents. Usually the data we deal with in computing comes to us in files, so we're going to push a little further into reading files and manipulating strings by writing a Python version of the venerable Unix `wc` (word count) program. We're going to write a program called `wc.py` that will count the characters, words, and lines for each input argument which should be files.

Given one or more valid files, it should print the number of lines, words, and characters, each in columns 8 characters wide, followed by a space and then the name of the file. Here's what it looks like for one file:

```
$ ./wc.py ../inputs/scarlet.txt
    7035   68061  396320 ../inputs/scarlet.txt
```

When there are many files, print the counts for each file and then print a "total" line summing each column:

```
$ ./wc.py ../inputs/s*.txt
    7035   68061  396320 ../inputs/scarlet.txt
      17     118     661 ../inputs/sonnet-29.txt
       3       7      45 ../inputs/spiders.txt
    7055   68186  397026 total
```

There may also be *no* arguments, in which case we'll read from "standard in" (`STDIN`) which is the complement to `STDOUT` we used in "Howler." You can use `STDIN` to chain programs together where the output of one program becomes the input for the next. To pass text via `STDIN`, you can use the `<` redirect from a file:

```
$ ./wc.py < ../inputs/fox.txt
       1       9      45 <stdin>
```

Or pipe (`|`) the output of one command into another:

```
$ cat ../inputs/fox.txt | ./wc.py
       1       9      45 <stdin>
```

Given a non-existent file, it should print an error message and exit with a non-zero exit value:

```
$ ./wc.py foo
usage: wc.py [-h] FILE [FILE ...]
wc.py: error: argument FILE: can't open 'foo': \
[Errno 2] No such file or directory: 'foo'
```

So our program is going to have to:

- Take 0 or more positional arguments.
- Validate that any given arguments are actually files.
- Read each file and count the lines, words, and characters.
- If no files are present, read from `STDIN`.
- Print the total number of lines, words, and characters if there is more than one file.

Defining file inputs

The first step will be to define your arguments to `argparse`. The program takes *zero or more* positional arguments and nothing else. (Remember that you never have to define the `-h` or `--help` arguments as `argparse` handles those automatically.)

In "Picnic," we used `nargs='+'` to indicate one or more items for our picnic. Here we want to use `nargs='\*'` to indicate *zero* or more. If there are no arguments, we'll read `STDIN`. Note that both `'+'` and `'*'` with `nargs` means that `argparse` will provide the values as a `list`. Even if there are no arguments, you will still get an empty `list` (`[]`).

In "Howler," we used the "standard out" (`STDOUT`) file handle with `sys.stdout`. To read `STDIN`, we'll use Python's `sys.stdin` file handle which is similarly always open and available to you. Because you are using `nargs='*'`, the values will be a list, and so the `default` should be a `list` as well. Can you figure out how to make the `default` value for your file argument be a `list` with `sys.stdin`?

Lastly, we should discuss the `type` of the positional arguments. If they are provided, they should be *readable files*. We saw in "Howler" how to test if the input argument was a file by using `os.path.isfile`. In that program, the input might be either plain text or a file name, so we had to check this ourselves. In this program, however, we will require that any arguments should be files, and so we can define the `type=argparse.FileType('r')`. When you do this, `argparse` takes on all the work to validate the inputs from the user and produce useful error messages. If the user provides valid input, then `argparse` will provide you with a `list` of *open file handles*. All in all, this saves you quite a bit of time.

Iterating lists

Your program will end up with a `list` of file handles. In "Jump The Five," we used a `for` loop to iterate through the characters in the input text. Here we can use a `for` loop over the `file` inputs.

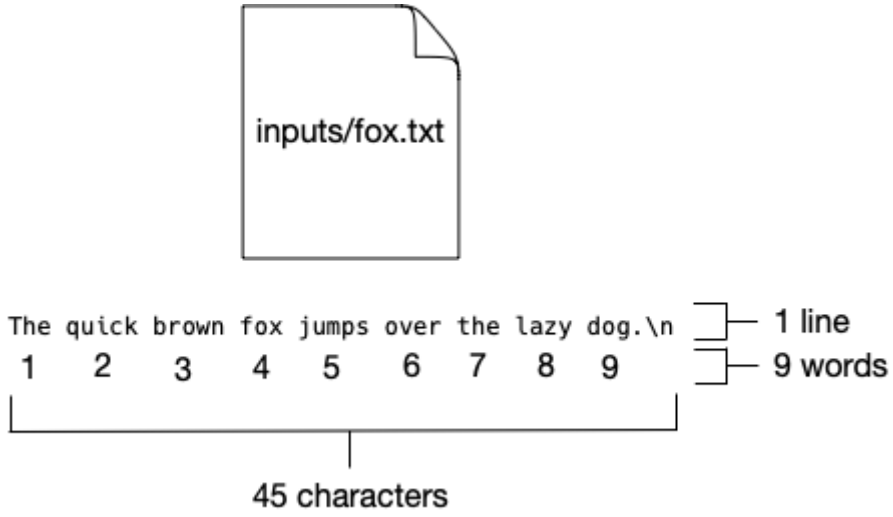
```
for fh in args.file:
    # read each file
```

The `fh` is a "file handle." We saw in "Howler" how to manually `open` and `read` a file. Here the `fh` is already open, so we can just `read` it. There are many ways to read a file, however. The `read` method will give you the *entire contents* of the file in one go. If the file is large — say, if the size of the file exceeds your available memory on your machine — then your program will crash. I would recommend, instead, that you use another `for` loop on the `fh`. Python will understand this to mean that you wish to read each `line` of input, one-at-a-time.

```
for fh in args.file:
    for line in fh:
        # process the line
```

So that's two levels of `for` loops, one for each file handle and then another for each line in each file handle.

What you're counting



The output for each file will be the number of lines, words, and characters, each printed in a field 8 characters wide followed by the name of the file which will be available to you via `fh.name`. Let's take a look at the output from the standard `wc` program on my system. Notice that when run with just one argument, it produces counts only for that file:

```
$ wc fox.txt
       1       9      45 fox.txt
```

When run with multiple files, it also shows a "total" line:

```
$ wc fox.txt sonnet-29.txt
       1       9      45 fox.txt
      17     118     669 sonnet-29.txt
      18     127     714 total
```

For each file, you will need to create variables that hold the numbers for lines, words, and characters. For instance, if you use the `for line in fh` loop that I suggest, then you need to have a variable like `num_lines` to increment on each iteration. That is, somewhere in your code you will need to set a variable to `0` and then, inside the `for` loop, make it go up by one. The idiom in Python for this is:

```
num_lines = 0
for line in fh:
    num_lines += 1
```

You also need to count the number of words and characters, so you'll need similar `num_words` and `num_chars` variables. In "Picnic," we discussed how we can convert back and forth between strings and lists. For the purposes of this exercise, we'll use the `str.split` method to break each `line` on spaces. You can then use the length of the resulting `list` as the number of words. For the number of characters, you can use the same length function on the `line` and add that to a `num_chars` variable.

Formatting your results

This is the first exercise where the output needs to be formatted in a particular way. Don't try handle this part manually. That way lies madness. Instead, you need to learn the magic of the `str.format` method. The `help` doesn't have much in the way of documentation, so I'd recommend you read PEP3101 (https://www.python.org/dev/peps/pep-3101/).

We've seen that the curlies (`{}`) inside the `str` part create placeholders that will be replaced by the values passed to the method:

```
>>> import math
>>> 'Pi is {}'.format(math.pi)
'Pi is 3.141592653589793'
```

You can put formatting information inside the curlies to specify how you want the value displayed. If you are familiar with `printf` from C-type languages, this is the same idea. For instance, I can print just two numbers of `pi` after the decimal. The `:` introduces the formatting options, and the `0.02f` describes two decimal points of precision:

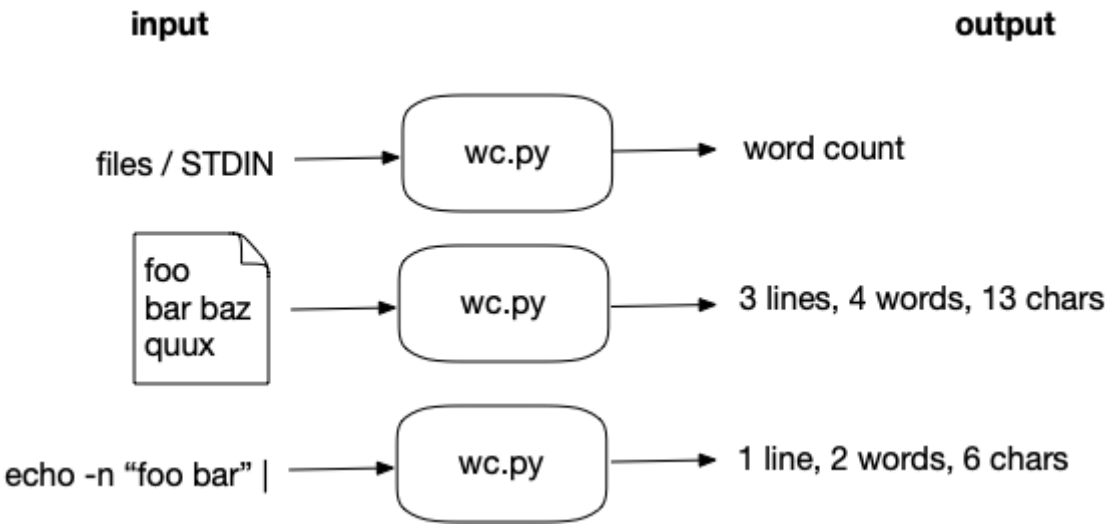
```
>>> 'Pi is {:0.02f}'.format(math.pi)
'Pi is 3.14'
```

The formatting information comes after the colon (`:`) inside the curlies. You can also use the f-string method where the variable comes *before* the colon:

```
>>> f'Pi is {math.pi:0.02f}'
'Pi is 3.14'
```

Here you need to use `{:8}` for each of lines, words, and characters so that they all line up in neat columns. The `8` describes the width of the field which is assumed to be a string. The text will be right-justified.

Writing `wc`



Now it's time to write out Python program called `wc.py` that will emulate the `wc` program in Unix that counts the lines, words, and characters in the given file arguments. If run with the `-h` or `--help` flags, the program should print usage:

```
$ ./wc.py -h
usage: wc.py [-h] [FILE [FILE ...]]

Argparse Python script

positional arguments:
  FILE        Input file(s) (default: [<_io.TextIOWrapper name='<stdin>'
              mode='r' encoding='UTF-8'>])

optional arguments:
  -h, --help  show this help message and exit
```

Hints:

- Start with `new.py` and delete all the non-positional arguments.
- Use `nargs='*'` to indicate zero or more positional arguments for your `file` argument.
- How could you use `sys.stdin` for the `default`? Remember that both `narg='*'` and `nargs='+'` mean that the arguments will be supplied as a `list`. How can you create a `list` that contains just `sys.stdin` for the `default` value?
- Remember that you are just trying to pass one test at a time. Create the program, get the help right, then worry about the first test.
- Compare the results of your version to the `wc` installed on your system. Note that not every Unix-like system has the same `wc`, so results may vary.

Solution

```
#!/usr/bin/env python3
"""Emulate wc (word count)"""

import argparse
import sys


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

    parser = argparse.ArgumentParser(
        description='Emulate wc (word count)',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file',
                        metavar='FILE',
                        nargs='*',
                        default=[sys.stdin], ①
                        type=argparse.FileType('r'), ②
                        help='Input file(s)')

    return parser.parse_args()


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

    args = get_args()

    total_lines, total_chars, total_words = 0, 0, 0 ③
    for fh in args.file:                ④
        lines, words, chars = 0, 0, 0   ⑤
        for line in fh:                 ⑥
            lines += 1                  ⑦
            chars += len(line)          ⑧
            words += len(line.split()) ⑨

        total_lines += lines ⑩
        total_chars += chars
        total_words += words

        print(f'{lines:8}{words:8}{chars:8} {fh.name}') ⑪

    if len(args.file) > 1: ⑫
        print(f'{total_lines:8}{total_words:8}{total_chars:8} total') ⑬
```

```
48
49
50  # --------------------------------------------------
51  if __name__ == '__main__':
52      main()
```

① If you set the `defaut` to a `list` with `sys.stdin`, then you have easily handled the `STDIN` option.

② This will make `argparse` handle all the validation of file inputs.

③ These are the variables for the "total" line, if we need them.

④ Iterate through the `list` of `arg.file` inputs. I use the variable `fh` to remind me that these are open file hanndles, even `STDIN`.

⑤ Initialize variables to count *just this file*.

⑥ Iterate through each `line` of `fh`.

⑦ For each line, we increment `lines` by 1.

⑧ The number of `chars` is incremented by the length of the `line`.

⑨ To get the number of words, we can `split` the `line` on spaces (the default). We length of that `list` is added to the `words`.

⑩ We add the numbers for this file to the `total_` variables.

⑪ Print the counts for this file using the `{:8}` option to print in a field 8 characters wide.

⑫ Check if we had more than 1 input.

⑬ Print the "total" line.

Discussion

Defining the arguments

This program is rather short and seems rather simple, but it's definitely not exactly easy. One part of the exercise is to really get familiar with `argparse` and the trouble it can save you. The key is in defining the `file` positional arguments. If you use `nargs='*'` to indicate zero or more arguments, then you know `argparse` is going to give you back a `list` with zero or more elements. If you use `type=argparse.FileType('r')`, then any arguments provided must be readable files. The `list` that `argparse` returns will be a `list` of *open file handles*. Lastly, if you use `default=[sys.stdin]`, then you understand that `sys.stdin` is essentially an open file handle to read from "standard in" (AKA `STDIN`), and you are letting `argparse` know that you want the default to be a `list` containing `sys.stdin`.

Reading a file using a `for` loop

I can create a list of open file handles in the REPL to mimic what I'd get from `args.file`:

```
>>> files = [open('../inputs/fox.txt')]
```

Before I use a `for` loop to iterate through them, I need to set up three variables to track the *total* number of lines, words, and characters:

```
>>> total_lines, total_chars, total_words = 0, 0, 0
```

Inside the `for` loop for each file handle, I initialize three more variables to hold the count of lines, characters, and words *for this particular file*. I then use another `for` loop to iterate over each line in the file handle (`fh`). For the `lines`, I can add `1` on each pass through the `for` loop. For the `chars`, I can add length of the line (`len(line)`) to track the number of characters. Lastly for the `words`, I can use `line.split()` to break the line on whitespace to create a `list` of "words." It's not actually a perfect way to count actual words, but it's close enough. I can use the `len` function on the `list` to add to the `words` variable. The `for` loop ends when the end of the file is reached, and that is when I can `print` out the counts and the file name using `{:8}` placeholders in the print template to indicate a text field 8 characters wide.

```
>>> for fh in files:
...     lines, words, chars = 0, 0, 0
...     for line in fh:
...         lines += 1
...         chars += len(line)
...         words += len(line.split())
...     print(f'{lines:8}{words:8}{chars:8} {fh.name}')
...     total_lines += lines
...     total_chars += chars
...     total_words += words
...
       1       9      45 ../inputs/fox.txt
```

Notice that the `print` statement lines up with the inner `for` loop so that it will run after we're done iterating over the lines in `fh`. I chose to use the f-string method to print each of `lines`, `words`, and `chars` in a space 8 characters wide. After printing, I can add the counts to my "total" variables to keep a running total.

Lastly, if the number of file arguments is greater than 1, I need to print my totals:

```
if len(args.file) > 1:
    print(f'{total_lines:8}{total_words:8}{total_chars:8} total')
```

Review

- The `nargs` (number of arguments) option to `argparse` allows you to validate the number of arguments from the user. The star (`'*'`) means zero or more while `'+'` means one or more.
- If you define an argument using `type=argparse.FileType('r')`, then `argparse` will validate that the user has provided a readable file and will make the value available in your code as an open file handle.
- You can read and write from the Unix standard in/out file handles by using `sys.stdin` and `sys.stdout`.
- You can nest `for` loops to handle multiple levels of processing.
- The `str.split` method will split a string on spaces into words.
- The `len` function can be used on both strings and lists. For the latter, it will tell you the number of elements contained.
- The `str.format` and Python's f-strings both recognize the same printf-style formatting options to allow you to control how a value is displayed.

Going Further

- Add the same flags as your system `wc` and change your program to create the same output; i.e., print all the columns for no arguments but only print the "lines" when `-l` is present.
- Implement other system tools like `head`, `tail`, `cat`, and `tac` (the reverse of `cat`).

Chapter 7: Gashlycrumb: Looking items up in a dictionary

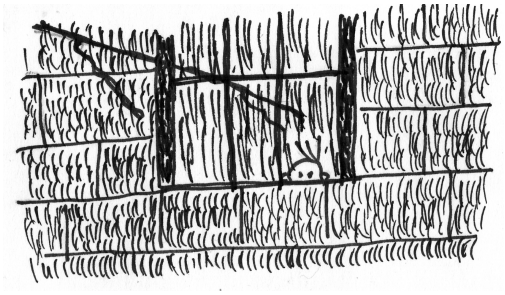


Figure 4. N is for Neville who died of ennui.

Every time you log into a website, the code behind it has to look up your username and password to compare to the values you put into the login form. Whenever you give your phone number at the hardware store or scan your library card to checkout a book, a computer program uses one piece of information to find other things like how often you buy compost or if you have any overdue books. Probably all these examples would be using a database to find that information. We're going to use a dictionary that we will fill with information from an input file.

In this exercise, we're going to look up a line of text from an input file that starts with the letter provided by the user. The text will come from Edward Gorey's "The Gashlycrumb Tinies," an abecedarian book that describes various and ghastly ways in which children die. Instead of "A is for apple, B is for bear," we get:

> A is for Amy who fell down the stairs. B is for Basil assaulted by bears.

Our `gashlycrumb.py` program will take a letter of the alphabet as a positional argument and will look up the line of text from an *optional* input file that starts with that letter. When our unfortunate user runs our program, here is what they would see. Notice that we will consider the letter in a case-insensitive fashion:

```
$ ./gashlycrumb.py a
A is for Amy who fell down the stairs.
```

So that means our program will need to:

- Accept a single positional argument we'll call `letter`
- Accept an optional `--file` argument which must be a file. The default value will be `'gashlycrumb.txt'` (provided).
- Read the file, find the first letter of each line, and build a data structure that associates that letter to the line of text.
- Check if the `letter` is a key in the dictionary.
- Print the line of text for the letter if it is.

- Print an error if it isn’t.

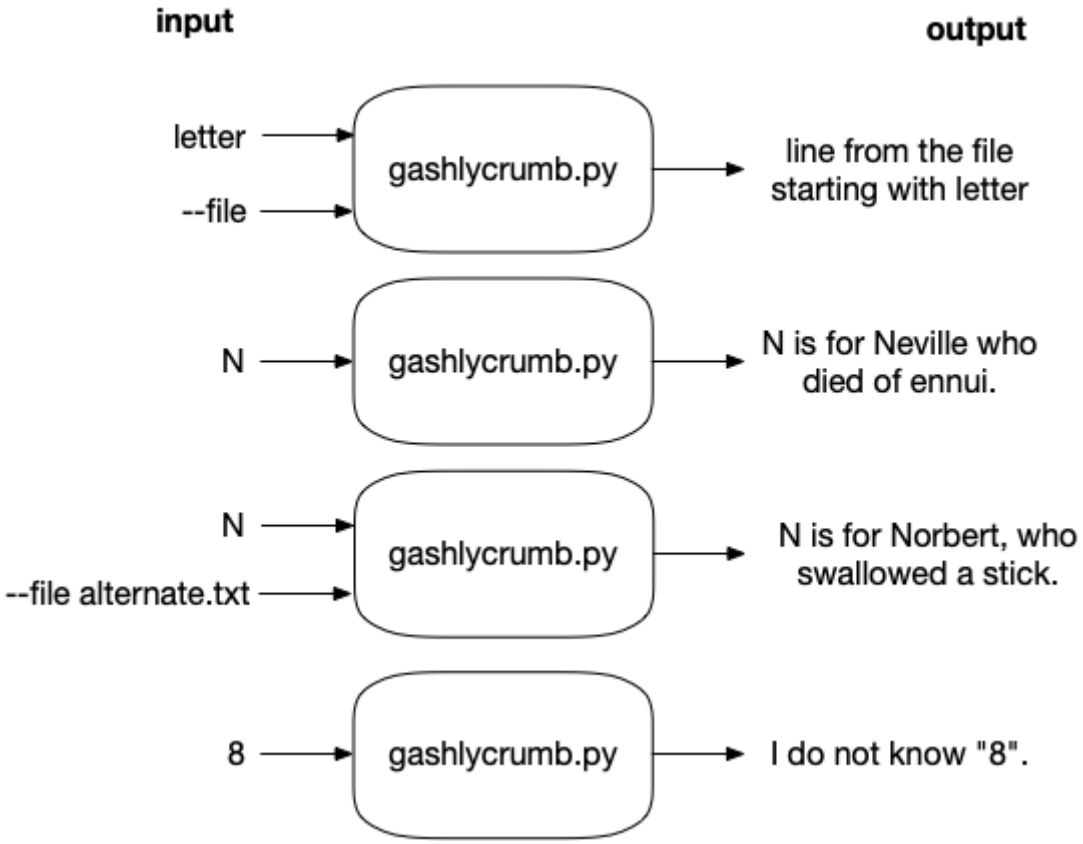
The input file will have each letter’s text on a separate line. From the "Word Count" program, you know how to take a file input and read it line-by-line. From the "Article" program, you know how to get the first letter of a bit of text. From the "Jump The Five" program, you know how to build a dictionary and lookup a value. Now you’ll put all those skills together to recite morbid poetry!

Writing `gashlycrumb.py`

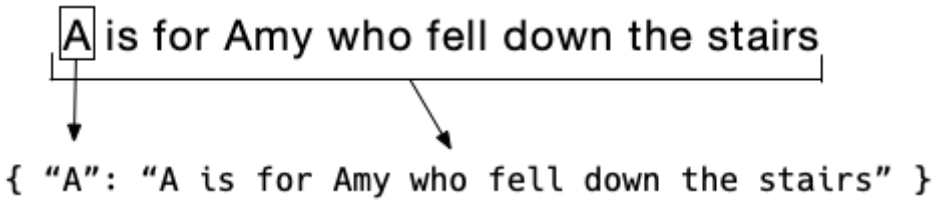
Lets' start by getting our arguments straight. Here is the help your program should generate usage with *no arguments* or for `-h|--help`:

```
$ ./gashlycrumb.py -h
usage: gashlycrumb.py [-h] [-f str] str

Gashlycrumb

positional arguments:
  str                 Letter

optional arguments:
  -h, --help          show this help message and exit
  -f str, --file str  Input file (default: gashlycrumb.txt)
```

The "letter" is a required positional argument, but the `-f|--file` argument is an option. Here is a string diagram showing how the program will work:



You can see the structure of the `gashlycrumb.txt` file:




```
$ head -3 gashlycrumb.txt
A is for Amy who fell down the stairs.
B is for Basil assaulted by bears.
C is for Clara who wasted away.
```

You will use the first character of the line as a lookup value:

```
$ ./gashlycrumb.py a
A is for Amy who fell down the stairs.
$ ./gashlycrumb.py z
Z is for Zillah who drank too much gin.
```

If given a value that does not exist in the list of first characters on the lines from the input file (when searched without regard to case), you should print a message:

```
$ ./gashlycrumb.py 3
I do not know "3".
$ ./gashlycrumb.py CH
I do not know "CH".
```

If provided a `--file` argument that does not exist, your program should exit with an error and message:

```
$ ./gashlycrumb.py -f sdfl b
usage: gashlycrumb.py [-h] [-f str] str
gashlycrumb.py: error: argument -f/--file: can't open 'sdfl': \
[Errno 2] No such file or directory: 'sdfl'
```

Hints:

- Start with `new.py` and remove everything but the positional and optional `argparse.FileType('r')` parameters.
- A dictionary is a natural data structure that you can use to associate some value like the letter "A" to some phrase like "A is for Amy who fell down the stairs." Create a new, empty `dict`.
- Once you have an open file handle, you can `read` the file line-by-line with a `for` loop.

- Each line of text is a string. How can you get the first character of a string?
- Using that first character, how can you set the value of a `dict` to be the key and the line itself to be the value?
- Once you have constructed the dictionary of letters to lines, how can you check that the user's `letter` argument is `in` the dictionary?
- Can you solve this without a `dict`?

Solution

```
#!/usr/bin/env python3
"""Lookup tables"""

import argparse


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

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

    parser.add_argument('letter', help='Letter', metavar='str', type=str) ①

    parser.add_argument('-f', ②
                        '--file',
                        help='Input file',
                        metavar='str',
                        type=argparse.FileType('r'),
                        default='gashlycrumb.txt')

    return parser.parse_args()


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

    args = get_args()
    letter = args.letter
    lookup = {line[0].upper(): line.rstrip() for line in args.file} ③

    if letter.upper() in lookup:      ④
        print(lookup[letter.upper()]) ⑤
    else:
        print(f'I do not know "{letter}".') ⑥


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

① The required argument (a letter).

② An optional `--file` that must be a readable file, if provided. Defaults to a known value.

③ Build the `lookup` using a dictionary comprehension that reads the given file. Use the `upper` function to disregard case.

④ See if the `letter` argument is in the `lookup` dictionary, checking the `upper` value to disregard case.

⑤ If so, `print` the line of text from the `lookup` for the `letter`.

⑥ Otherwise, print a message that the `letter` is unknown.

Discussion

Handling the arguments

I prefer to have all the logic for parsing and validating the command-line arguments in the `get_args` function. In particular, `argparse` can do a fine job verifying tedious things such as an argument being an existing, readable `--file` which is why I use `type=argparse.FileType('r')` for that argument. If the user doesn't supply a valid argument, then `argparse` will throw an error, printing a helpful message along with the short usage and exiting with an error code.

By the time I get to the line `args = get_args()`, I know that I have a valid, open file handle in the `args.file` slot. In the REPL, I can manually do what `argparse` has done by using `open` to get a file handle which I like to usually call `fh`:

```
>>> fh = open('gashlycrumb.txt')
```

Reading the input file

I know that I want to use a dictionary where the keys are the first letters of each line and the values are the lines themselves. That means I need to start by creating a new, empty dictionary either by using the `dict()` function or by setting a variable equal to an empty set of curly braces (`{}`). I'll call my variable `lookup`:

```
>>> lookup = {}
```

I will use a `for` loop to read each `line` of text. From Crow's Nest, I know I can use `line[0].upper()` to get the first letter of `line` and uppercase it. I can use that as the key into `lookup`. Each `line` of text ends with a newline that I'd like to remove. I can use the `str.rstrip` method to strip whitespace from the right side of the `line` (`rstrip` = *right strip*). The result of that will be the value for my `lookup`:

```
>>> for line in fh:
...     lookup[line[0].upper()] = line.rstrip()
```

I'd like to look at the resulting `lookup` dictionary. I can `print` it from the program or type `lookup` in the REPL, but it's going to be hard to read. I encourage you to try it. Luckily there is a lovely module called `pprint` to "pretty print" data structures. Here is how I can import the `pprint.pprint` function as the shortcut `pp`:

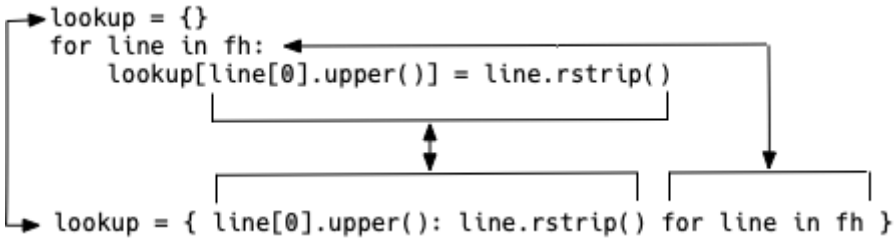
```
>>> from pprint import pprint as pp
```

Now let's take a peek at `lookup`:

```
>>> pp(lookup)
{'A': 'A is for Amy who fell down the stairs.',
 'B': 'B is for Basil assaulted by bears.',
 'C': 'C is for Clara who wasted away.',
 'D': 'D is for Desmond thrown out of a sleigh.',
 'E': 'E is for Ernest who choked on a peach.',
 'F': 'F is for Fanny sucked dry by a leech.',
 'G': 'G is for George smothered under a rug.',
 'H': 'H is for Hector done in by a thug.',
 'I': 'I is for Ida who drowned in a lake.',
 'J': 'J is for James who took lye by mistake.',
 'K': 'K is for Kate who was struck with an axe.',
 'L': 'L is for Leo who choked on some tacks.',
 'M': 'M is for Maud who was swept out to sea.',
 'N': 'N is for Neville who died of ennui.',
 'O': 'O is for Olive run through with an awl.',
 'P': 'P is for Prue trampled flat in a brawl.',
 'Q': 'Q is for Quentin who sank on a mire.',
 'R': 'R is for Rhoda consumed by a fire.',
 'S': 'S is for Susan who perished of fits.',
 'T': 'T is for Titus who flew into bits.',
 'U': 'U is for Una who slipped down a drain.',
 'V': 'V is for Victor squashed under a train.',
 'W': 'W is for Winnie embedded in ice.',
 'X': 'X is for Xerxes devoured by mice.',
 'Y': 'Y is for Yorick whose head was bashed in.',
 'Z': 'Z is for Zillah who drank too much gin.'}
```

Hey, that looks like a handy data structure. Hooray for us!

Looping with `for` versus a list comprehensions



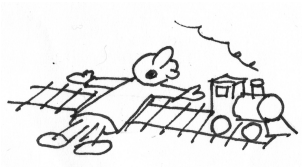
We used three lines of code to build our `lookup` dictionary. We can actually accomplish that in *one* line by using a "dictionary comprehension." We've seen list comprehensions by essentially sticking a `for` inside brackets `[]`. A dictionary comprehension is the same but the `for` loop is inside curlies `{}`.

If you are following along by pasting code into the REPL, note that we have exhausted the file handle `fh` just above by reading it. (Refer back to the Howler to read about file handles.) I need to `open` it again for this next bit:

```
>>> fh = open('gashlycrumb.txt')
>>> lookup = { line[0].upper(): line.rstrip() for line in fh }
```

If you `pprint` it again, you should see the same output as above. It may seem like showing off to write one line of code instead of three, but it really does make a good deal of sense to write compact, idiomatic code. More code always means more chances for bugs, so I usually try to write code that is as simple as possible (but no simpler).

Dictionary lookups



Now that I have a `lookup`, I can ask if some value is `in` the keys. Note that I know the letters are in uppercase and I assume the user could give me lower, so I use `letter.upper()` to only compare that case:

```
>>> letter = 'a'
>>> letter.upper() in lookup
True
>>> lookup[letter.upper()]
'A is for Amy who fell down the stairs.'
```

If the letter is found, I can print the line of text for that letter; otherwise, I can print the message that I don't know that letter:

```
>>> letter = '4'
>>> if letter.upper() in lookup:
...     print(lookup[letter.upper()])
... else:
...     print('I do not know "{}".'.format(letter))
...
I do not know "4".
```

`dict` vs `list` of `tuple`

I don't have to use a `dict` to solve this problem. I could, for example, use a `list` of `tuple` values. We haven't talked about the `tuple` yet. It's an immutable `list`, and you can create a tuple by separating items with commas:

```
>>> x = 'foo', 'bar'
>>> type(x)
<class 'tuple'>
```

It's common to put parentheses around the values if only to make your intent clearer. For instance, this is valid syntax for a tuple with only one value:

```
>>> x = 'foo',
```

To me, that looks like an incomplete statement. I think this is easier to understand:

```
>>> x = ('foo',)
```

Note that the trailing comma is required or you end up with a single value:

```
>>> x = ('foo')
>>> type(x)
<class 'str'>
```

Let's compare and contrast a `tuple` and a `list`:

```
>>> x = ('foo', 'bar')
>>> y = ['foo', 'bar']
```

We can use the `len` function to find their lengths:

```
>>> len(x)
2
>>> len(y)
2
```

And we can use index values to get specific elements:

```
>>> x[0]
'foo'
>>> y[1]
'bar'
```

We can alter the `list` called `y` like maybe adding a new value:

```
>>> y.append('baz')
>>> y
['foo', 'bar', 'baz']
```

But we cannot do this to the `tuple` called `x`:

```
>>> x.append('baz')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

We can also change any element inside a `list`:

```
>>> y[0] = 'quux'
>>> y
['quux', 'bar', 'baz']
```

But, again, a `tuple` is immutable:

```
>>> x[0] = 'quux'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

The `dict.items` method will return a `list` of `tuple` values. Our `lookup` is rather large, so here's a way to look at the first three values of that using `pp` (pretty print) function. Note the outermost square brackets `[]` show that this is a `list`. Each element of the list is enclosed in parentheses to indicate they are `tuple` values:

```
>>> pp(list(lookup.items())[:3])
[('A', 'A is for Amy who fell down the stairs.'),
 ('B', 'B is for Basil assaulted by bears.'),
 ('C', 'C is for Clara who wasted away.')]
```

So a `dict` can be thought of as a list of tuples which I could make directly using a list comprehension:

```
>>> fh = open('gashlycrumb.txt')
>>> lookup = [ (line[0].upper(), line.rstrip()) for line in fh ]
```

I can unpack `lookup` into `char` and `line` to extract, say, just the first 10 `char` elements:

```
>>> [char for char, line in lookup][:10]
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

And then use `in` to see if my `letter` is present:

```
>>> letter = 'a'
>>> letter.upper() in [char for char, line in lookup]
True
```

And get the value like so:

```
>>> [line for char, line in lookup if char == letter.upper()]
['A is for Amy who fell down the stairs.']
```

The problem is that the cost of the search is proportional to the number of values. That is, if we were searching a million keys in a list, then Python starts searching at the beginning of the list and goes until it finds the value. When you store items in a `dict`, the search time for a key can be much shorter, often nearly instantaneous. The method `dict.items()` will return a `list` of `tuple` values containing `(key, value)`, so the line between these two is already blurry.

Review

- A dictionary comprehension is a way to build a dictionary in a one-line `for` loop.
- Defining file input arguments using `argparse.FileType` saves you time and code.
- Looking up a key in a `dict` is faster than searching for an element in a `list`. The time to search a `list` is proportional to its length while dictionary keys are stored in a way optimized for finding them.
- Python's `pprint` module is used for "pretty printing" complex data structures.

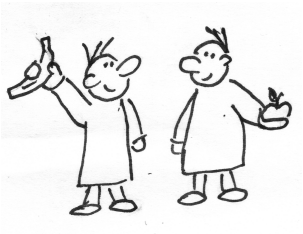
Going Further

- Write an interactive version that takes input directly from the user. Use `while True` to set up an infinite loop and keep using `input` to get the user's next `letter`:

```
$ ./gashlycrumb_interactive.py
Please provide a letter [! to quit]: t
T is for Titus who flew into bits.
Please provide a letter [! to quit]: 7
I do not know "7".
Please provide a letter [! to quit]: !
Bye
```

Chapter 8: Apples and Bananas: Find and replace

Have you ever misspelled a word? I haven't, but I've heard that many other people often do. We can use computers to find and replace all instances of an misspelled word with the correction. Or maybe you'd like to replace all mentions of your ex's name in your poetry with your new love's name? Find and replace is your friend.



To get us started, let us consider the children's song "Apples and Bananas"? After a rousing introduction wherein we consider our favorite fruits to consume:

```
I like to eat, eat, eat apples and bananas
```

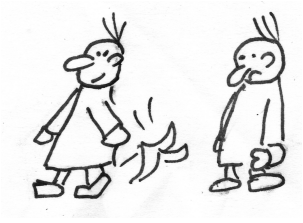
Subsequent verses substitute the main vowel sound in the fruits for various other vowel sounds, such as the long "a" (as in "hay") sound:

```
I like to ate, ate, ate ay-ples and ba-nay-nays
```

Or the ever-popular long "e" (as in "knee"):

```
I like to eat, eat, eat ee-ples and bee-nee-nees
```

And so forth. In this exercise, we'll write a Python program called `apples.py` takes some text given as a single positional argument and replaces all the vowels in the text with a given `-v|--vowel` (default `a`).



The program should handle text on the command line:

```
$ ./apples.py foo
faa
```

And accept the `-v` or `--vowel` option:

```
$ ./apples.py foo -v i
fii
```

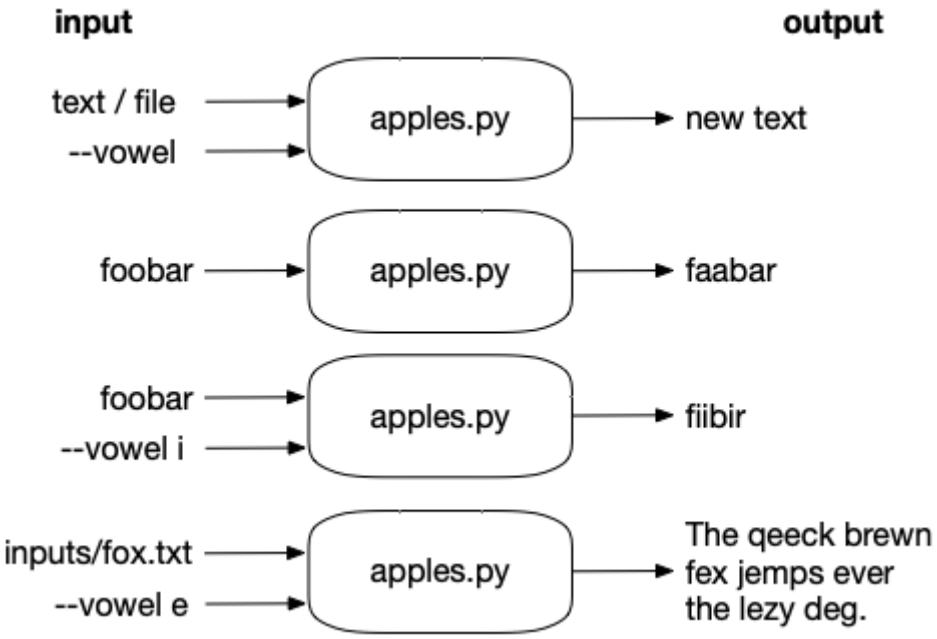
You program should *preserve the case* of the input vowels:

```
$ ./apples.py -v i "APPLES AND BANANAS"
IPPLIS IND BININIS
```

As with the "Howler" program, the text argument may name a file in which case your program should read the contents of the file.

```
$ ./apples.py ../inputs/fox.txt
Tha qaack brawn fax jamps avar tha lazy dag.
$ ./apples.py --vowel e ../inputs/fox.txt
The qeeck brewn fex jemps ever the lezy deg.
```

It might help to look at a diagram of the program's inputs and output:



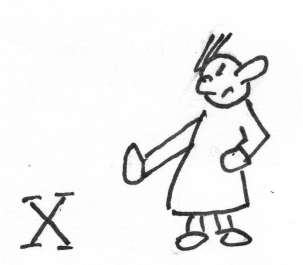
Here is the usage that should print when there are *no argumments*:

```
$ ./apples.py
usage: apples.py [-h] [-v str] str
apples.py: error: the following arguments are required: str
```

And the program should always print usage for the `-h` and `--help` flags:

```
$ ./apples.py -h
usage: apples.py [-h] [-v str] str

Apples and bananas

positional arguments:
  str                   Input text or file

optional arguments:
  -h, --help            show this help message and exit
  -v str, --vowel str   The vowel to substitute (default: a)
```

Note that the program should complain if the `--vowel` argument is not a single, lowercase vowel:



```
$ ./apples.py -v x foo
usage: apples.py [-h] [-v str] str
apples.py: error: argument -v/--vowel: \
invalid choice: 'x' (choose from 'a', 'e', 'i', 'o', 'u')
```

So our program is going to need to:

- Take a positional argument that might be some plain text or may name a file.
- If the argument is a file, use the contents as the input text.
- Take an optional `-v` or `--vowel` argument that should default to the letter "a".
- Verify that the `--vowel` option is in the set of vowels "a," "e," "i," "o," and "u."
- Replace all instances of vowels in the input text with the specified (or default) `--vowel` argument.
- Print the new text to `STDOUT`.

Altering strings

So far in our discussions of Python strings, numbers, lists, and dictionaries, we've seen how easily we can change or *mutate* variables. Something we haven't discussed is that *strings are immutable*. That is, if we have a `text` variable that holds our input text:


```
>>> text = 'The quick brown fox jumps over the lazy dog.'
```

And we wanted to turn the first "e" into an "i," we cannot do this:

```
>>> text[2] = 'i'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

To change `text`, we need to set it equal to an entirely new value. In "Jump the Five" we saw that you can use a `for` loop to iterate over the characters in a string. For instance, I could laboriously uppercase the `text` like so:


```
>>> new = ''                ①
>>> for char in text:       ②
...     new += char.upper() ③
```

① Set a `new` variable equal to the empty string.

② Iterate through each character in the text.

③ Append the uppercase version of the character to the `new` variable.

We can inspect the `new` value to verify that it is all uppercase:

```
>>> new
'THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.'
```

Using this idea, you could iterate through the characters of `text` and build up a new string. Whenever you have a vowel, you substitute the given vowel, otherwise you use the character itself. (We had to identify vowels in Crow's Nest, so you can refer back to how you did that.)

In that same exercise, we saw the `str.replace` and `str.translate` methods that might work. Let's look at the `help`:


```
>>> help(str.replace)
replace(self, old, new, count=-1, /)
    Return a copy with all occurrences of substring old replaced by new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    If the optional argument count is given, only the first count occurrences are
    replaced.
```

Let's play with that in the REPL. I could `replace` "T" for "X". Can you see how to replace all the vowels using this idea?


```
>>> text.replace('T', 'X')
'Xhe quick brown fox jumps over the lazy dog.'
```

The `str.translate` documentation is a bit more cryptic. I showed an example of how to use this in the "Jump The Five" discussion. Can you figure out to use this?


```
>>> help(str.translate)
translate(self, table, /)
    Replace each character in the string using the given translation table.

      table
        Translation table, which must be a mapping of Unicode ordinals to
        Unicode ordinals, strings, or None.

    The table must implement lookup/indexing via __getitem__, for instance a
    dictionary or list.  If this operation raises LookupError, the character is
    left untouched.  Characters mapped to None are deleted.
```

If you know about regular expressions, that's a strong solution. If you haven't heard of those, don't worry as I'll introduce them in the discussion. The point is for you to go *play* with this and come up with a solution. I found 8 ways to change all the vowels to a new character, so there are many ways you could approach this.

How many *different* methods can you find on your own before you look at my solution?

Hints:

- Consider using the `choices` option in the `argparse` documentation for how to constrain the `--vowel` options.
- Be sure to change both lower- and uppercase versions of the vowel, preserving the case of the input characters.

Now is the time to dig in and see what you can do before you look at my solutions. Remember:

> Those hours of practice, and failure, are a necessary part of the learning process. - Gina Sipley

Solution

```
#!/usr/bin/env python3
"""Apples and Bananas"""

import argparse
import os
import re ①


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

    parser = argparse.ArgumentParser(
        description='Apples and bananas',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input text or file') ②

    parser.add_argument('-v',
                        '--vowel',
                        help='The vowel(s) allowed',
                        metavar='str',
                        type=str,
                        default='a',
                        choices=list('aeiou')) ③

    args = parser.parse_args()

    if os.path.isfile(args.text): ④
        args.text = open(args.text).read().rstrip() ⑤

    return args


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

    args = get_args()
    text = args.text
    vowel = args.vowel

    # Method 1: Iterate every character
    new_text = []                           ⑥
    for char in text:                       ⑦
        if char in 'aeiou':                 ⑧
            new_text.append(vowel)          ⑨
```

```
48          elif char in 'AEIOU':               ⑩
49              new_text.append(vowel.upper()) ⑪
50          else:
51              new_text.append(char)           ⑫
52      text = ''.join(new_text)                ⑬
53
54      print(text)                             ⑭
55
56
57  # --------------------------------------------------
58  if __name__ == '__main__':
59      main()
```

① This brings in the regular expression module.

② The input might be text or a file name, so define as a string.

③ Use the `choices` to restrict the user to one of the listed vowels.

④ Check if the `text` argument is a file.

⑤ If it is, read the file, using the `str.rstrip` to remove any trailing whitespace.

⑥ Create a new list to hold the characters we'll select.

⑦ Iterate through each character of the text.

⑧ See if the current character is in the list of lowercase vowels.

⑨ If it is, use the `vowel` instead of the character.

⑩ See if the current character is in the list of uppercase vowels.

⑪ If it is, use the `vowel.upper` instead of the character.

⑫ Otherwise, take the character itself.

⑬ Replace the `text` with a new string made by joining the `new_text` list on the empty string.

⑭ Print the text to `STDOUT`.

Discussion

I'll show seven other ways to solve the `main` part, but they all share the same `get_args`, so let's look at that first.

Defining the parameters

This is one of those problems that has many valid and interesting solutions. The first problem to solve is, of course, getting and validating the user's input. As always, I will use `argparse`. I usually define all my required parameters first. The `text` parameter is a positional string that *might* be a file name:

```
parser.add_argument('text', metavar='str', help='Input text or file')
```

The `--vowel` option is also a string, and I decided to use the `choices` option to have `argparse` validate that the user's input is in the `list('aeiou')`:

```
parser.add_argument('-v',
                    '--vowel',
                    help='The vowel to substiute'',
                    metavar='str',
                    type=str,
                    default='a',
                    choices=list('aeiou'))
```

That is, `choices` wants a `list` of options. I could pass in `['a', 'e', 'i', 'o', 'u']`, but that's a lot of typing on my part. It's must easier to type `list('aeiou')` to have Python turn the `str` "aeiou" into a `list` of the characters. Both of these are the same:

```
>>> ['a', 'e', 'i', 'o', 'u']
['a', 'e', 'i', 'o', 'u']
>>> list('aeiou')
['a', 'e', 'i', 'o', 'u']
```

We can even write a test for this. No `AssertionError` error means that it's OK:

```
>>> assert ['a', 'e', 'i', 'o', 'u'] == list('aeiou')
```

The next task is detecting if `text` is the name of a file that should be read for the text or is the text itself. This is the same code I used in Howler, and again I choose to handle the `text` argument inside the `get_args` function so that, by the time I get `text` inside my `main`, it's all be handled:

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```

At this point, the user's arguments to the program have been fully vetted. We've got `text` either from the command line or from a file, and we've verified that the `--vowel` is actually one of the allowed characters. To me, this is a single "unit" where I've handled the arguments, and processing can now go forward by returning the arguments:

```
return args
```

Eight ways to replace the vowels

How many ways did you find to replace the vowels? You only needed one, of course, to pass the test, but I hope you probed the edges of the language to see how many different techniques there are. I know that the Zen of Python says:

> There should be one — and preferably only one — obvious way to do it. - https://www.python.org/dev/peps/pep-0020/

But I really come from the Perl mentality that "There Is More Than One Way To Do It" (TIMTOWTDI or "Tim Toady").

Method 1: Iterate every character

The first method is similar to "Jump The Five" where we can use a `for` loop on a string to access each character. Here is code you can copy and paste into the `ipython` REPL

```
>>> text = 'Apples and Bananas!'           ①
>>> vowel = 'o'                            ②
>>> new_text = []                          ③
>>> for char in text:                      ④
...     if char in 'aeiou':                ⑤
...         new_text.append(vowel)         ⑥
...     elif char in 'AEIOU':              ⑦
...         new_text.append(vowel.upper()) ⑧
...     else:
...         new_text.append(char)          ⑨
...
>>> text = ''.join(new_text)               ⑩
>>> text
'Opplos ond Bononos!'
```

① Set a `text` variable to the string "Apples and Bananas!"

② Set the `vowel` variable to the string "o."

③ Set the variable `new_text` to an empty `list`.

④ Use a `for` to iterate `text`, putting each character into the `char` variable.

⑤ If the character is in the set of lowercase vowels,

⑥ Substitute in the `vowel` to the `new_text`.

⑦ If the character is in the set of uppercase vowels,

⑧ Substitute the `vowel.upper()` version into `new_text`.

⑨ Otherwise, use the `char` as-is.

⑩ Turn the `list` called `new_text` into new `str` by joining it on the empty string (`''`).

Note that it would be just fine to start off making `new_text` an empty string and then concatenating

the new characters. Then you wouldn't have to `join` them at the end. Whatever you prefer:

```
new_text += vowel
```

Method 2: str.replace

For the alternate solutions, I'll just show the `main` function:

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel

    # Method 2: str.replace
    for v in 'aeiou': ①
        text = text.replace(v, vowel).replace(v.upper(), vowel.upper()) ②

    print(text)
```

① Iterate through the list of vowels. We don't have to say `list('aeiou')` here because Python will automatically treat the string `'aeiou'` like a `list` because we are using it in a *list context* with the `for`.

② Use the `str.replace` method twice to replace both the lower- and upper-case versions of the vowel in the `text`.

I mentioned in the introduction the `str.replace` method that will return a new string with all instances of one string replaced by another.

```
>>> s = 'foo'
>>> s.replace('o', 'a')
'faa'
>>> s.replace('oo', 'x')
'fx'
```

Note that the original string remains unchanged:

```
>>> s
'foo'
```

You don't have to chain the two `replace` methods. The longer way to write this would be this.

```
for v in 'aeiou':
    text = text.replace(v, vowel)
    text = text.replace(v.upper(), vowel.upper())
```

Method 3: str.translate

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel

    # Method 3: str.translate
    trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5) ①
    text = text.translate(trans) ②

    print(text)
```

① Create the translation table.

② Call the `str.translate` method on the `text` variable passing the `trans` table as the argument.

How can we use `str.translate` method to solve this? I showed in "Jump The Five" how the `jumper` dictionary could be used to create a translation table using the `str.maketrans` method to convert each number to another number. In this problem I need to change all the lower- and upper-case vowels (10 total) to some given `vowel`. That is, to make all the vowels into the letter "o," I want to create something like this:

```
a => o
e => o
i => o
o => o
u => o
A => O
E => O
I => O
O => O
U => O
```

To start, I need to repeat my `vowel` five times, which I can do using the `*` operator that you normally associate with numeric multiplication. This is (sort of) "multiplying" a string, so, OK, I guess:

```
>>> vowel * 5
'ooooo'
```

Now to handle the uppercase version, too:

```
>>> vowel * 5 + vowel.upper() * 5
'ooooo00000'
```

Now to make the translation table:

```
>>> trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5)
```

Let's inspect the `trans` table. I want to "pretty print" the data structure so I can see it, so I will use the `pprint.pprint (pretty print) function:

```
>>> from pprint import pprint as pp
>>> pp(trans)
{65: 79,
 69: 79,
 73: 79,
 79: 79,
 85: 79,
 97: 111,
 101: 111,
 105: 111,
 111: 111,
 117: 111}
```

The enclosing curlies `{}` tell us that `trans` is a `dict`. Each character is represented by it's *ordinal* value, which is the character's position in the ASCII table (http://www.asciitable.com/). (You will use this later in the "Gematria" program.) You can go back and forth from characters and their ordinal values by using `chr` and `ord`. Here are the `ord` values for the vowels:

```
>>> for char in 'aeiou':
...     print(char, ord(char))
...
a 97
e 101
i 105
o 111
u 117
```

And here you can create the same output but starting with the `ord` values to get the `chr` values:

```
>>> for num in [97, 101, 105, 111, 117]:
...     print(chr(num), num)
...
a 97
e 101
i 105
o 111
u 117
>>>
```

If you'd like to inspect all the ordinal values for all the printable characters, you can run this:

```
>>> import string
>>> for char in string.printable:
...     print(char, ord(char))
```

I don't include the output because there are 100 printable characters:

```
>>> print(len(string.printable))
100
```

So the `trans` table is a mapping. The lowercase vowels ("aeiou") all map to the ordinal value `111` which is "o." The uppercase vowels ("AEIOU") map to 79 which is "O." I can use the `dict.items` method to iterate over the key/value pairs of `trans` to verify this is the case:

```
>>> for x, y in trans.items():
...     print(f'{chr(x)} => {chr(y)}')
...
a => o
e => o
i => o
o => o
u => o
A => O
E => O
I => O
O => O
U => O
```

The original `text` will be unchanged by the `translate` method, so we overwrite `text` with the new version:

```
>>> text = 'Apples and Bananas!' ①
>>> trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5) ②
>>> text = text.translate(trans) ③
>>> text
'Opplos ond Bononos!'
```

① Initialize `text`.

② Make a translation table.

③ Use the translation table as the argument to `text.translate`. Overwrite the original value of `text` with the result.

That was a lot of explanation about `ord` and `chr` and dictionaries and such, but look how simple and elegant that solution is! This is much shorter than Method 1. Fewer lines of code (LOC) means fewer opportunities for bugs!

Method 4: List comprehension

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel

    # Method 4: Use a list comprehension
    new_text = [ ①
        vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c ②
        for c in text
    ]
    text = ''.join(new_text) ③

    print(text)
```

① Use a list comprehension.

② Use a compound `if` expression to handle three cases (lowercase vowel, uppercase vowel, the default).

③ Replace the `text` with the `new_text` values.

Following up on Method 1, we can use a "list comprehension" to significantly shorten the `for` loop. In Gashlycrumb we looked at a "dictionary comprehension" as a one-line method to create a new dictionary using a `for` loop. Here we can do the same, creating a new `list`.

For example, let's generate a list of the squared values of the numbers 1 through 4. We can use the `range` function to get the numbers from a starting number to an ending number (not inclusive!). `range` is a *lazy* function, which means it won't actually produce values until your program actually needs them. In the REPL, I must use `list` to see the values, but most of the time your code doesn't need to do this:

```
>>> list(range(1, 5))
[1, 2, 3, 4]
```

I can write a `for` loop to `print` the squares:

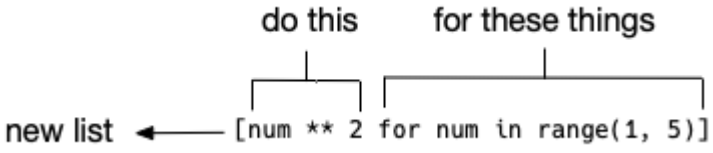
```
>>> for num in range(1, 5):
...     print(num ** 2)
...
1
4
9
16
```

But what I really want is a `list` that contains those values. A simple way to do this is to create an empty `list` to which we will `append` those values:

```
>>> squares = []
>>> for num in range(1, 5):
...     squares.append(num ** 2)
```

And now I can verify that I have my squares:

```
>>> assert len(squares) == 4
>>> assert squares == [1, 4, 9, 16]
```



A list comprehension is novel from a `for` loop in that it *returns a new list*:

```
>>> [num ** 2 for num in range(1, 5)]
[1, 4, 9, 16]
```

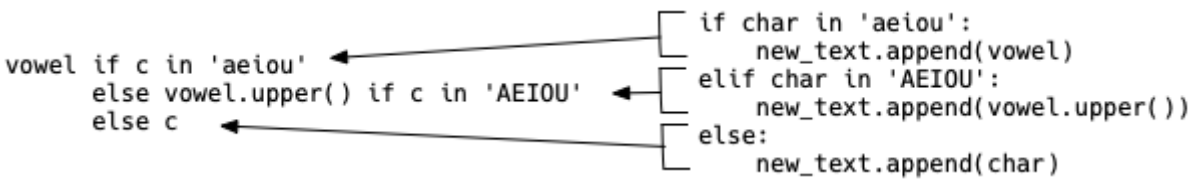
I can assign this to the `squares` variable and verify that I still have what I expected. Ask yourself which version of the code you'd rather maintain, the longer one with the `for` loop or the shorter one with the list comprehension?

```
>>> squares = [num ** 2 for num in range(1, 5)]
>>> assert len(squares) == 4
>>> assert squares == [1, 4, 9, 16]
```

For our program, we're going to condense the `if`/`elif`/`else` logic from Method 1 into a compound `if` expression. First let's see how we could shorter the `for` loop version:

```
>>> text = 'Apples and Bananas!'
>>> new_text = []
>>> for c in text:
...     new_text.append(vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else
c)
...
>>> ''.join(new_text)
'Opplos ond Bononos!'
```

Here's a diagram that shows how the parts of the expression match up to the original `if`/`elif`/`else`:



And now to turn that into a list comprehension:

```
>>> text = 'Apples and Bananas!'
>>> new_text = [
...     vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c ①
...     for c in text ] ②
...
>>> ''.join(new_text)
'Opplos ond Bononos!'
```

① Do this

② For these things.

You have to find the start of the `for` loop `for c in text` which is "for character in text." We then use our handy compound `if` expression to decide whether to return the chosen `vowel if c in 'aeiou'` or the same check with the upper-case version, and finally we default to the character `c` itself if it fails both of those conditions.

Method 5: List comprehension with function

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel

    # Method 5: Define a function, use list comprehension
    def new_char(c): ①
        return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c ②

    text = ''.join([new_char(c) for c in text]) ③

    print(text)
```

① Define a function that closes over the `vowel`.

② Use the compound `if` expression to select the correct character.

③ Use a list comprehension to process all the characters in `text`.

Do you notice that the compound `if` expression in the previous method smells a bit like a function? Here's a way we could write this:

```
>>> vowel = 'o'
>>> def new_char(c):
...     return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c
...
```

It should always return the letter "o" if the argument is a lowercase vowel:

```
>>> new_char('a')
'o'
```

And "O" if the argument is an uppercase vowel:

```
>>> new_char('A')
'O'
```

Otherwise it should return the given character:

```
>>> new_char('b')
'b'
```

We can use our handy `new_char` function like so:

```
>>> text = 'Apples and Bananas!'
>>> text = ''.join([new_char(c) for c in text])
>>> text
'Opplos ond Bononos!'
```

A note about the fact that the `new_char` function is declared **inside** the `main` function. Yes, you can do that! The function is then only "visible" inside the `main` function. Here I define a `foo` function that has a `bar` function inside it. I can call `foo` and it will call `bar`, but from outside of `foo` the `bar` function does not exist ("is not visible" or "is not in scope"):

```
>>> def foo():
...     def bar():
...         print('This is bar')
...     bar()
...
>>> foo()
This is bar
>>> bar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bar' is not defined
```

I did this because I actually created a special type of function with `new_char` called a "closure" because it is "closing" around the `vowel`. If I had defined `new_char` outside of `main`, the `vowel` would not be visible to `new_char` because it only exists inside the `main` function. If we draw a box around the `main` function, that is where the `new_char` function can be seen. Outside of that, the function is consider undefined.

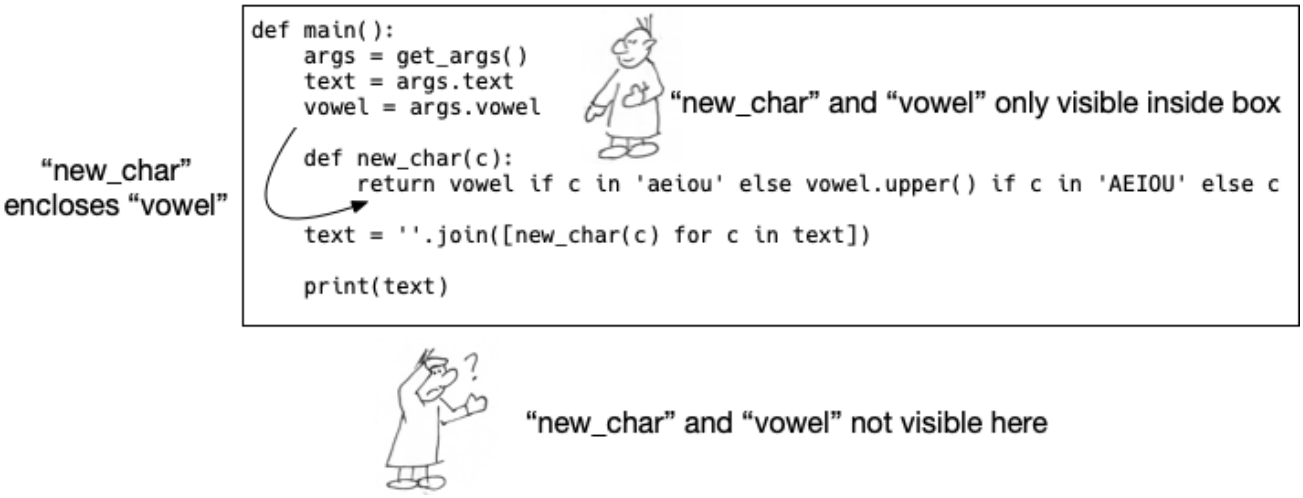


Figure 5. Visibility of the `new_char` *function*

There are many reasons why to write a function like this, even if it has just one line. I think of functions as *units* of code that describe some concept — ideally just *one* per function! We can formalize our understand of our functions with assertions:

```
>>> assert all([new_char(v) == 'O' for v in 'AEIOU'])
>>> assert all([new_char(v) == 'o' for v in 'aeiou'])
```

In two lines of code, I've just tested all 10 vowels by using list comprehensions plus the `all` function. Let's take a moment to understand `all` and `any`. If you read `help(all)`, you'll see "Return True if bool(x) is True for all values x in the iterable." So all the values need to the `True` for the entire expression to be `True`:

```
>>> all([True, True, True])
True
```

If any value is `False`, then the whole expression is `False`:

```
>>> all([True, False, True])
False
```

The `any` function returns `True` if *any* of the values are `True`:

```
>>> any([True, False, True])
True
```

And `False` if there are no `True` values:

```
>>> any([False, False, False])
False
```

We can check the values manually:

```
>>> [new_char(v) == 'O' for v in 'AEIOU']
[True, True, True, True, True]
```

So then `all` should be `True` for the entire expression:

```
>>> all([new_char(v) == 'O' for v in 'AEIOU'])
True
```

We can move these into formal `test_` functions that can be run with `pytest`. This is the idea of *unit testing* where I personally think of functions as units. Some people have other ideas about what is the best *unit* to test. I recommend you read further about testing to understand the range of opinions.

Method 6: `map` with a `lambda`

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel

    # Method 6: Use a `map` to iterate with a `lambda`
    text = ''.join( ①
        map(        ②
            lambda c: vowel if c in 'aeiou' else vowel.upper() ③
            if c in 'AEIOU' else c, text)) ④

    print(text)
```

① The `map` will return a new `list`, so here we'll put it back into a string using `str.join`.

② The `map` function wants a function for the first argument and a `list` for the second.

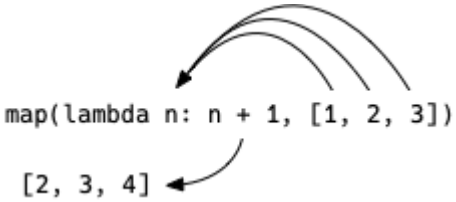
③ Use `lambda` to create an *anonymous* function that accepts a single `c` (character)

④ The `text` is the second argument to `map`. Because `map` expects this to be a `list`, it will treat the string `text` like a list of characters (which is what we want).

If you are comfortable with list comprehensions, then it's a short step to using the `map` function. Functions like `map` and another we'll use later called `filter` are in the class of "higher-order functions" because they take *other functions* as arguments, which is wicked cool.

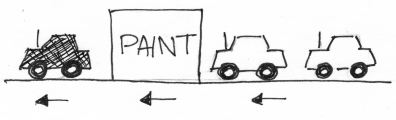
The `map` function accepts two arguments:

1. A function
2. An iterable like a list, lazy function, or generator.



The `map` function applies the given function to every member of the iterable. For instance, say I want to increment a series of numbers by 1, I could use this `map`. Note the use of `list` because `map` is another lazy function like `range`:

```
>>> list(map(lambda n: n + 1, [1, 2, 3]))
[2, 3, 4]
```



I like to think of `map` like a paint booth: You load up the booth with, say, blue paint. Unpainted cars go in, blue paint is applied, and blue cars come out. I can create a function to "paint" my cars by adding the string "blue" to the beginning:

```
>>> list(map(lambda car: 'blue ' + car, ['BMW', 'Alfa Romeo', 'Chrysler']))
['blue BMW', 'blue Alfa Romeo', 'blue Chrysler']
```

The `lambda` keyword is used to create an *anonymous* function. That is, with the regular `def` keyword, the function name follows. With `lambda`, there is no name, only the list of parameters and the function body.

Think about regular named functions like `add1` that adds `1` to a value:

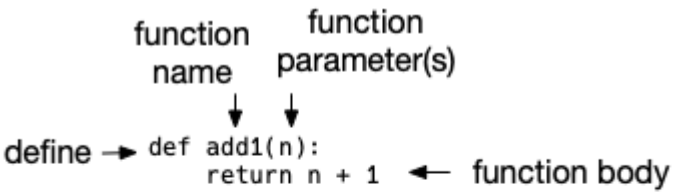
```
>>> def add1(n):
...     return n + 1
```

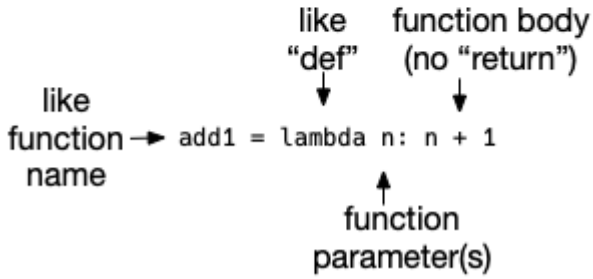
It works as expected:

```
>>> assert add1(10) == 11
>>> assert add1(add1(10)) == 12
```

Now compare to the `lambda` version. We can assign it to the variable `add1` which then kinda sorta acts like the function's name:

```
>>> add1 = lambda n: n + 1
```





It works exactly the same as using `def` to define a function the normal way:

```
>>> assert add1(10) == 11
>>> assert add1(add1(10)) == 12
```

The function body for a `lambda` pretty much needs to fit on one line, and they don't have `return` at the end. In both versions, the argument to the function is `n`. In the usual `def add(n)`, the argument is defined in the parentheses just after the function name. In the `lambda n` version, there is no function name and we just define the argument `n`.

There is no difference in how you can use them. They are both functions:

```
>>> type(lambda x: x)
<class 'function'>
```

There is a decent amount of debate on exactly when one should use `lambda` to create a function and when `def`. Some linters like `pylint` will complain if you use a `lambda` anywhere but as the argument to `map` and `filter`, but I personally find reasons to assign `lambda` expressions to variables and pass them around. It's a personal style issue, and I say you do you.

Here is how I could cram the idea of the `new_char` function into a `lambda`:

```
>>> text = 'Apples and Bananas!'
>>> text = ''.join(map(lambda c: vowel if c in 'aeiou' else vowel.upper()
...                    if c in 'AEIOU' else c, text))
>>> text
'Opplos ond Bononos!'
```

Method 7: `map` with `new_char`

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel

    # Method 7: `map` with the function
    def new_char(c): ①
        return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c

    text = ''.join(map(new_char, text)) ②

    print(text)
```

① Define a function that will return the proper character.

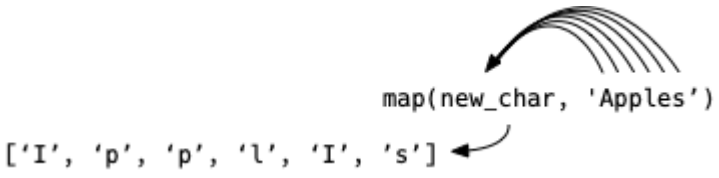
② Use `map` to apply this *named* function to all the characters in `text`. The result is a `list` which is joined into a new string to overwrite `text`.

I feel the previous version is not exactly easy to read. Instead of using a `lambda`, I can use the `new_char` function we previously defined. To me, this is the cleanest and most readable solution:

```
>>> text = 'Apples and Bananas!'
>>> text = ''.join(map(new_char, text))
>>> text
'Opplos ond Bononos!'
```

Notice that `map` takes `new_char` *without parentheses* as the first argument. If you added the parens, you'd be *calling* the function and would see this error:

```
>>> text = ''.join(map(new_char(), text))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: new_char() missing 1 required positional argument: 'c'
```



What happens is that `map` takes each character from `text` and passes it as the argument to the `new_char` function which decides whether to return the `vowel` or the original character. The result of mapping these characters is a new list of characters that we `join` on the empty string to create a new version of `text`.

Method 8: Regular expressions

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel

    # Method 8: Regular expressions
    text = re.sub('[aeiou]', vowel, text)         ①
    text = re.sub('[AEIOU]', vowel.upper(), text) ②

    print(text)
```

① Substitute any of the lowercase vowels for the given vowel (which is lowercase because of the restrictions in `get_args`).

② Substitute any of the uppercase vowels for the uppercased vowel.

The last method I will introduce uses regular expressions which are a separate domain-specific language (DSL) you can use to describe patterns of text. They are incredibly powerful and well worth the effort to learn them. To use them in your program, you `import re` and then use the `re.sub` function to *substitute* all instances of text matching a given pattern for a new string.

We can use square brackets around the vowels `'[aeiou]'` to create a "character class" meaning anything matching one of the characters listed inside the brackets. The second argument is the string that will replace the found strings — here our `vowel` provided by the user. The third argument is the string we want to change, the `text` from the user.

```
>>> import re
>>> text = 'Apples and Bananas!'
>>> vowel = 'o'
>>> re.sub('[aeiou]', vowel, text)
'Applos ond Bononos!'
```

Note that `re.sub` returns a new string, and the original `text` remains unchanged by the operation:

```
>>> text
'Apples and Bananas!'
```

That almost worked, but it missed the uppercase vowel "A". I could overwrite the `text` in two steps to get both lower- and uppercase:

```
>>> text = re.sub('[aeiou]', vowel, text)
>>> text = re.sub('[AEIOU]', vowel.upper(), text)
>>> text
'Opplos ond Bononos!'
```

Or do it in one step just like the `str.replace` method above:

```
>>> text = 'Apples and Bananas!'
>>> text = re.sub('[AEIOU]', vowel.upper(), re.sub('[aeiou]', vowel, text))
>>> text
'Opplos ond Bononos!'
```

One of the biggest differences with this solution to all the others is how we use regular expressions to describe what we were looking for and didn't have to write the code to actually find the characters! This is more along the lines of *declarative* programming. We declare what we want, and the computer does the grunt work!

Refactoring with tests

There are many ways to solve this problem. The most important step is to get your program to work properly. Tests let you know when you've reached that point. From there, you can explore other ways to solve the problem and keep using the tests to ensure you still have a correct program. Tests actually provide great freedom to be creative. Always be thinking about tests you can write for your own programs so that, when you change them later, they will always keep working!

Review

- You can use `argparse` to limit an argument's values to a `list` of `choices` that you define.
- Strings cannot be directly modified, but the `replace` and `translate` methods can create a *new, modified string* from an existing string.
- A `for` loop on a string will iterate the characters of the string.
- A list comprehension is a short-hand way to write a `for` loop inside `[]` to create a new `list`.
- Functions can be defined inside other functions. Their visibility is then limited to the enclosing function. Further, if that enclosed function incorporates a value inside the body, it creates a "closure."
- The `map` function is similar to a list comprehension. It will create a new, modified list by applying some function to every member of a given list. The original list will not be changed.
- Regular expressions provide a syntax for describing patterns of text with the `re` module. The `re.sub` method will substitute found patterns with new text. The original text will be unchanged.

Going Further

- I will be talking more about higher-order functions like `map` and `filter` as go along. You should search for "functional programming in Python" to explore more of these ideas.
- These ideas come from "functional programming" languages which might sound weird because Python and most every other language all have "functions." It's more correct to talk about *purely* functional languages, however, like Haskell, where programs are made *entirely* of functions which are fit together quite rigorously using the types of arguments they accept and return. My style of writing Python is influenced by languages like this, for better or worse.
- Regular expression were invented in the 1950s by Stephen Cole Kleene. They are widely used in Unix tools that manipulate text and in almost every programming language from Perl to Java. They are cryptic at first but wildly useful. No time spent learning regular expressions is wasted time!

Chapter 9: Dial-A-Curse: Generating random insults from lists of words

> "He or she is a slimy-sided, frog-mouthed, silt-eating slug with the brains of a turtle." — Dial-A-Curse

Random events are at the heart of interesting games and puzzles. Humans quickly grow bored of things that are always the same, so let's learn how to make our programs more interesting by having them behave differently each time they are run. This exercise will introduce how to randomly select one or more elements from lists of options. To explore randomness, we'll create a program called `abuse.py` that will insult the user by randomly selecting adjectives and nouns to create slanderous epithets.

In order to test randomness, though, we need to control it. It turns out that "random" events on computers are rarely actually random but only "pseudo-random," which means we can control them using a "seed."[2] Each time you use the same seed, you get the same "random" choices!

Shakespeare had some of the best insults, so we'll draw from the vocabulary of his works. Here is the list of adjectives you should use:

> bankrupt base caterwauling corrupt cullionly detestable dishonest false filthsome filthy foolish foul gross heedless indistinguishable infected insatiate irksome lascivious lecherous loathsome lubbery old peevish rascaly rotten ruinous scurilous scurvy slanderous sodden-witted thin-faced toad-spotted unmannered vile wall-eyed

And these are the nouns:

> Judas Satan ape ass barbermonger beggar block boy braggart butt carbuncle coward coxcomb cur dandy degenerate fiend fishmonger fool gull harpy jack jolthead knave liar lunatic maw milksop minion ratcatcher recreant rogue scold slave swine traitor varlet villain worm

For instance, it might produce the following:

```
$ ./abuse.py
You slanderous, rotten block!
You lubbery, scurilous ratcatcher!
You rotten, foul liar!
```

In this exercise, you will learn to:

- Use `parser.error` from `argparse` to throw errors
- Learn about random seeds to control randomness
- Take random choices and samples from Python lists
- Iterate an algorithm a specified number of times with a `for` loop
- Format output strings

Writing `abuse.py`

The arguments to this program are options that have default values, meaning it can run with no arguments at all. The `-n` or `--number` option will default to `3` and will determine how many insults are created:

```
$ ./abuse.py --number 2
You filthsome, cullionly fiend!
You false, thin-faced minion!
```

And the `-a` or `--adjectives` option should default to `2` and will determine how many adjectives are used in each insult:

```
$ ./abuse.py --adjectives 3
You caterwauling, heedless, gross coxcomb!
You sodden-witted, rascaly, lascivious varlet!
You dishonest, lecherous, foolish varlet!
```

Lastly, your program should accept a `-s|--seed` argument (default `None`) that will control the randomness of the program. The following should be exactly reproducible, no matter who runs the program on any machine at any time:

```
$ ./abuse.py --seed 1
You filthsome, cullionly fiend!
You false, thin-faced minion!
You sodden-witted, rascaly cur!
```

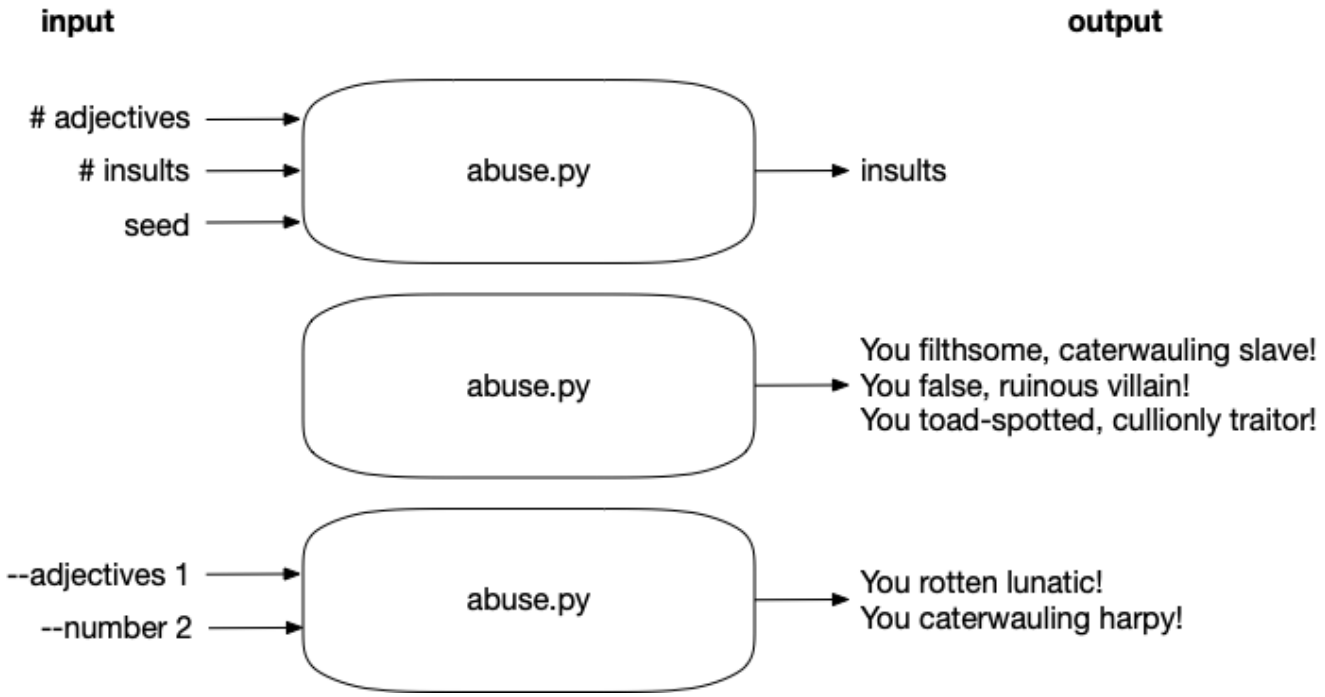
When run with no arguments, the program should generate insults using the defaults:

```
$ ./abuse.py
You foul, false varlet!
You filthy, insatiate fool!
You lascivious, corrupt recreant!
```

I recommend you start by copying the `template/template.py` to `abuse/abuse.py` or by using `new.py` to create the `abuse.py` program in the `abuse` directory of your repository. The next step is to make your program produce the following usage for `-h` or `--help`:

```
$ ./abuse.py -h
usage: abuse.py [-h] [-a int] [-n int] [-s int]

Heap abuse

optional arguments:
  -h, --help            show this help message and exit
  -a int, --adjectives int
                        Number of adjectives (default: 2)
  -n int, --number int  Number of insults (default: 3)
  -s int, --seed int    Random seed (default: None)
```

Here is a string diagram to help you see the program:



Validating arguments

All of the options for number of insults and adjectives as well as the random seed should all be `int` values. If you define each using `type=int` (remember there are no quotes around the `int`), then `argparse` will handle the validation and conversion of the arguments to an `int` value for you. That is, just by defining `type=int`, the following error will be generated for you:

```
$ ./abuse.py -n foo
usage: abuse.py [-h] [-a int] [-n int] [-s int]
abuse.py: error: argument -n/--number: invalid int value: 'foo'
```

Additionally, if either `--number` or `--adjectives` less than 1, your program should exit with an error code and message:

```
$ ./abuse.py -a -4
usage: abuse.py [-h] [-a int] [-n int] [-s int]
abuse.py: error: --adjectives "-4" must be > 1
$ ./abuse.py -n -4
usage: abuse.py [-h] [-a int] [-n int] [-s int]
abuse.py: error: --number "-4" must be > 1
```

Let's take a look at one of the tests in `test.py` to understand what is expected:

```
def test_bad_adjective_num():                                  ①
    """bad_adjectives"""

    n = random.choice(range(-10, 0))                           ②
    rv, out = getstatusoutput(f'{prg} -a {n}')                 ③
    assert rv != 0                                             ④
    assert re.search(f'--adjectives "{n}" must be > 0', out)    ⑤
```

① The name of the function must start with `test_` in order for `pytest` to find and run it.

② Use the `random.choice` function to randomly select a value from the `range` of numbers from -10 to 0. We will use this same function in our program, so note here how it is called!

③ Run the program with the bad `-a` argument, getting back a return value (`rv`) and the standard out (`out`).

④ Assert that the return value is not 0 where "0" would indicate success (or "zero errors").

⑤ Assert that the output somewhere contains the statement that the `--adjectives` argument must be greater than 0.

I would recommend you look at the section on "Manually checking arguments" in chapter 1. There I introduce the `parser.error` function that you can call inside the `get_args` function to do the following:

1. Print the short usage statement.
2. Print an error message to the user.
3. Stop execution of the program.
4. Exit with a non-zero exit value to indicate an error.

That is, your `get_args` normally finishes with:

```
return args.parse_args()
```

Instead, put the `args` into a variable and check the `args.adjectives` value to see if it's less than 1:

```
args = parser.parse_args()

if args.adjectives < 1:
    parser.error('--adjectives "{}" must be > 0'.format(args.adjectives))
```

Also do this for the `args.number`. If they are both fine, then you can `return` the arguments to the calling function:

```
return args
```

Importing and seeding the `random` module

Once you have defined and validated all the program's arguments, you are ready to heap scorn upon the user. We need to add `import random` to our program so we can use functions from that module to select adjectives and nouns. I recommend you add it just after the `import argparse` statement near the top of your program.

As usual first thing we need to do in the `main` is to call `get_args` to get our arguments:

```
args = get_args()
```

And the very next step is to pass the `args.seed` value to the `random.seed` function:

```
random.seed(args.seed)
```

We can read about the `random.seed` function in the REPL:

```
>>> import random
>>> help(random.seed)
```

There we learn that the function does the following:

```
Initialize internal state from hashable object.
```

That is, we set an initial value from some *hashable* Python type. Both `int` and `str` types are hashable, and I suggest you define the `seed` argument as an `int`. The default value for `args.seed` is `None`, so, if the user has not indicated any seed, this is the same as not setting it at all.

If you look at the `test.py` program, you will notice that all the tests that expect a particular output pass a `-s` argument. Here is the first test for output:

```
def test_01():
    out = getoutput(f'{prg} -s 1 -n 1') ①
    assert out.strip() == 'You filthsome, cullionly fiend!' ②
```

① Run the program with the seed value 1, requesting 1 insult.

② Verify that the entire output is the one expected insult.

Defining the adjectives and nouns

Above I've given you a long list of adjectives and nouns that you should use in your program. In order to pass the tests, they must be in the same order as I have provided. (You may notice that they are alphabetically sorted.) You could create a `list` by individually quoting each word:

```
>>> adjectives = ['bankrupt', 'base', 'caterwauling']
```

Or you could save yourself a good bit of typing if you use the `str.split` function to create a new `list` from a `str` by splitting on spaces:

```
>>> adjectives = 'bankrupt base caterwauling'.split()
>>> adjectives
['bankrupt', 'base', 'caterwauling']
```

If you try to make one giant string of all the adjectives, it will wrap around and look ugly. I'd recommend you use triple quotes (either single or double quotes) that allow you to include newlines:

```
>>> """
... bankrupt base
... caterwauling
... """.split()
['bankrupt', 'base', 'caterwauling']
```

Once you have variables for `adjectives` and `nouns`, you might check that you have the right number:

```
>>> assert len(adjectives) == 36
>>> assert len(nouns) == 39
```

Taking random samples and choices

In addition to the `random` module's `seed` function, we will also use the `choice` and `sample` functions. In the `test_bad_adjective_num` function above, we saw one example of using `random.choice`. We can use it similarly to select a noun from the `list` of `nouns`. Notice that this function returns a single item, so, given a `list` of `str` values, it will return a single `str`:

```
>>> random.choice(nouns)
'braggart'
>>> random.choice(nouns)
'milksop'
```

For the `adjectives`, you should use `random.sample`. If you read the `help(random.sample)`, you will see this function takes the `list` of `adjectives` and a `k` parameter for how many items to sample:

```
sample(population, k) method of random.Random instance
    Chooses k unique random elements from a population sequence or set.
```

Note that this function returns a new `list`:

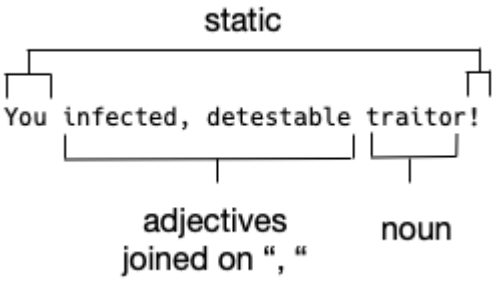
```
>>> random.sample(adjectives, 2)
['detestable', 'peevish']
>>> random.sample(adjectives, 3)
['slanderous', 'detestable', 'base']
```

There is also a `random.choices` that works similarly to `sample` but which might select the same items twice because it samples "with replacement," so we will not use that.

Formatting the output

The output of the program is some `--number` of insults, which you could generate using a `for` loop and the `range` function. It doesn't matter here that `range` starts at zero. What's important is that it generated three values. This all we need if we want to generate three insults:

```
>>> for n in range(3):
...   print(n)
...
0
1
2
```



You can loop the `--number` of times needed, select your sample of adjectives and your noun, and then format the output. Each insult should start with the string "You ", then have the adjectives joined on a comma and a space, then the noun, and finish with an exclamation point. You could use either an f-string or the `str.format` function to `print` the output to `STDOUT`.

Hints:

- Perform the check for positives values for `--adjectives --number` *inside* the `get_args` function and use `parser.error` to throw the error while printing a message and the usage.
- If you set the default for `args.seed` to `None` while using a `type=int`, you should be able to directly pass the argument's value to `random.seed` to control testing.
- Use a `for` loop with the `range` function to create a loop that will execute `--number` of times to generate each insult.
- Look at the `sample` and `choice` functions in the `random` module for help in selecting some adjectives and a noun.
- You can use three single (`'''`) or double quotes (`"""`) to create a multi-line string and then `split()` that to get a list of strings. This is easier than individually quoting a long list of shorter strings (e.g., the list of adjectives and nouns).
- To construct an insult string to print, you can use the `+` operator to concatenate strings, use the `str.join` method, or use format strings.

Now give this your best shot before reading ahead to the solution!

[2] "The generation of random numbers is too important to be left to chance." — Robert R. Coveyou

Solution

```
#!/usr/bin/env python3
"""Heap abuse"""

import argparse
import random                                           ①


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

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

    parser.add_argument('-a',                           ②
                        '--adjectives',
                        help='Number of adjectives',
                        metavar='int',
                        type=int,
                        default=2)

    parser.add_argument('-n',                           ③
                        '--number',
                        help='Number of insults',
                        metavar='int',
                        type=int,
                        default=3)

    parser.add_argument('-s',                           ④
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    args = parser.parse_args()                          ⑤

    if args.adjectives < 1:                             ⑥
        parser.error('--adjectives "{}" must be > 0'.format(args.adjectives))

    if args.number < 1:                                 ⑦
        parser.error('--number "{}" must be > 0'.format(args.number))

    return args                                         ⑧


```

```
48  # --------------------------------------------------
49  def main():
50      """Make a jazz noise here"""
51
52      args = get_args()                                   ⑨
53      random.seed(args.seed)                              ⑩
54
55      adjectives = """                                    ⑪
56      bankrupt base caterwauling corrupt cullionly detestable dishonest false
57      filthsome filthy foolish foul gross heedless indistinguishable infected
58      insatiate irksome lascivious lecherous loathsome lubbery old peevish
59      rascaly rotten ruinous scurilous scurvy slanderous sodden-witted
60      thin-faced toad-spotted unmannered vile wall-eyed
61      """.strip().split()
62
63      nouns = """                                         ⑫
64      Judas Satan ape ass barbermonger beggar block boy braggart butt
65      carbuncle coward coxcomb cur dandy degenerate fiend fishmonger fool
66      gull harpy jack jolthead knave liar lunatic maw milksop minion
67      ratcatcher recreant rogue scold slave swine traitor varlet villain worm
68      """.strip().split()
69
70      for _ in range(args.number):                        ⑬
71          adjs = ', '.join(random.sample(adjectives, k=args.adjectives)) ⑭
72          print(f'You {adjs} {random.choice(nouns)}!')  ⑮
73
74
75  # --------------------------------------------------
76  if __name__ == '__main__':
77      main()
```

① Bring in the `random` module so we can use functions.

② Define the parameter for the number of adjectives, setting the `type=int` and the default value.

③ Similarly define the parameter for the number insults.

④ The random seed default should be `None`.

⑤ Get the `args` from parsing the argument. Error such as non-integer values will be handled at this point by `argparse`.

⑥ If we make it to this point, we can perform additional checking such as verifying that the `args.adjectives` is greater than 0. If there is a problem, call `parser.error` with the error message to print.

⑦ Similarly check for the `args.number`.

⑧ If we make it to this point, then all the user's arguments have been validated.

⑨ This is where the program actually begins as it is the first action inside our `main`. We always start off by getting the arguments.

⑩ Set the `random.seed` using whatever value was passed by the user. Any `int` value is valid, and we know that `argparse` has handled the validation and conversion of the argument to an `int`.

⑪ Create a `list` of `adjectives` by splitting the very long string contained in the triple quotes.

⑫ Do the same for the `list` of `nouns`.

⑬ Use a `for` loop over the `range` of the `args.number`. Since we don't actually need the value from the `range`, we can use the `_` to disregard it.

⑭ Use the `random.sample` function to select the correct number of adjectives and join them on the comma-space `str`.

⑮ Use an f-string to format the output to `print`.

Discussion

Defining the arguments

More than half of my solution is just in defining the program's arguments to `argparse`. The effort is well worth the result, because `argparse` will ensure that each argument is a valid integer value because I set `type=int`. Notice there are no quotes around the `int` — it's not the string `'int'` but a reference to the class in Python.

```
parser.add_argument('-a',                          ①
                    '--adjectives',                ②
                    help='Number of adjectives',   ③
                    metavar='int',                 ④
                    type=int,                      ⑤
                    default=2)                     ⑥
```

① The short flag.

② The long flag.

③ The help message.

④ A type hint for the user, note that this is a `str` value.

⑤ The actual Python `type` for converting the input, note that this is the bareword `int` for the integer class.

⑥ The default value for the number of adjectives per insult.

For `--adjectives` and `--number`, I can set reasonable defaults so that no input is required from the user. This makes your program dynamic, interesting, and testable. How do you know if your values are being used correctly unless you change them and test that the proper change was made in your program? Maybe you started off hardcoding the number of insults and forgot to change the `range` to use a variable. Without changing the input value and testing that the number of insults changed accordingly, it might be a user who discovers your bug, and that's somewhat embarrassing.

Using `parser.error`

I really love the `argparse` module for all the work it saves me. For instance, the `type=int` saves me all the trouble of verifying that the input is an integer, creating an error message, and then converting the user's input to an actual `int` value.

Something else I enjoy about `argparse` is that, if I find there is a problem with an argument, I can use `parser.error` to do four things:

1. Print the short usage of the program to the user
2. Print a specific message about the problem
3. Halt execution of the program
4. Return an error code to the operating system

I can't very easily tell `argparse` that the `--number` should be a positive integer, only that it must be of type `int`. I can, however, inspect the value myself and call `parser.error('message')` if there is a problem. I do all this inside `get_args` so that, by the time I call `args = get_args()` in my `main` function, I know that all the arguments have been validated.

I highly recommend you tuck this tip into your back pocket. It can prove quite handy, saving you loads of time validating user input and generating useful error messages. (And it's really quite likely that the future user of your program will be *you*, and you will really appreciate your efforts!)

Program exit values and STDERR

I would like to highlight the exit value of your program. Under normal circumstances, your program should exit with a value of `0`. In computer science, we often think of `0` as a `False` value, but here it's quite positive. In this instance we should think of it like "zero errors." If you use `sys.exit()` in your code to exit a program prematurely, the default exit value is `0`. If you want to indicate to the operating system or some calling program that your program exited with an error, you should return *any value other than* `0`. The `parser.error` function does this for you automatically.

Additionally, it's common for all error messages to be printed not to `STDOUT` (standard out) but to `STDERR` (standard error). On Unix command line, we can segregate these two output channels using `1` for `STDOUT` and `2` for `STDERR`. Notice how I can use `2>` to redirect `STDERR` to the file called `err` so that nothing appears on `STDOUT`:

```
$ ./abuse.py -a -1 2>err
```

And now we can verify that the expected error messages are in the `err` file:

```
$ cat err
usage: abuse.py [-h] [-a int] [-n int] [-s int]
abuse.py: error: --adjectives "-1" must be > 0
```

If you were to handle all of this yourself, you would need to write something like:

```
if args.adjectives < 1:
    parser.print_usage() ①
    print('--adjectives "{}" must be > 0'.format(args.adjectives), file=sys.stderr) ②
    sys.exit(1) ③
```

① Print the short "usage". You can also `parser.print_help()` to print the more verbose output for `-h`.

② Print the error message to the `sys.stderr` file handle. This is similar to the `sys.stdout` file handle we used in the "Howler" exercise.

③ Exit the program with a value that is not `0` to indicate an error.

As you write more programs in your career, you may eventually start chaining them together. We often call these "pipelines" as the output of one program is "piped" to become the input for the next program. If there is an error in any part of the pipeline, we want the entire operation to stop so that the problems can be fixed. A non-zero return value from any program is a warning flag to halt operations.

Controlling randomness with `random.seed`

Once I'm in `main` and have my arguments, I can control the randomness of the program by calling `random.seed(args.seed)` because:

1. The default value of the `seed` is `None`, and setting `random.seed` to `None` is the same as not setting it at all.
2. The `type` of `args.seed` is `int` which is the proper type for `random.seed`. I do not have to validate the argument further. Negative integers are valid values.

Iterating `for` loops with `range`

To generate some `--number` of insults, I use the `range` function. Because I don't need the number of the insult, I can use the underscore (_) as a throwaway value:

```
>>> num_insults = 2
>>> for _ in range(num_insults):
...     print('An insult!')
...
An insult!
An insult!
```

The underscore is a way to unpack a value and indicate that you do not intend to use it. That is, it's not possible to write this:

```
>>> for in range(num_insults):
  File "<stdin>", line 1
    for in range(num_insults):
```

You have to put *something* after the `for` that looks like a variable. If you put a named variable like `n` and then don't use it in the loop, some tools like `pylint` will detect this as a possible error (and well it could be). The `_` shows that you won't use it, which is good information for your future self, some other user, or external tools to know.

You can use multiple _, e.g., here I can unpack a 3-tuple so as to get the middle value:

```
>>> x = 'Jesus', 'Mary', 'Joseph'
>>> _, name, _ = x
>>> name
'Mary'
```

Constructing the insults

To create my list of adjectives, I used the `str.split` method on a long, multi-line string I created using three quotes:

```
>>> adjectives = """
... bankrupt base caterwauling corrupt cullionly detestable dishonest
... false filthsome filthy foolish foul gross heedless indistinguishable
... infected insatiate irksome lascivious lecherous loathsome lubbery old
... peevish rascaly rotten ruinous scurilous scurvy slanderous
... sodden-witted thin-faced toad-spotted unmannered vile wall-eyed
... """.strip().split()
>>> nouns = """
... Judas Satan ape ass barbermonger beggar block boy braggart butt
... carbuncle coward coxcomb cur dandy degenerate fiend fishmonger fool
... gull harpy jack jolthead knave liar lunatic maw milksop minion
... ratcatcher recreant rogue scold slave swine traitor varlet villain worm
... """.strip().split()
>>> len(adjectives)
36
>>> len(nouns)
39
```

To select some number of adjectives, I chose to use `random.sample` function since I needed more than one:

```
>>> import random
>>> random.sample(adjectives, k=3)
['filthsome', 'cullionly', 'insatiate']
```

For just one randomly selected value, I use `random.choice`:

```
>>> random.choice(nouns)
'boy'
```

To concatenante them together, I need to put `', '` (a comma and a space) between each of the adjectives, and I can use `str.join` for that:

```
>>> adjs = random.sample(adjectives, k=3)
>>> adjs
['thin-faced', 'scurvy', 'sodden-witted']
>>> ', '.join(adjs)
'thin-faced, scurvy, sodden-witted'
```

And feed all this to a format string:

```
>>> adjs = ', '.join(random.sample(adjectives, k=3))
>>> print(f'You {adjs} {random.choice(nouns)}!')
You heedless, thin-faced, gross recreant!
```

And now you have a handy way to make enemies and influence people.

Review

- To indicate a problem to `argparse`, use the `parser.error` function to print a short usage, report the problem, and exit the program with an error value to indicate a problem.
- The `str.split` method is a useful way to create a `list` of string values from a long string.
- The `random.seed` function can be used to make reproducible "random" selections each time a program is run.
- The `random` module's `choice` and `sample` functions are useful for randomly selecting one or several items from a list of choices, respectively.

Going Further

- Read your adjective and nouns from files that are passed as arguments.
- Add tests to verify that the files are processed correctly and new insults are still stinging.