# 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, landlubber 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, each chapter will present a coding challenge that you should complete on your own. I will discuss the key ideas you'll need to solve the problems as well as how to use the provided tests to determine when your program is correct. You should have a copy of the Git repository locally (see the setup instructions in the book's introduction), and you should write each program in that chapter's directory. For example, this chapter's program should be written in the 02_crowsnest directory, where the tests for the program live.
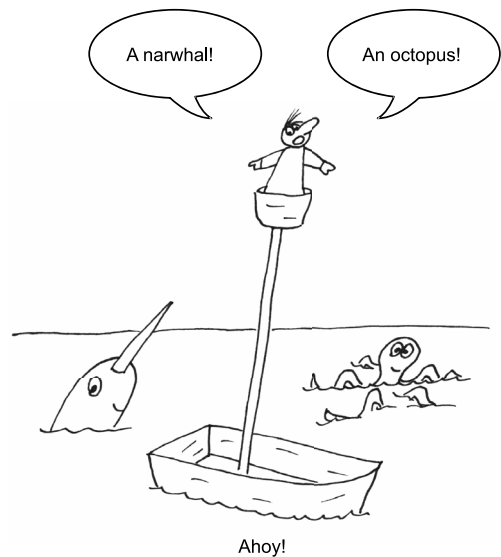
In this chapter, we're going to start working with strings. By the end, you will be able to

- 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 you'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 string that puts those two values into the "Ahoy" phrase.

2.1 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 you can use the tests to know when your program is working and how you might get started programming.

2.1.1 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 to run the tests so you can look at the first failed test:

```
$ cd 02_crowsnest
$ make test
```

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

```
$ pytest -xv test.py
============================ test session starts =============================
...
```

```
collected 6 items

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

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

You'll also see lots of other output 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, 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 at bit daunting at first, but you'll get used to reading the messages and finding your errors.

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.

```
=================================== FAILURES ===================================
_____ test_exists _____

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

>       assert os.path.isfile(prg)
E       AssertionError: assert False
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 !!!!!!!!!!!!!!!!!!!!!!!!!!!!
============================== 1 failed in 0.05s ===============================
```

This is the actual code inside test.py that is running. It's a function called test_exists().

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.

This warns that no more tests will run after the one failure. This is because we ran it with the flag to stop testing at the first failure.

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

2.1.2 Creating programs with new.py

In order to pass the first test, you 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 the following 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 can copy the template/template.py program:

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

You should now have the outline of a working 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
```

Run it with ./crowsnest.py --help. It will print a longer help message too.

> **NOTE**   Those are not the correct parameters for our program, just the default examples supplied by new.py. You will need to modify them to suit this program.

2.1.3   Write, test, repeat

You just created the program, so you ought to be able to pass the first test. 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 the tests again:

The program will respond to -h and --help. The fact that the help is actually incorrect is not important at this point. The tests are only checking that you seem to have the outline of a program that will run and process the help flags.

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

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

The expected file exists, so this test passes.

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.

As you can see, creating a new program with new.py will make you pass the first *two* tests:

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

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

2.1.4 Defining your arguments

Figure 2.1 is sure to shiver your timbers, showing the inputs (or *parameters*) and output of the program. We'll use these diagrams throughout the book to imagine how code and data work together. In this program, the input is a word, and a phrase incorporating that word with the correct article is the output.
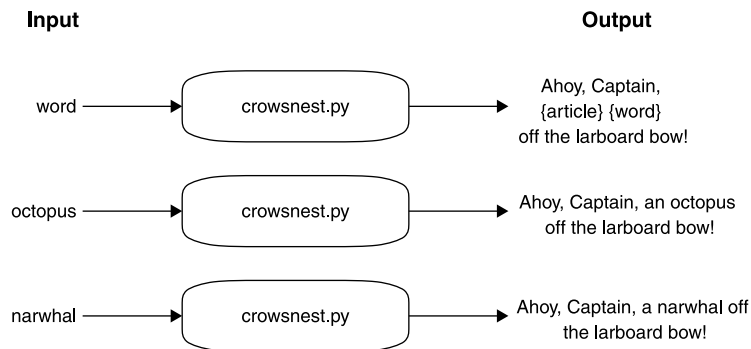
Input   **Output**

```
word ──────▶  crowsnest.py  ──────▶   Ahoy, Captain,
                                      {article} {word}
                                      off the larboard bow!

octopus ───▶  crowsnest.py  ──────▶   Ahoy, Captain, an octopus
                                      off the larboard bow!

narwhal ───▶  crowsnest.py  ──────▶   Ahoy, Captain, a narwhal off
                                      the larboard bow!
```

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 accept a single, positional argument. If you're unsure what a "positional" argument is, be sure to read the appendix, especially section A.4.1.

The `get_args()` function created by the template names the first argument `positional`. Remember that positional arguments are defined by their positions 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 longer usage documentation 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

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

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

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 preceding!

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`:

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

Then test that it works:

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

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

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 of the code in this function is anything you should worry about being able to write yet.

```
=================================== FAILURES ===================================
_____ test_consonant _____


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

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

The line starting with ">" shows the 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.

This line starts with "E" to indicate the error.

The line starting with the plus sign (+) is what the test expected: "Ahoy, Captain, a brigantine off the larboard bow!"

The line starting with a hyphen (-) is what the test got when it ran with the argument "brigantine"—it got back the word "brigantine."

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

2.1.5   *Concatenating strings*

Putting strings together is called *concatenating* or *joining* strings. To demonstrate, I'll 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 REPL is a Read-Evaluate-Print-Loop—Python will *read* each line of input, *evaluate* it, 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.
>>>
```

The ">>>" is a prompt where you can type code. Remember *not* to type that part! To exit the REPL, either type `quit()` or press Ctrl-D (the Control key plus the letter *D*).

> **NOTE**   You may prefer 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.

Let's start off by assigning the variable `word` to the value "narwhal." In the REPL, type `word = 'narwhal'` and press 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 and Flake8 that help you find errors in your code) ask you to use exactly one space on either side.

If you type `word` and press Enter, Python will print the current value of `word`:

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

Now type `werd` and press 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 `werd` for you when you assign it a value.

We need to insert the `word` between two other 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 string instead of just printing the word, you should be able to pass 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 you 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             ?                  +
```

We hardcoded the "a" before the word, but we really need to figure out whether to use "a" or "an" depending on whether the word starts with a vowel. How can we do that?

2.1.6   *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 it holds. Because we put the value for word in quotes ('narwhal'), 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 you what kind of data Python thinks something is:

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

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

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

> **WARNING**   If you forget the quotes, 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.

2.1.7 *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 can 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 = 'narwhal'
>>> word[0]
'n'
```

You can index into a literal string value too:

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

Because the index values start with 0, that means the last index is *one less than the length* of the string, 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 index numbers to count backwards from the end, so the last index is also -1:

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

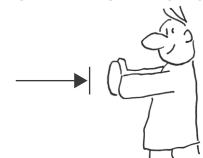You can use *slice notation* [start:stop] to get a range of characters. Both start and stop are optional. The default value for start is 0 (the beginning of the string), and the stop value is *not inclusive*:

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

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

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

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

2.1.8   Finding help in the REPL

The `str` class 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 (and discover) 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, in this case 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 sometimes 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").

2.1.9   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 will see that there is a function called `str.upper()`. Here is how you can *call* or *execute* that function:

```
>>> 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 the `str.upper()` function on the variable `word`, so we add the parentheses. Note that the function returns an uppercase version of the word but *does not* change the value of `word`:

```
>>> 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 `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 must 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.

2.1.10 *String comparisons*

You now 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 your 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 the 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
> pronounce "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 (see figure 2.2).

Statement   **Expression**

`word = "narwhal"`   `word == "narwhal"`
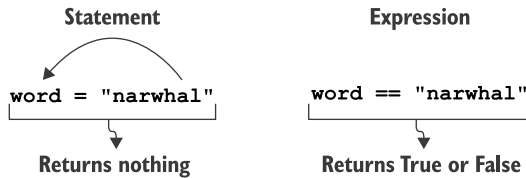
Returns nothing   **Returns True or False**

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 com-
parisons, 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, 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, which will tell us if the value x is in
the collection y. We can ask whether the letter 'a' is in the
longer string 'aeiou':

```
>>> 'a' in 'aeiou'
True
```
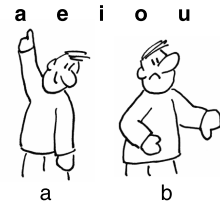
But the letter `'b'` is not:

```
>>> '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
```

2.1.11 *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: 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 the case where 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 we'll use an `if/else` statement to figure out what to put in it:

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

Initialize article to the empty string.

Check if the first, lowercased character of word is a vowel.

Set article to "an" if the first character is a vowel.

Set article to "a" if the first character is not a vowel.

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, and then the value if the predicate is `False` (figure 2.3).

Value if predicate is True   **Value if predicate is False**

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

Predicate or test

Figure 2.3   The `if` expression will return the first value if the predicate is `True` and the second value otherwise.

This approach 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:

```
>>> char = 'o'
>>> article = 'an' if char in 'aeiou' else 'a'
```
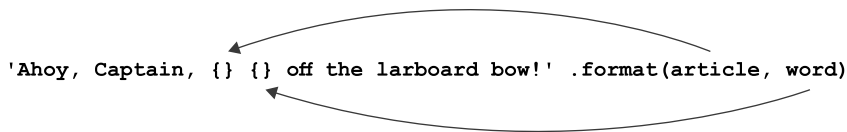
Let's verify that we have the correct `article`:

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

2.1.12 String formatting

Now we have two variables, `article` and `word`, that need to be incorporated into our "Ahoy!" phrase. You saw earlier that we can use the plus sign (+) to concatenate strings. Another method for creating new strings from other strings is to use the `str.format()` method.

To do so, you create a string template with curly brackets {}, which indicate placeholders for values. The values that will be substituted are arguments to the `str.format()` method, and they are substituted in the same order that the {} appear (figure 2.4).

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

Figure 2.4   The `str.format()` method is used to expand the values of variables inside strings.

Here it is in code:

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

Another method for combining strings uses the special "f-string" where you can put the variables directly into the curly brackets {}. It's a matter of taste which approach you choose; I tend to prefer this style because I don't have to think about which variable goes with which set of brackets:

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

NOTE 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, it can never hold a different type of value, like a string. This is called *static typing* because the type of the variable can never change.

Python is a *dynamically typed* language, which means 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."

2.1.13 *Time to write*

Here are a few hints for writing your solution:

- Start your program with new.py and fill in `get_args()` with a single positional 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 whether 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 whether 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`, with the collection here being a `list`.
- Use `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 that your program compiles and is on the right track.

Now go write the program before you turn the page and study my solution. Look alive, you ill-tempered shabaroon!

2.2 Solution

Following is one way you could write a program that satisfies the test suite:

```
#!/usr/bin/env python3
"""Crow's Nest"""

import argparse
```

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

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

    parser = argparse.ArgumentParser(
        description="Crow's Nest -- choose the correct article",
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('word', metavar='word', help='A word')

    return parser.parse_args()

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

    args = get_args()
    word = args.word
```

The parser will parse the arguments.

Show the default values for each parameter in the usage.

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

Define a positional argument called word.

The result of parsing the arguments will be returned to main().

args contains the return value from the get_args() function.

Define the main() function where the program will start.

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

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

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


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

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

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

If we are in the "main" namespace, call the main() function to make the program start.

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.

2.3   Discussion

I'd like to stress that the preceding listing 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:

- `get_args()`, where I define the arguments to the program
- `main()`, where the program starts

Let's talk about these two functions.

2.3.1   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 define and validate 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='word', help='Word')
```

I recommend you never leave the positional argument named `'positional'` because it is an entirely nondescriptive term. Naming your variables according to *what they are* will make your code more readable.

The program doesn't need any of the other options created by new.py, so 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 that I put into the variable args:

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

If `argparse` is not able to parse the arguments—for example, if 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.)

2.3.2   *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, but it's an extremely common idiom in Python. Every program I present will start with a `main()` function that will first call `get_args()` to get the program's inputs:

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

I can now access the `word` by calling `args.word`. Note the lack of parentheses. It's not `args.word()` because it is not a function call. Think of `args.word` as being 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'
```

2.3.3   *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`. 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 the `word.lower()` function. Then 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 list of vowels):

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

You can use membership in the list of vowels as a condition to choose "an"; otherwise, we choose "a." As mentioned in the introduction, the `if` expression is the shortest and safest way to make a *binary* choice (where there are only two possibilities):

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

The safety of the `if` expression comes from the fact that Python will not even run this program if you forget the `else`. Try it and see what error you get.

Let's change the value of `word` to "galleon" and check that it still works:

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

2.3.4   Printing the results

Finally we need to print out the phrase with our article and word. As noted in the introduction, you can use the `str.format()` function to incorporate the variables into a string:

```
>>> 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 choose 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.

2.3.5   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 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 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*:

- Write tests *before* we write the software.
- Run the tests to verify that our as-yet-unwritten software fails to deliver on some task.
- Write the software to fulfill the request.
- Run the tests to check that it now *does* work.
- 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* 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*!

2.4   Going further

- Have your program match the case of the incoming word (for example, "an octopus" and "An Octopus"). Copy an existing `test_` function in test.py to verify that your program works correctly while still passing all the other tests. Try writing the test first, and 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 rudder, which typically was on the right side of the boat for right-handed sailors.

Summary

- All Python's documentation is available at https://docs.python.org/3/ and via 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 arguments.
- F-strings like `f'{article} {word}'` allow variables and code to go directly inside the brackets.
- The `x in y` expression will report whether the value `x` is present in the collection `y`.
- Statements like `if/else` do not return a value, whereas 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.