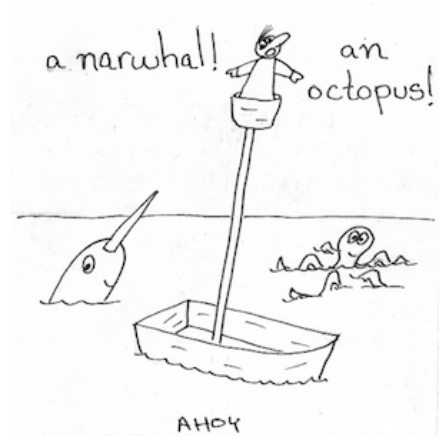


1. The Crow's Nest: Working with strings

Avast, you corny-faced gollumpus! Ye are barrelman for this watch. D'ye ken what I mean, ye addle pated blunderbuss?! Ah, land lubber ye be! OK, then, 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.)



From this point on, I will describe a coding challenge that you should write on your own. I will discuss key ideas you'll need to solve the problems as well as how to use the provided tests to help you know when your program is correct. You should have a copy of the Git repository locally (see the setup instructions). You should write your program in the chapter's directory, like this program should be written in the `02_crowsnest` directory where the tests for the program live.

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

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 with the word "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.

1.1. Getting started

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

1.1.1. 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 run the tests so you we can look at the first failed test:

```
$ cd 02_crownsnest
$ make test
```

Instead of `make test` you can also run `pytest -xv test.py`. Among all the output, you'll notice this line:

```
$ pytest -xv test.py
===== test session starts ===== ①
...
collected 6 items

test.py::test_exists FAILED [ 16%] ②
```

① This the test start of the output from `pytest`.

② This test **FAILED**. There are more tests after this, but testing stops here because of the `-x` flag to `pytest`.

If you read more, you'll see lots of other output all trying to convince you that the expected file, `crownsnest.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.

Let's take a gander at the output. It does look a bit daunting at first, but you'll get used to read the messages and finding what needs to be fixed:

```

===== FAILURES =====
----- test_exists -----

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

> assert os.path.isfile(prg)                      ❷
E     AssertionError: assert False                ❸
E       + where False = <function isfile at 0x1086f1310>('./crowsnest.py')
E       +   where <function isfile at 0x1086f1310> = <module 'posixpath' from
'/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/posixpath.py'>.isfile
E       +   where <module 'posixpath' from
'/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/posixpath.py'> =
os.path

test.py:22: AssertionError
!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures !!!!!!!!!!!!!!!!!!!!!!! ❹
===== 1 failed in 0.05s =====

```

- ❶ This is the actual code inside `test.py` that is running. It's a function called `test_exists`.
- ❷ The `>` at the beginning of this line indicates this is the line where the error starts. The test is checking if there is a file called `crowsnest.py`. If you haven't created it, this will fail as expected.
- ❸ The `E` at the beginning of this line is the "Error" you should read. It's very difficult to understand what the test is trying to tell you, but essentially the `'./crowsnest.py'` file does not exist.
- ❹ A warning that no more tests will run after the one failure.

The first test for every program in the book checks that the expected file exists, so let's create it!

1.1.2. Creating programs with `new.py`

In order to pass this test, we need to create a file called `crowsnest.py` inside the `02_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. From the top level of the repository, you can run this command to create the new program:

```

$ bin/new.py 02_crowsnest/crowsnest.py
Done, see new script "02_crowsnest/crowsnest.py."

```

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

```

$ cp template/template.py 02_crowsnest/crowsnest.py

```

At this point you should have the outline of a working program that accepts command-line arguments. If you run your the 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
```

Run it like `./crowsnest.py --help` and see that it will print a longer help message, too.

NOTE

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

1.1.3. Write, test, repeat

We just created the program, so we ought to be able to pass the first test. If you run your tests again, you will in fact pass the first *two* tests:

1. Does the program exist? Yes, we just created it.
2. Does the program print a help message when we ask for help? Yes, we ran it above with no arguments and the `--help` flag and saw that it will produce help messages.

The cycle I hope you'll develop is to write a very small amount of code — literally one or two lines at most! — and then run the program or the tests to see how you're doing. Let's run our tests again:

```
$ make test
pytest -xv test.py
===== test session starts =====
...
collected 6 items

test.py::test_exists PASSED [ 16%] ①
test.py::test_usage PASSED [ 33%] ②
test.py::test_consonant FAILED [ 50%] ③
```

- ① The expected file exists, so this test passes.
- ② The program will respond to `-h` and `--help`. The fact that the help is actually *incorrect* is not important at this point. At this point, the test are only checking that you seem to have the outline of a program that will run and process the "help" flags.
- ③ The `test_consonant` test is failing. That's OK! We haven't even started writing the actual program, but at least we have a place to start!

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.

1.1.4. Defining your arguments

Here is a diagram sure to shiver your timbers showing the inputs (or *parameters*) 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 output.

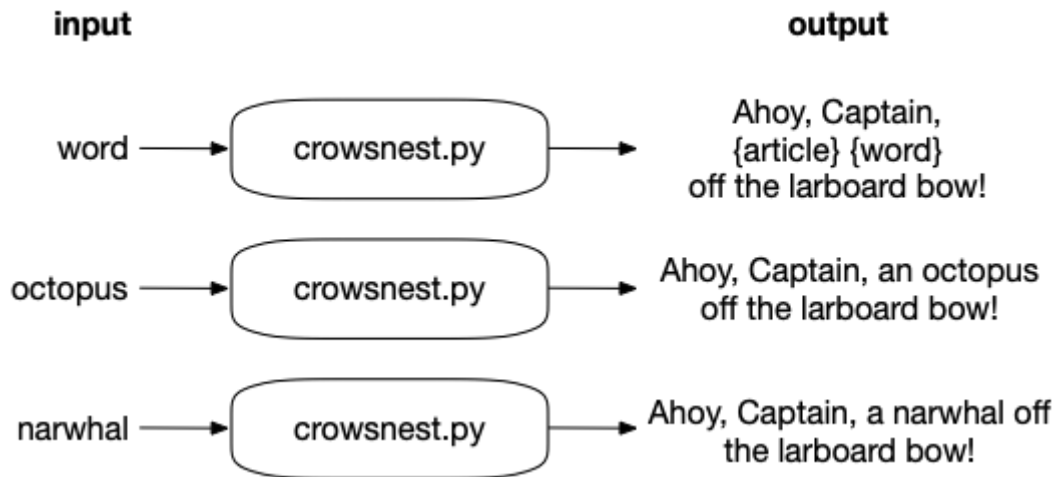


Figure 2. 1. The input to the program is a word, and the output is that word plus its proper article (and some other stuff).

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, and our program needs to take a single, positional argument. If you're unsure what a "positional" argument is, be sure to read the appendix on `argparse`, especially the example "A single, positional argument."

The default `get_args` created by our template names the first argument `'positional'`. Remember that positional arguments are defined by their position and 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] word
crowsnest.py: error: the following arguments are required: word
```

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

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

Crow's Nest -- choose the correct article

positional arguments:
  word          A word ①

optional arguments:
  -h, --help  show this help message and exit ②
```

① You need to define a `word` parameter. Notice that it is listed as a "positional" argument.

② The `-h` and `--help` flags are created automatically by `argparse`. You are not allowed to use these as options. They are used to create the documentation for your program.

Do not proceed until your usage matches the above!

When your program prints the correct usage, you can get the `word` argument inside the `main` function. Modify your program so that it will print the `word`:

```
1 def main():
2     args = get_args()
3     word = args.word
4     print(word)
```

And then test that it works:

```
$ ./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(f'{prg} {word}')
>         assert out.strip() == template.format('a', word)
E         AssertionError: assert 'brigatine' == 'Ahoy, Captai...larboard bow!'
E             - brigatine
E             + Ahoy, Captain, a brigatine off the larboard bow!
```

- ① It's not terribly important right now to understand this line, but the `getoutput` function is running the program with a `word`. We're going to talk about the f-string in this chapter. The output from running the program will go into the `out` variable which will be used to see if the program created the correct output for a given `word`. None the code in this function is anything you should worry about being able to write yet!
- ② The line starting with `>` shows code that produced an error. The output of the program is compared to an expected string. Since it didn't match, the `assert` produces an exception.
- ③ This line starts with `E` to indicate the "error."
- ④ The line starting with a `-` is what the test got when it ran with the argument `'brigatine'` — it got back just the word "brigatine."
- ⑤ 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?

1.1.5. Concatenating strings

Putting strings together is called "concatenating" or "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*. Here's what it looks like on my system:

```
$ python3
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You may also like to use Python's IDLE (integrated development and learning environment) program, `ipython`, or Jupyter Notebooks to interact with the language. I'll stick to the `python3` REPL throughout the book. 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. Remember *not* to type that part! 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
```

WARNING

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?

1.1.6. 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'>
```

WARNING

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.

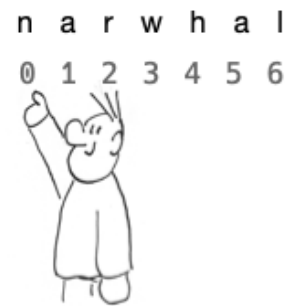

```
>>> word = narwhal
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'narwhal' is not defined
```

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

1.1.7. 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**.

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



Or directly on a string:

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

You can use this with a variable:

n	a	r	w	h	a	l
0	1	2	3	4	5	6

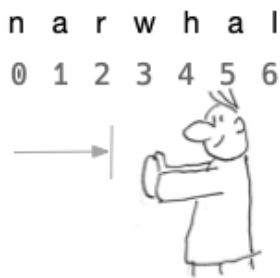
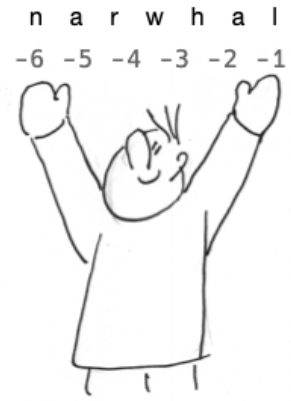


This means that the last index is *one less than the length*, which is often confusing. The length of "narwhal" is 7, but the last character is found at index **6**:

```
>>> 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:]
'hal'
```

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.

1.1.8. 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 the letter `f` (or sometime `CTRL-f`) to jump forward to the next page, and the letter `b` (or sometimes `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").

1.1.9. 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 `str.upper()` function on the variable `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 `str.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 **word.isupper()** returns **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
>>> len(word)
7
```



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

1.1.10. 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'
```

If you check the **type** of our new **char** variable, it is a **str**. Even a single character is still considered by Python to be a "string":

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

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
```

NOTE

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`.

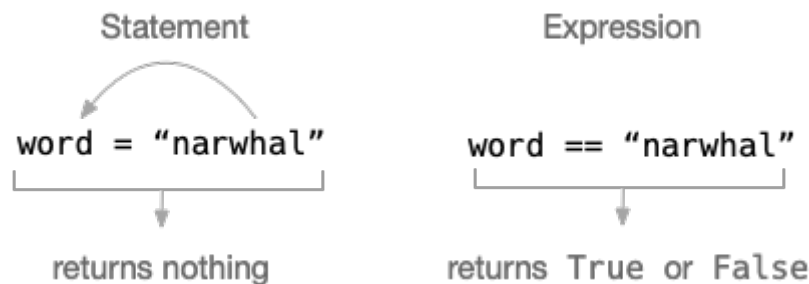


Figure 2. 2. An expression returns a value. A statement does not.

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
```

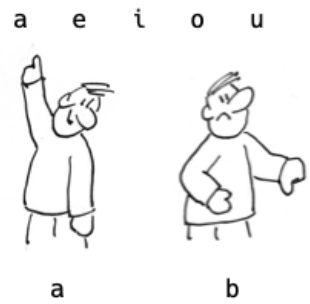
What if the `word` is "Octopus" or "OCTOPUS"?

```
>>> 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 `word[0]`? Remember, that `word[0]` returns a `str`, and so we can chain other `str` methods onto that:

```
>>> word = 'OCTOPUS'
>>> char = word[0].lower()
>>> 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
```

Let's use that to test the first character of the lowercased `word` (which is `'o'`):

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

1.1.11. Conditional branching

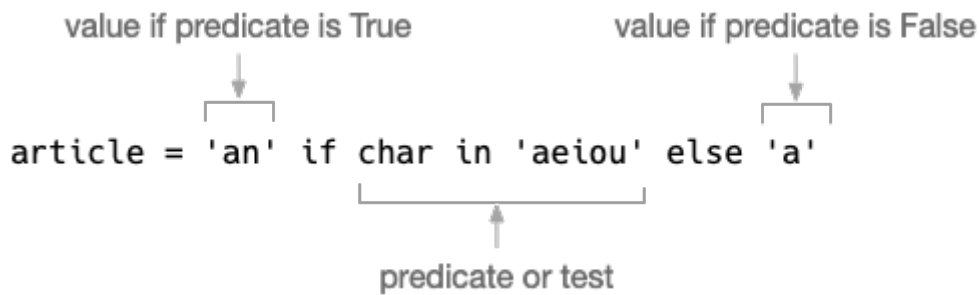
Once you have figured out if the first letter is a vowel, you will need to select an article. We'll use a very simple rule that, if the word starts with a vowel, choose "an," otherwise choose "a." This misses exceptions like when the initial "h" in a word is silent, for instance, we say "a hat" but "an 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 word[0].lower() in 'aeiou':             ②
...     article = 'an'                          ③
... else:                                       ④
...     article = 'a'                          ⑤
... 
```

- ① Initialize `article` to the empty string.
- ② Check if the first, lowercased character of `word` is a vowel.
- ③ If it is, set `article` to `'an'`
- ④ Otherwise,
- ⑤ Set `article` to `'a'`.

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'
```

Let's verify that we have the correct *article*:

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

1.1.12. 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 *str.format()*, and they are substituted in the same order that the {} appear:

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

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."



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 `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! Look alive, you ill-tempered shabaroon!

1.2. 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='word', 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.

1.3. Discussion

I'd like to stress that the preceding 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.

1.3.1. 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:

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

I would really recommend you never leave the "positional" argument named '`positional`' because it is an entirely un-descriptive 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`:

```
1 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. (On the command line, an exit value of `0` means there were 0 errors. Anything other than `0` is considered an error.)

1.3.2. 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'
```

1.3.3. 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):

n a r w h a l
↓
a

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

o c t o p u s
↓
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'
```

1.3.4. 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. While it's a matter of personal taste which you choose, I find f-strings a bit easier to read as my eyes don't have to jump back and forth from the `{}` placeholders to the variables that will go inside them.

1.3.5. 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 Sadow"

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 truly free from 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.

We won't be discussing how to *write* our own tests just yet. That will come later. For now, I've written all the tests for you. I hope that by the end of this book, you will see the value of testing and will always start off by writing *tests first and code second*!

1.4. 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 they come into existence when you assign a value to them.
- Strings have methods like `str.upper()` and `str.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 criteria of correctness. Every feature of a program should have tests, and writing and running test suites should be an integral part of writing programs.

1.5. 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.^[2]). 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.

