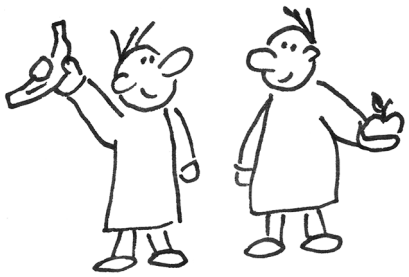
8 Apples and Bananas: Find and replace

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 a 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's consider the children's song "Apples and Bananas," wherein we intone 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" sound (as in "hay"):

```
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 that takes some text, given as a single positional argument, and replaces all the vowels in the text with the given `-v` or `--vowel` options (with the default being `a`).

The program should be written in the 08_apples_and_bananas directory and should handle text on the command line:

```
$ ./apples.py foo
faa
```

And accept the `-v` or `--vowel` option:

```
$ ./apples.py foo -v i
fii
```

Your program should *preserve the case* of the input vowels:

```
$ ./apples.py -v i "APPLES AND BANANAS"
IPPLIS IND BININIS
```

As with the Howler program in chapter 5, 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.
```

Figure 8.1 shows a diagram of the program's inputs and output.

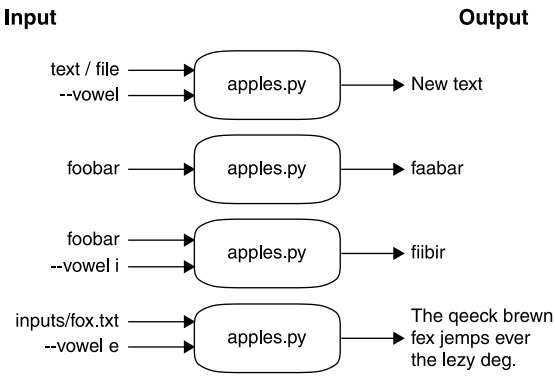


Figure 8.1 Our program will accept some text and possibly a vowel. All the vowels in the given text will be changed to the same vowel, resulting in hilarity.

Here is the usage statement that should print when there are no arguments:

```
$ ./apples.py
usage: apples.py [-h] [-v vowel] text
apples.py: error: the following arguments are required: text
```

And the program should always print usage for the `-h` and `--help` flags:

```
$ ./apples.py  -h
usage: apples.py [-h] [-v vowel] text
```

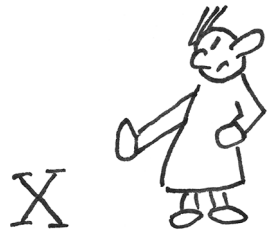
```
Apples and bananas

positional arguments:
  text                  Input text or file

optional arguments:
  -h, --help            show this help message and exit
  -v vowel, --vowel vowel
                        The vowel to substitute (default: a)
```

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')
```



Your program is going to need to do the following:

- 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`

8.1 Altering strings

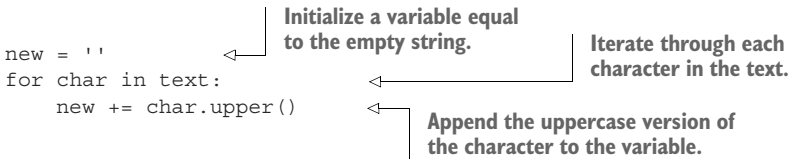
So far in our discussions of Python strings, numbers, lists, and dictionaries, we've seen how easily we can change or *mutate* variables. There is a problem, however, in that *strings are immutable*. Suppose we have a `text` variable that holds our input text:

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
```

If we wanted to turn the first "e" (at index 2) 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 chapter 4 you saw that you can use a `for` loop to iterate over the characters in a string. For instance, I could laboriously uppercase `text` like so:



We can inspect the value of `new` 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 the character is a vowel, you could change it for the given `vowel`; otherwise, you could use the character itself. We had to identify vowels in chapter 2, so you can refer back to how you did that.

8.1.1 Using the str.replace() method

In chapter 4 we talked about using the `str.replace()` method to replace all the numbers in a string with a different number. Maybe that would be a good way to solve this problem? Let's look at the documentation for that using `help(str.replace)` in the REPL:

```
>>> 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 give that a try. We could replace "T" with "X":

```
>>> text.replace('T', 'X')
'Xhe quick brown fox jumps over the lazy dog.'
```

This seems promising! Can you see a way to replace all the vowels using this idea? Remember that this method never mutates the given string but instead returns a new string that you will need to assign to a variable.

8.1.2 Using str.translate()

We also looked at the `str.translate()` method in chapter 4. There we created a dictionary that described how to turn one character, like "1," into another string like "9." Any character not mentioned in the dictionary was left alone.

The documentation for this method is a bit more cryptic:

```
>>> 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.
```

In my solution, I created the following dictionary:

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

That is the argument to the `str.maketrans()` function, which creates a translation table that is then used with `str.translate()` to change all the characters present as keys in the dictionary to their corresponding values:

```
>>> '876-5309'.translate(str.maketrans(jumper))
'234-0751'
```

What keys and values should you have in a dictionary if you want to change all the vowels, both lower- and uppercase, to some other value?

8.1.3 Other ways to mutate strings

If you know about regular expressions, that's a strong solution. If you haven't heard of them, don't worry—I'll introduce them in the discussion.

The point is for you to *play* with this and come up with a solution. I found eight 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?

Here are a few hints:

- Consider using the `choices` option in the `argparse` documentation to constrain the `--vowel` options. Be sure to read section A.4.3 in the appendix for an example.
- Be sure to change both lower- and uppercase versions of the vowels, 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 solution.

8.2 Solution

Here is the first solution I wanted to share. After this, we'll explore several more.

```
#!/usr/bin/env python3
"""Apples and Bananas"""

import argparse
import os


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

    parser = argparse.ArgumentParser(
        description='Apples and bananas',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text', help='Input text or file')

    parser.add_argument('-v',
                        '--vowel',
                        help='The vowel(s) allowed',
                        metavar='vowel',
                        type=str,
                        default='a',
                        choices=list('aeiou'))

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read().rstrip()

    return args


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

    args = get_args()
    text = args.text
    vowel = args.vowel
    new_text = []

    for char in text:
        if char in 'aeiou':
            new_text.append(vowel)
        elif char in 'AEIOU':
            new_text.append(vowel.upper())
        else:
            new_text.append(char)
```

The input might be text or a filename, so I defined it as a string.

Use "choices" to restrict the user to one of the listed vowels.

Check if the text argument is a file.

If it is, read the file using str.rstrip() to remove any trailing whitespace.

Create a new list to hold the characters for the transformed text.

Iterate through each character of the text.

Check if the current character is in the list of lowercase vowels.

If it is, use the vowel value instead of the character.

Check if the current character is in the list of uppercase vowels.

If it is, use the value of vowel.upper() instead of the character.

Otherwise, use the character itself.

```
    print(''.join(new_text))

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

Print a new string made by joining the new text list on the empty string.

8.3 Discussion

I came up with eight ways to write my solution. All of them start with the same `get_args()` function, so let's look at that first.

8.3.1 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 filename:

```
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 substitute'',
                    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 much easier to type `list('aeiou')` and have Python turn the `str` "aeiou" into a `list` of the characters. Both approaches produce the same results, because `list(str)` creates a `list` of the individual characters in a given string. And remember, the use of single or double quotes doesn't matter. Any value enclosed in either type of quotes is a `str`, even if it's just one character:

```
>>> ['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. The absence of any error means that it's OK:

```
>>> assert ['a', 'e', 'i', 'o', 'u'] == list('aeiou')
```

The next task is detecting whether `text` is the name of a file that should be read for the text, or if it is the text itself. This is the same code I used in chapter 5, and again I

chose to handle the `text` argument inside the `get_args()` function so that, by the time I get `text` inside `main()`, it's all been handled. Figure 8.2 illustrates how we can chain the `open()` function to the `read()` method of a file handle to the `rstrip()` method of a string.

```
if os.path.isfile(args.text):
    args.text = open(args.text).read().rstrip()
```

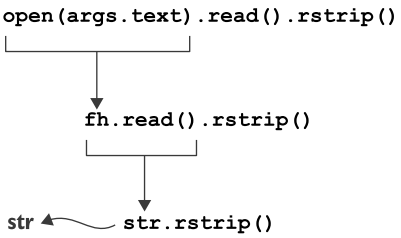


Figure 8.2 We can chain methods together to create pipelines of operations. The open() returns a file handle that we can read. The `read()` operation returns a string that we strip of whitespace.

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` value is one of the allowed characters. To me, this code is a single "unit" where I've handled the arguments. Processing can now go forward by returning the arguments:

```
return args
```

8.3.2 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 tests, 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.*
>
> www.python.org/dev/peps/pep-0020/

But I really come from the Perl mentality, where "There Is More Than One Way To Do It" (TIMTOWTDI or "Tim Toady").

METHOD 1: ITERATING THROUGH EVERY CHARACTER

The first method is similar to what we did in chapter 4, where we used a `for` loop on a string to access each character. Here is some code you can copy and paste into the `ipython` REPL:

Set the new_text variable to an empty list.

Set text to the string "Apples and Bananas!"

Set the vowel variable to the string "o". That is, we'll replace all the vowels with this one.

Use a for to iterate text, putting each character into the char variable.

```
>>> text = 'Apples and Bananas!'
>>> vowel = 'o'
>>> new_text = []
>>> for char in text:
```

```
...     if char in 'aeiou':
...         new_text.append(vowel)
...     elif char in 'AEIOU':
...         new_text.append(vowel.upper())
...     else:
...         new_text.append(char)
...
>>> text = ''.join(new_text)
>>> text
'Opplos ond Bononos!'
```

If the character is in the set of lowercase vowels, add the vowel "o" to the new text.

If the character is in the set of uppercase vowels, substitute the vowel.upper() version "O" into the new text.

Otherwise, add the current character to the new text.

Turn the new_text list into a 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. With that approach, you wouldn't have to `str.join()` them at the end. Whatever you prefer:

```
new_text += vowel
```

Next I'm going to show you several alternate solutions. They're all functionally equivalent because they all pass the tests—the point here is to explore the Python language and understand it. For the alternate solutions, I'll just show the `main()` function.

METHOD 2: USING THE STR.REPLACE() METHOD

Here is a way to solve the problem using the `str.replace()` method:

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel

    for v in 'aeiou':
        text = text.replace(v, vowel).replace(v.upper(), vowel.upper())

    print(text)
```

Iterate through the list of vowels. We don't have to say list('aeiou') here—Python will automatically treat the string 'aeiou' like a list because we are using it in a list context with the for loop.

Use the str.replace() method twice to replace both the lower- and uppercase versions of the vowel in the text.

Earlier in the chapter, I mentioned the `str.replace()` method, which 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 `str.replace()` methods. It could be written as two separate statements, as illustrated in figure 8.3.

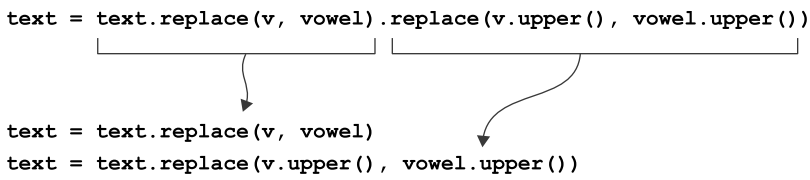


Figure 8.3 The chained calls to `str.replace()` can be written as two separate statements if you prefer.

METHOD 3: USING THE STR.TRANSLATE() METHOD

Can we use the `str.translate()` method to solve this? I showed in chapter 4 how you could use a dictionary called `jumper` to change a character like "1" to the character "9." In this problem, we need to change all the lower- and uppercase vowels (10 total) to some given `vowel`. For instance, to change all the vowels into the letter "o," we could create a translation table `t` like so:

```
t = {'a': 'o',
     'e': 'o',
     'i': 'o',
     'o': 'o',
     'u': 'o',
     'A': 'O',
     'E': 'O',
     'I': 'O',
     'O': 'O',
     'U': 'O'}
```

We could use `t` with the `str.translate()` method:

```
>>> 'Apples and Bananas'.translate(str.maketrans(t))
'Opplos ond Bononos'
```

If you read the documentation for `str.maketrans()`, you will find that another way to specify the translation table is to supply two strings of equal lengths:

```
maketrans(x, y=None, z=None, /)
    Return a translation table usable for str.translate().

    If there is only one argument, it must be a dictionary mapping Unicode
    ordinals (integers) or characters to Unicode ordinals, strings or None.
    Character keys will be then converted to ordinals.
    If there are two arguments, they must be strings of equal length, and
    in the resulting dictionary, each character in x will be mapped to the
    character at the same position in y. If there is a third argument, it
    must be a string, whose characters will be mapped to None in the result.
```

The first string should contain the letters you want to replace, which are the lower- and uppercase vowels `'aeiouAEIOU'`. The second string is composed of the letters to use for substitution. We want to use `'ooooo'` for `'aeiou'` and `'OOOOO'` for `'AEIOU'`.

We can repeat `vowel` five times using the `*` operator that you'll normally associate with numeric multiplication. This is (sort of) "multiplying" a string, so, OK, I guess:

```
>>> vowel * 5
'ooooo'
```

Next we handle the uppercase version:

```
>>> vowel * 5 + vowel.upper() * 5
'oooooOOOOO'
```

And now we can make the translation table in one line of code like this:

```
>>> trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5)
```

Let's inspect the `trans` table. We'll use the `pprint.pprint()` (pretty-print) function so we can read it easily:

```
>>> 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 its *ordinal* value, which is the character's position in the ASCII table (www.asciitable.com).

You can go back and forth between characters and their ordinal values by using the `chr()` and `ord()` functions. We will explore and use these functions later in chapter 18. 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
```

You can create the same output by 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 haven't included the output because there are 100 printable characters:

```
>>> print(len(string.printable))
100
```

So the `trans` table is a mapping from one character to another, just like in the "Jump the Five" exercise in chapter 4. The lowercase vowels ("aeiou") all map to the ordinal value `111`, which is "o." The uppercase vowels ("AEIOU") map to 79, which is "O." You can use the `dict.items()` method to iterate over the key/value pairs of `trans` to verify that 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 `str.translate()` method, so we can overwrite `text` with the new version. Here's how I wrote that idea in my solution:

```
def main():
    args = get_args()
    vowel = args.vowel
    trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5)
    text = args.text.translate(trans)
    print(text)
```

Create a translation table from each of the vowels, both lower- and uppercase, to their respective characters. The lowercase vowels will be matched to the lowercase vowel argument, and the uppercase vowels will be matched to the uppercase vowel argument.

Call the str.translate() method on the text variable, passing the translation table as an argument.

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: USING A LIST COMPREHENSION

Following up on method 1, we can use a *list comprehension* to significantly shorten the `for` loop. In chapter 7 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`:

```
def main():
    args = get_args()
    vowel = args.vowel
    text = [
        vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c
        for c in args.text
    ]
    print(''.join(text))
```

Use a list comprehension to process all the characters in args.text to create a new list called text.

Use a compound if expression to handle three cases: lowercase vowel, uppercase vowel, and the default.

Print the translated string by joining the text list on the empty string.

Let's talk just a bit more about list comprehensions. As an example, we can generate a list of the squared values of the numbers 1 through 4 by using the `range()` function to get the numbers from a starting number to an ending number (not inclusive). In the REPL, we must use the `list()` function to force the production of the values, but usually your code won't need to do this:

```
>>> list(range(1, 5))
[1, 2, 3, 4]
```

> **NOTE** `range()` is another example of a *lazy* function in Python, which means it won't actually produce values until your program needs them—a lazy function is a promise to do something. If your program branches in such a way that you never need to produce the values, the work is never done, meaning your code is more efficient.

We can write a `for` loop to `print()` the squares:

```
>>> for num in range(1, 5):
...     print(num ** 2)
...
1
4
9
16
```

Instead of printing the values, imagine that we wanted to create a new `list` that contains those values. One way to do this would be to create an empty `list` and then use `list.append()` to add each value in a `for` loop:

```
>>> squares = []
>>> for num in range(1, 5):
...     squares.append(num ** 2)
```

Now we can verify that we have our squares:

```
>>> assert len(squares) == 4
>>> assert squares == [1, 4, 9, 16]
```

We can achieve the same result in fewer lines of code using a list comprehension to generate our new `list`, as shown in figure 8.4.

```
>>> [num ** 2 for num in range(1, 5)]
[1, 4, 9, 16]
```

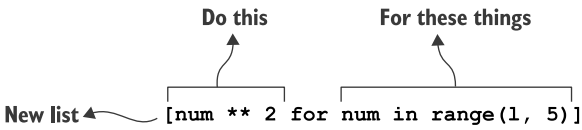


Figure 8.4 A list comprehension creates a new list using a `for` loop to iterate over the source values.

We can assign this list to the variable `squares` and verify that we still have what we 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 this version of the program, we'll condense the `if/elif/else` logic from method 1 into a compound `if` expression. First let's see how we could shorten the `for` loop version:

```
>>> text = 'Apples and Bananas!'
>>> new = []
>>> for c in text:
...     new.append(vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU'
     else c)
...
>>> ''.join(new)
'Opplos ond Bononos!'
```

Figure 8.5 shows how the parts of the expression match up to the original `if/elif/else`:

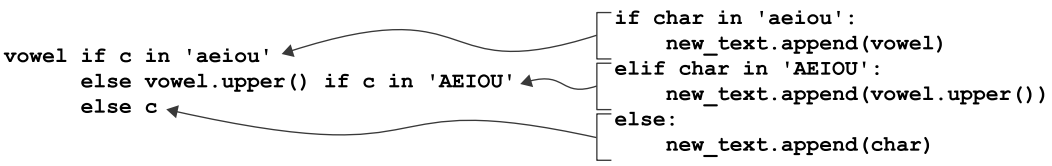


Figure 8.5 The three conditional branches can be written using two `if` expressions.

Now let's 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
...     for c in text ]
...
>>> ''.join(new_text)
'Opplos ond Bononos!'
```

Select the character using the compound if expression.

Perform this action for each character in the text.

The code is denser than the previous `for` loop, but it has advantages in that

- The list comprehension is shorter and generates our list rather than using the side effects of `list.append()`.
- The compound `if` expression will not compile if we forget one of the conditional branches.

METHOD 5: USING A LIST COMPREHENSION WITH A FUNCTION

The compound `if` expression inside the list comprehension is complicated enough that it probably should be a function. We can *define* a new function with the `def` statement and call it `new_char()`. It accepts a character we'll call `c`. After that, we can use the same compound `if` expression as before:

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

    def new_char(c):
        return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c

    text = ''.join([new_char(c) for c in args.text])

    print(text)
```

Define a function to choose a new character. Note that it uses the vowel variable because the function has been declared in the same scope. This is called a closure, because new_char() closes over the variable.

Use the compound if expression to select the correct character.

Use a list comprehension to process all the characters in text.

You can play with the `new_char()` function by putting this into your REPL:

```
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'
```

It should return "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 the `new_char()` function to process all the characters in `text`, using a list comprehension:

```
>>> text = 'Apples and Bananas!'
>>> text = ''.join([new_char(c) for c in text])
>>> text
'Opplos ond Bononos!'
```

Note 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. I've done this because we want to reference the `vowel` variable inside the function without passing it as an argument.

As an example, let's define a `foo()` function that has a `bar()` function inside it. We can call `foo()`, and it will call `bar()`. But from outside of `foo()`, the `bar()` function does not exist (it "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 declared the `new_char()` function inside `main()` because I wanted to reference the `vowel` variable inside the function, as shown in figure 8.6. Because `new_char()` "closes" around the `vowel`, it is a special type of function called a *closure*.

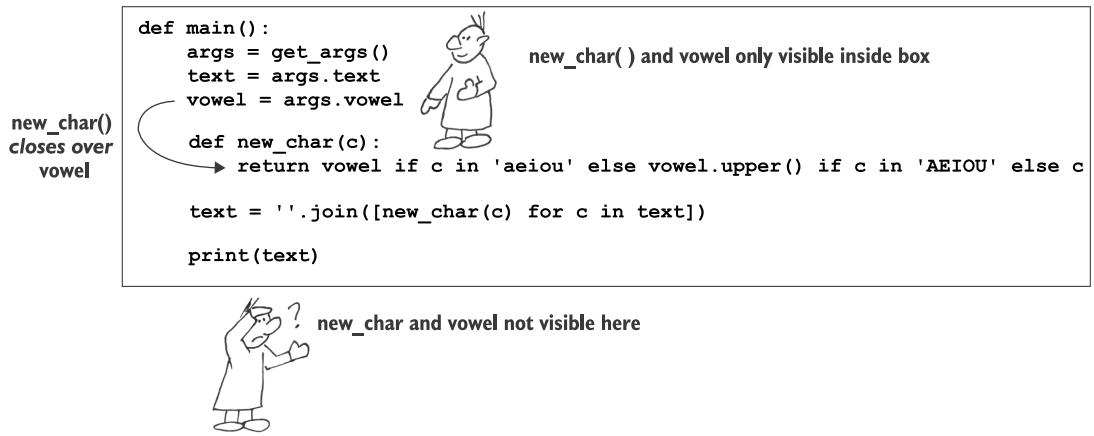


Figure 8.6 The `new_char()` function can only be seen within the `main()` function. It creates a closure because it references the `vowel` variable. Code outside of `main()` cannot see or call `new_char()`.

If we don't write this as a closure, we will have to pass the `vowel` as an argument:

```
def main():
    args = get_args()
    print(''.join([new_char(c, args.vowel) for c in args.text]))


def new_char(char, vowel):
    return vowel if char in 'aeiou' else \
        vowel.upper() if char in 'AEIOU' else char
```

We need to pass args.vowel as an argument to the new_char() function.

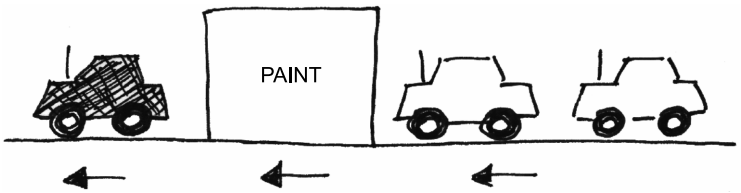
The vowel is only visible inside the main() function. Since new_char() is no longer declared in the same scope, we need to accept vowel as an argument.

While the closure method is interesting, this version is arguably easier to understand. It would also be easier to write a unit test for it, which is something we'll start doing soon.

METHOD 6: USING THE MAP() FUNCTION

For this method, I'll introduce the `map()` function, as it's quite similar to a list comprehension. The `map()` function accepts two arguments:

- A function
- An iterable like a list, a lazy function, or a generator



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.

We can create a function to "paint" 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 first argument you see here starts with the keyword `lambda`, which is used to create an *anonymous* function. With the regular `def` keyword, the function name follows. With `lambda`, there is no name, only the list of parameters and the function body.

For example, an `add1()` function that adds `1` to a value is a regular named function:

```
def add1(n):
    return n + 1
```

It works as expected:

```
>>> assert add1(10) == 11
>>> assert add1(add1(10)) == 12
```

Compare the preceding definition to one created using `lambda`, which we assign to the variable `add1`:

```
>>> add1 = lambda n: n + 1
```

This definition of `add1` is functionally equivalent to the first version. We call it just like the `add1()` function:

```
>>> assert add1(10) == 11
>>> assert add1(add1(10)) == 12
```

The body for a `lambda` is a brief (usually one-line) expression. There is no `return` statement because the final evaluation of the expression is returned automatically. In figure 8.7, you can see that the `lambda` will return the result of `n + 1`.

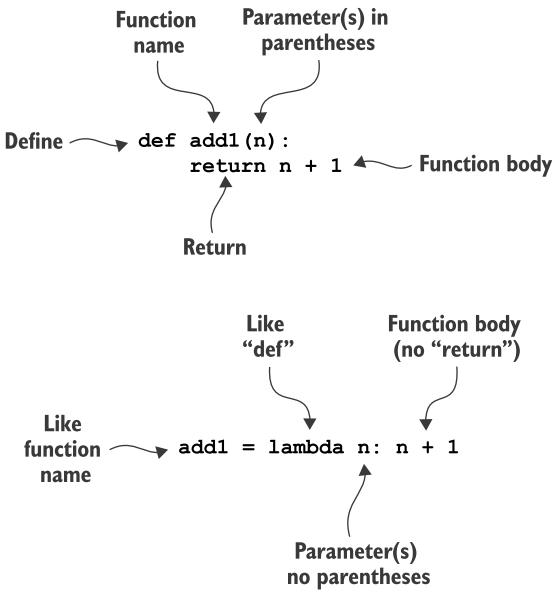


Figure 8.7 Both `def` and `lambda` are used to create functions.

In both versions of the `add1` definition, using `def` and `lambda`, the argument to the function is `n`. In the usual named function, `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 no parentheses around the function's parameter, `n`.

There is no difference in how you can use the two types of functions. They are both functions:

```
>>> type(lambda x: x)
<class 'function'>
```

If you are comfortable with using `add1()` in a list comprehension, like this,

```
>>> [add1(n) for n in [1, 2, 3]]
[2, 3, 4]
```

it's a short step to using the `map()` function.

The `map()` function is a lazy function, like the `range()` function we looked at earlier. It won't create the values until you actually need them, as compared to a list comprehension, which will produce the resulting `list` immediately. I don't personally tend to worry about the performance of the code as much as I do the readability. When I write code for myself, I prefer to use `map()`, but you should write code that makes the most sense for you and your teammates.

To force the results from `map()` in the REPL, we need to use the `list()` function:

```
>>> list(map(add1, [1, 2, 3]))
[2, 3, 4]
```

We can write the list comprehension with the `add1()` code in line:

```
>>> [n + 1 for n in [1, 2, 3]]
[2, 3, 4]
```

That looks very similar to the `lambda` code (as illustrated in figure 8.8):

```
>>> list(map(lambda n: n + 1, [1, 2, 3]))
[2, 3, 4]
```

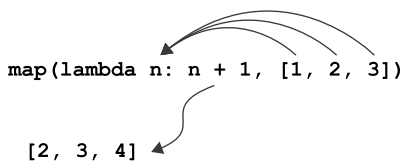


Figure 8.8 The `map()` function will create a new list from processing each element of an iterable through a given function.

Here is how we could use `map()`:

```
def main():
    args = get_args()
    vowel = args.vowel
    text = map(
        lambda c: vowel if c in 'aeiou' else vowel.upper()
        if c in 'AEIOU' else c, args.text)

    print(''.join(text))
```

The map() function wants a function for the first argument and an iterable for the second.

Use lambda to create an anonymous function that accepts a character, c.

args.text is the second argument to map(). Technically, args.text is a string, but, because map() expects this argument to be a list, the string will be coerced to a list.

map() returns a new list to the text variable. We join it on the empty string to print it.

Higher-order functions

The `map()` function is called a *higher-order function* (HOF) because it takes *another function* as an argument, which is wicked cool. Later we'll use another HOF called `filter()`.

METHOD 7: USING MAP() WITH A NAMED FUNCTION

We are not required to use `map()` with a `lambda` expression. Any function at all will work, so let's go back to using our `new_char()` function:

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

    def new_char(c):
        return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c

    print(''.join(map(new_char, args.text)))
```

Define a function that will return the proper character. Note that I'm using the closure version so as to reference the "vowel" argument.

Use map() to apply new_char() to all the characters in args.text. The result is a list of characters, and we can use str.join() to turn them into a new string for print().

Notice that `map()` uses `new_char` *without parentheses* as the first argument. If you added the parentheses, 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'
```

As shown in figure 8.9, `map()` takes each character from `text` and passes it as the argument to the `new_char()` function, which decides whether to return a `vowel` or the original character. The result of mapping these characters is a new list of characters that we `str.join()` on the empty string to create a new version of `text`.

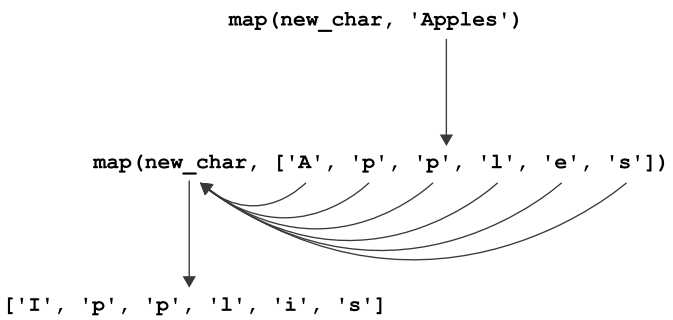


Figure 8.9 `map()` will apply a given function to each element of an iterable. A string will be processed as a list of characters.

METHOD 8: USING REGULAR EXPRESSIONS

A *regular expression* is a way to describe patterns of text. Regular expressions (also called "regexes") are a separate domain-specific language (DSL). They really have nothing whatsoever to do with Python. They have their own syntax and rules, and they are used in many places, from command-line tools to databases. Regexes are incredibly powerful and well worth the effort to learn.

To use regular expressions, you must `import re` in your code to import the regular expression module:

```
>>> import re
```

In this example, we're trying to find characters that are vowels, which we can define as the letters "a," "e," "i," "o," and "u." To describe this idea using a regular expression, we put those characters inside square brackets:

```
>>> pattern = '[aeiou]'
```

We can use the "substitute" function, `re.sub()`, to find all the vowels and replace them with the given `vowel`. The square brackets around the vowels `'[aeiou]'` 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 it is the `vowel` provided by the user. The third argument is the string we want to change, which is the `text` from the user:

```
>>> vowel = 'o'
>>> re.sub(pattern, vowel, 'Apples and bananas!')
'Applos ond bononos!'
```

That misses the capital "A," so we'll have to handle both lower- and uppercase. Here is how we could write that:

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel
    text = re.sub('[aeiou]', vowel, text)
    text = re.sub('[AEIOU]', vowel.upper(), text)
    print(text)
```

Substitute any of the lowercase vowels with the given vowel (which is lowercase because of the restrictions in get_args()).

Substitute any of the uppercase vowels with the uppercase vowel.

If you prefer, we could squash the two calls to `re.sub()` into one, just as we did with the `str.replace()` method shown earlier:

```
>>> text = 'Apples and Bananas!'
>>> text = re.sub('[AEIOU]', vowel.upper(), re.sub('[aeiou]', vowel, text))
>>> text
'Opplos ond Bononos!'
```

One of the biggest differences between this solution and all the others is that we use regular expressions to describe what we are looking for. We didn't have to write the

code to identify the vowels. This is more along the lines of *declarative* programming. We declare what we want, and the computer does the grunt work!

8.4 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 provide you with 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.

I showed many ways to solve this seemingly trivial problem. Some of the techniques using higher-order functions and regular expression are quite advanced techniques. It might seem like driving a finishing nail with a sledgehammer, but I want to start introducing you to programming ideas that I'll visit again and again in later chapters.

If you only really understood the first few solutions, that's fine! Just stick with me. The more times you see these ideas applied in different contexts, the more they will begin to make sense.

8.5 Going further

Write a version of the program that collapses multiple adjacent vowels into a single substituted value. For example, "quick" should become "qack" and not "qaack."

Summary

- 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 `str.replace()` and `str.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 shorthand 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.
- Functions can reference variables declared within the same scope, creating 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.