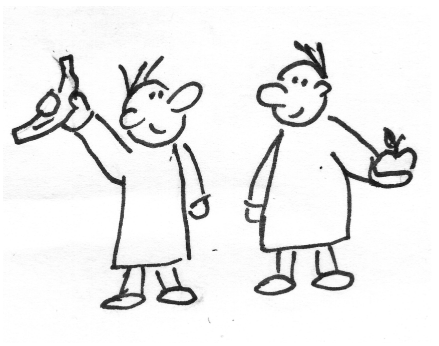
1. Apples and Bananas: Find and replace

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 us consider the children's song "Apples and Bananas" wherein we intone about our favorite fruits to consume:

```
I like to eat, eat, eat apples and bananas
```

Subsequent verses substitute the main vowel sound in the fruits for various other vowel sounds, such as the long "a" (as in "hay") sound:

```
I like to ate, ate, ate ay-ples and ba-nay-nays
```

Or the ever-popular long "e" (as in "knee"):

```
I like to eat, eat, eat ee-ples and bee-nee-nees
```

And so forth. In this exercise, we'll write a Python program called `apples.py` takes some text given as a single positional argument and replaces all the vowels in the text with a given `-v` or `--vowel` options (default `a`).

The program should handle text on the command line:

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

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

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

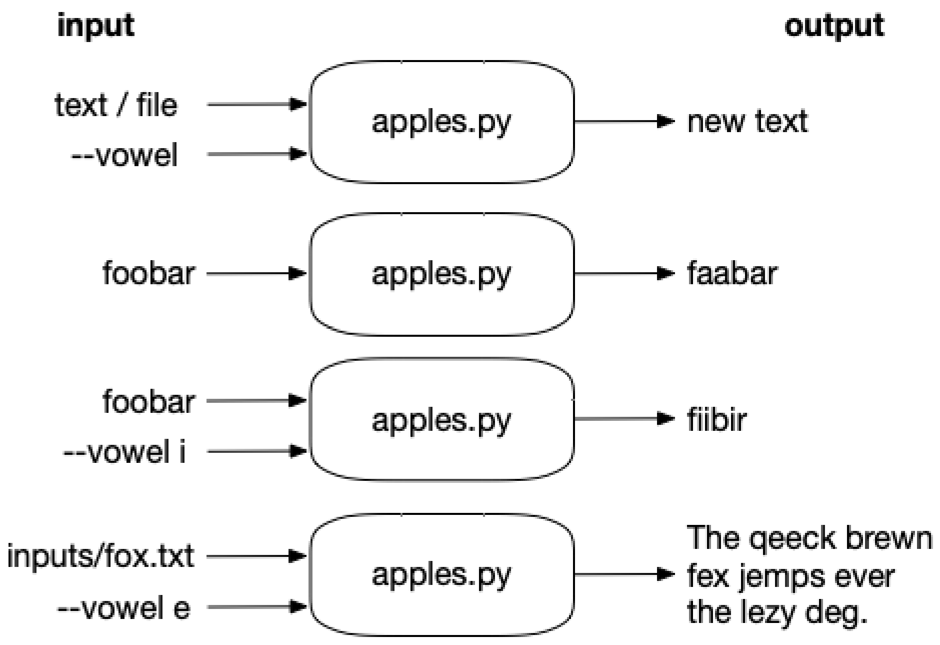
You program should *preserve the case* of the input vowels:

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

As with the "Howler" program, the text argument may name a file in which case your program should read the contents of the file.

```
$ ./apples.py ../inputs/fox.txt
Tha qaack brawn fax jamps avar tha lazy dag.
$ ./apples.py --vowel e ../inputs/fox.txt
The qeeck brewn fex jemps ever the lezy deg.
```

It might help to look at a diagram of the program's inputs and output:



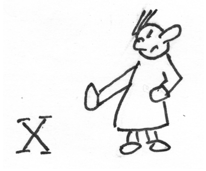
Here is the usage that should print when there are *no arguments*:

```
$ ./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')
```

So our program is going to need to:

- Take a positional argument that might be some plain text or may name a file.
- If the argument is a file, use the contents as the input text.
- Take an optional `-v` or `--vowel` argument that should default to the letter "a".
- Verify that the `--vowel` option is in the set of vowels "a," "e," "i," "o," and "u."
- Replace all instances of vowels in the input text with the specified (or default) `--vowel` argument.
- Print the new text to `STDOUT`.

1.1. 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 "Jump the Five" we saw that you can use a `for` loop to iterate over the characters in a string. For instance, I could laboriously uppercase the `text` like so:

```
new = ''                ①
for char in text:       ②
    new += char.upper() ③
```

① Set a `new` variable equal to the empty string.

② Iterate through each character in the text.

③ Append the uppercase version of the character to the `new` variable.

We can inspect the `new` value to verify that it is all uppercase:

```
>>> new
'THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.'
```

Using this idea, you could iterate through the characters of `text` and build up a new string. Whenever the character is a vowel, you can change it for the given `vowel`, otherwise you use the character itself. We had to identify vowels in "Crow's Nest," so you can refer back to how you did that.

1.1.1. Using the `str.replace()` method

In "Jump The Five," we used the `str.replace()` method that might work. Let's look at the `help`:

```
>>> help(str.replace)
replace(self, old, new, count=-1, /)
    Return a copy with all occurrences of substring old replaced by new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    If the optional argument count is given, only the first count occurrences are
    replaced.
```

Let's play with that in the REPL. I could replace "T" for "X". Can you see a way to replace all the vowels using this idea?

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

Remember that this method never mutates the given string but instead returns a *new string* that you will need to assign to a variable.

1.1.2. Using `str.translate()`

We also looked at the `str.translate()` method. The documentation 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 "Jump The Five," we created a `dict` that associated the string `'1'` to the string `'9'` and so forth:

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

That was the argument to the `str.maketrans()` function that makes a translation table suitable to use with `str.translate()` to change all the characters present in as keys in the dictionary to their corresponding values:

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

What should be the keys and values of the `dict` if you want to change all the vowels, both lower- and uppercase, to some other value?

1.1.3. Other ways to mutate strings

If you know about regular expressions, that's a strong solution. If you haven't heard of those, don't worry as I'll introduce them in the discussion. The point is for you to go *play* with this and come up with a solution. I found 8 ways to change all the vowels to a new character, so there are many ways you could approach this.

How many *different* methods can you find on your own before you look at my solution?

Hints:

- Consider using the `choices` option in the `argparse` documentation for how to constrain the `--vowel` options.
- Be sure to change both lower- and uppercase versions of the vowel, preserving the case of the input characters.

Now is the time to dig in and see what you can do before you look at my solutions.

1.2. Solution

```python
#!/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') ①

    parser.add_argument('-v',
                        '--vowel',
                        help='The vowel(s) allowed',
                        metavar='vowel',
                        type=str,
                        default='a',
                        choices=list('aeiou'))      ②

    args = parser.parse_args()

    if os.path.isfile(args.text):                   ③
        args.text = open(args.text).read().rstrip() ④

    return args


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

    args = get_args()
    text = args.text
    vowel = args.vowel
    new_text = []                          ⑤

    for char in text:                      ⑥
        if char in 'aeiou':                ⑦
            new_text.append(vowel)         ⑧
        elif char in 'AEIOU':              ⑨
            new_text.append(vowel.upper()) ⑩
        else:
            new_text.append(char)          ⑪

    print(''.join(new_text))               ⑫


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

① The input might be text or a file name, so define as a string.

② Use the `choices` to restrict the user to one of the listed vowels.

③ Check if the `text` argument is a file.

④ If it is, read the file, using the `str.rstrip()` to remove any trailing whitespace.

⑤ Create a new list to hold the characters we'll select.

⑥ Iterate through each character of the text.

⑦ See if the current character is in the list of lowercase vowels.

⑧ If it is, use the `vowel` instead of the character.

⑨ See if the current character is in the list of uppercase vowels.

⑩ If it is, use the value of `vowel.upper()` instead of the character.

⑪ Otherwise, take the character itself.

⑫ Print a new string made by joining the `new_text` list on the empty string.

1.3. Discussion

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

1.3.1. 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 file name:

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

The `--vowel` option is also a string, and I decided to use the `choices` option to have `argparse` validate that the user's input is in the `list('aeiou')`:

```
parser.add_argument('-v',
                    '--vowel',
                    help='The vowel to 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')` to have Python turn the `str` "aeiou" into a `list` of the characters. Both of these are the same 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 quote is a `str`, even if it's just one character:

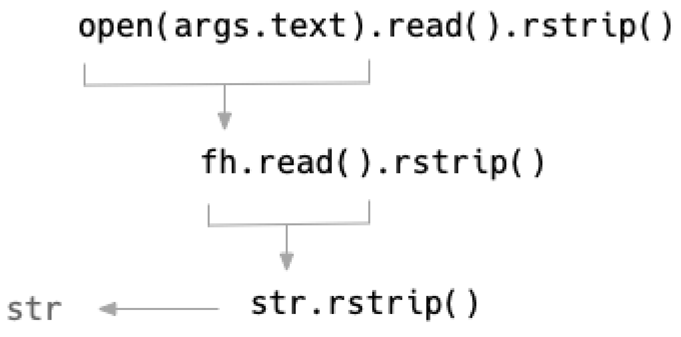
```
>>> ['a', 'e', 'i', 'o', 'u']
['a', 'e', 'i', 'o', 'u']
>>> list('aeiou')
['a', 'e', 'i', 'o', 'u']
```

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 if `text` is the name of a file that should be read for the text or is the text itself. This is the same code I used in "Howler," and again I choose to handle the `text` argument inside the `get_args()` function so that, by the time I get `text` inside my `main()`, it's all be handled:

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



At this point, the user's arguments to the program have been fully vetted. We've got `text` either from the command line or from a file, and we've verified that the `--vowel` is actually one of the allowed characters. To me, this is a single "unit" where I've handled the arguments, and processing can now go forward by returning the arguments:

```
return args
```

1.4. Eight ways to replace the vowels

How many ways did you find to replace the vowels? You only needed one, of course, to pass the test, but I hope you probed the edges of the language to see how many different techniques there are. I know that the Zen of Python says:

> There should be one — and preferably only one — obvious way to do it. - https://www.python.org/dev/peps/pep-0020/

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

1.4.1. Method 1: Iterate every character

The first method is similar to "Jump The Five" where we can use a `for` loop on a string to access each character. Here is code you can copy and paste into the `ipython` REPL

```
>>> text = 'Apples and Bananas!'          ①
>>> vowel = 'o'                          ②
>>> new_text = []                        ③
>>> for char in text:                    ④
...     if char in 'aeiou':              ⑤
...         new_text.append(vowel)       ⑥
...     elif char in 'AEIOU':            ⑦
...         new_text.append(vowel.upper()) ⑧
...     else:
...         new_text.append(char)        ⑨
...
>>> text = ''.join(new_text)             ⑩
>>> text
'Opplos ond Bononos!'
```

① Set a `text` variable to the string "Apples and Bananas!"

② Set the `vowel` variable to the string "o."

③ Set the variable `new_text` to an empty `list`.

④ Use a `for` to iterate `text`, putting each character into the `char` variable.

⑤ If the character is in the set of lowercase vowels,

⑥ Substitute in the `vowel` to the `new_text`.

⑦ If the character is in the set of uppercase vowels,

⑧ Substitute the `vowel.upper()` version into `new_text`.

⑨ Otherwise, use the `char` as-is.

⑩ Turn the `list` called `new_text` into new `str` by joining it on the empty string (`''`).

Note that it would be just fine to start off making `new_text` an empty string and then concatenating the new characters. Then you wouldn't have to `str.join()` them at the end. Whatever you prefer:

```
new_text += vowel
```

1.4.2. Method 2: Using the `str.replace()` method

I'm going to show you several alternate solutions. They're all functionally equivalent because they all pass the tests, but the point here is to explore the Python language and understand it!

For the alternate solutions, I'll just show the `main()` function. 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': ①
        text = text.replace(v, vowel).replace(v.upper(), vowel.upper()) ②

    print(text)
```

① Iterate through the list of vowels. We don't have to say `list('aeiou')` here because Python will automatically treat the string `'aeiou'` like a `list` because we are using it in a *list context* with the `for`.

② Use the `str.replace()` method twice to replace both the lower- and upper-case versions of the vowel in the `text`.

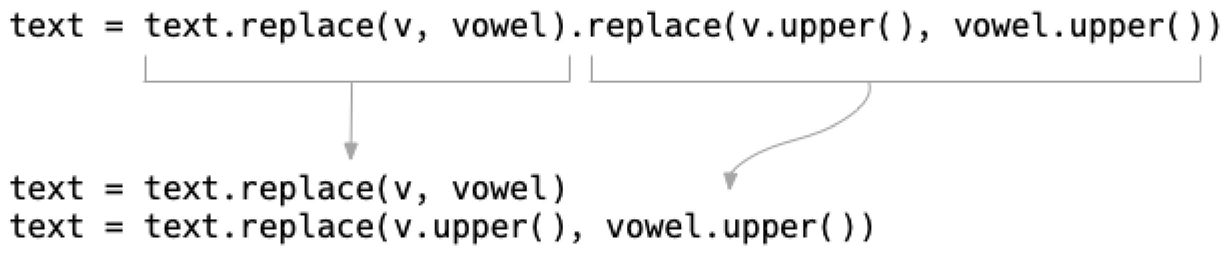
I mentioned in the introduction the `str.replace()` method that will return a new string with all instances of one string replaced by another.

```
>>> s = 'foo'
>>> s.replace('o', 'a')
'faa'
>>> s.replace('oo', 'x')
'fx'
```

Note that the original string remains unchanged:

```
>>> s
'foo'
```

You don't have to chain the two `str.replace()` methods. It could be written as two separate statements:



```
text = text.replace(v, vowel).replace(v.upper(), vowel.upper())

text = text.replace(v, vowel)
text = text.replace(v.upper(), vowel.upper())
```

1.4.3. Method 3: Using the `str.translate()` method

```python
def main():
    args = get_args()
    vowel = args.vowel
    trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5) ①
    text = args.text.translate(trans) ②

    print(text)
```

① Create the translation table.

② Call the `str.translate()` method on the `text` variable passing the `trans` table as the argument.

How can we use `str.translate()` method to solve this? I showed in "Jump The Five" how the `jumper` dictionary could be used to create a translation table using the `str.maketrans()` method to convert each number to another number. In this problem I need to change all the lower- and upper-case vowels (10 total) to some given `vowel`. For instance, to make all the vowels into the letter "o," I could create translation table `t` like so:

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

I can 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 make the translation 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 I want to replace which are the lower- and uppercase vowels `'aeiouAEIOU'`. The second string is composed of the letters to use for substitution. I would like to use `'ooooo'` for the `'aeiou'` and `'OOOOO'` for `'AEIOU'`. I can repeat my `vowel` five times using the `*` operator that you normally associate with numeric multiplication. This is (sort of) "multiplying" a string, so, OK, I guess:

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

Now to handle the uppercase version, too:

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

Now to make the translation table:

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

Let’s inspect the `trans` table. I want to "pretty print" the data structure so I can see it, so I will use the `pprint.pprint` (pretty print) function:

```
>>> from pprint import pprint as pp
>>> pp(trans)
{65: 79,
 69: 79,
 73: 79,
 79: 79,
 85: 79,
 97: 111,
 101: 111,
 105: 111,
 111: 111,
 117: 111}
```

The enclosing curlies `{}` tell us that `trans` is a `dict`. Each character is represented by it’s *ordinal* value, which is the character’s position in the ASCII table (http://www.asciitable.com/). (You will use this later in the "Gematria" program.) You can go back and forth from characters and their ordinal values by using the `chr()` and `ord()` functions. Here are the `ord()` values for the vowels:

```
>>> for char in 'aeiou':
...     print(char, ord(char))
...
a 97
e 101
i 105
o 111
u 117
```

And here you can create the same output but starting with the `ord()` values to get the `chr()` values:

```
>>> for num in [97, 101, 105, 111, 117]:
...     print(chr(num), num)
...
a 97
e 101
i 105
o 111
u 117
>>>
```

If you’d like to inspect all the ordinal values for all the printable characters, you can run this:

```
>>> import string
>>> for char in string.printable:
...     print(char, ord(char))
```

I don’t include the output because there are 100 printable characters:

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

So the `trans` table is a mapping. The lowercase vowels ("aeiou") all map to the ordinal value `111` which is "o." The uppercase vowels ("AEIOU") map to 79 which is "O." I can use the `dict.items()` method to iterate over the key/value pairs of `trans` to verify this is the case:

```
>>> for x, y in trans.items():
...     print(f'{chr(x)} => {chr(y)}')
...
a => o
e => o
i => o
o => o
u => o
A => O
E => O
I => O
O => O
U => O
```

The original `text` will be unchanged by the `str.translate()` method, so we overwrite `text` with the new version:

```
>>> text = 'Apples and Bananas!' ①
>>> trans = str.maketrans('aeiouAEIOU', vowel * 5 + vowel.upper() * 5) ②
>>> text = text.translate(trans) ③
>>> text
'Opplos ond Bononos!'
```

① Initialize `text`.

② Make a translation table.

③ Use the translation table as the argument to `text.translate()`. Overwrite the original value of `text` with the result.

That was a lot of explanation about `ord()` and `chr()` and dictionaries and such, but look how simple and elegant that solution is! This is much shorter than Method 1. Fewer lines of code (LOC) means fewer opportunities for bugs!

1.4.4. Method 4: List comprehension

```
def main():
    args = get_args()
    vowel = args.vowel
    text = [ ①
        vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c ②
        for c in args.text
    ]
    print(''.join(text)) ③
```

① Use a list comprehension to process all the characters in `args.text` to create a new `list` called `text`.

② Use a compound `if` expression to handle three cases (lowercase vowel, uppercase vowel, the default).

③ Print a new string by joining `text` on the empty string.

Following up on Method 1, we can use a "list comprehension" to significantly shorten the `for` loop. In Gashlycrumb we looked at a "dictionary comprehension" as a one-line method to create a new dictionary using a `for` loop. Here we can do the same, creating a new `list`.

For example, let's generate a list of the squared values of the numbers 1 through 4. We can use the `range()` function to get the numbers from a starting number to an ending number (not inclusive!). `range()` is a *lazy* function, which means it won't actually produce values until your program actually needs them. That is, 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, then the work is never done, meaning your code is more efficient!

In the REPL, I must use the `list()` function to force the production of the values, but most of the time your code doesn't need to do this:

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

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

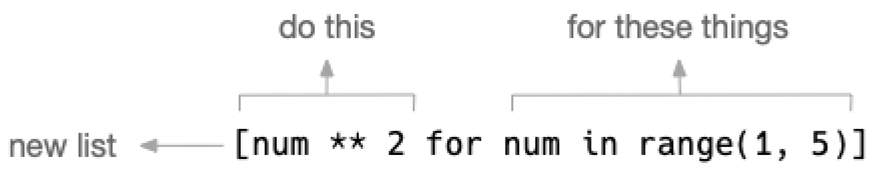
```
>>> for num in range(1, 5):
...     print(num ** 2)
...
1
4
9
16
```

But what I really want is a `list` that contains those values. A simple way to do this is to create an empty `list` to which we will `list.append()` those values:

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

And now I can verify that I have my squares:

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



```
[num ** 2 for num in range(1, 5)]
```

A list comprehension is novel from a `for` loop in that it *returns a new list*:

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

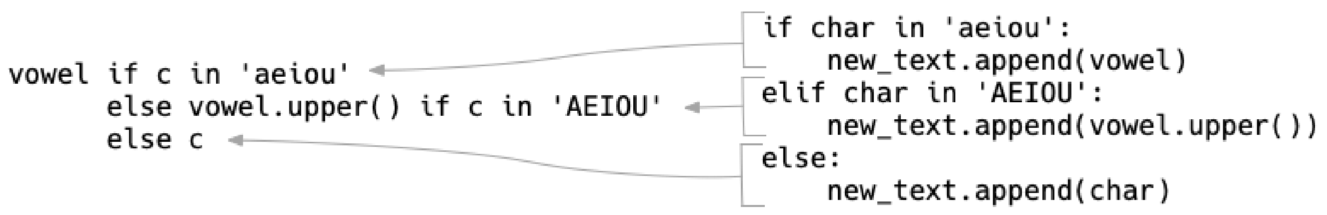
I can assign this to the `squares` variable and verify that I still have what I expected. Ask yourself which version of the code you'd rather maintain, the longer one with the `for` loop or the shorter one with the list comprehension?

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

For our program, we're going to condense the `if`/`elif`/`else` logic from Method 1 into a compound `if` expression. First let's see how we could shorter the `for` loop version:

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

Here's a diagram that shows how the parts of the expression match up to the original `if`/`elif`/`else`:



```
vowel if c in 'aeiou'
      else vowel.upper() if c in 'AEIOU'
      else c
```

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

And now to turn that into a list comprehension:

```
>>> text = 'Apples and Bananas!'
>>> new_text = [
...     vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c ①
...     for c in text ] ②
...
>>> ''.join(new_text)
'Opplos ond Bononos!'
```

① Do this

② For these things.

1.4.5. Method 5: List comprehension with function

The compound `if` expression inside the list comprehension borders 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, it's the same compound `if` expression as before:

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

    def new_char(c): ①
        return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c ②

    text = ''.join([new_char(c) for c in args.text]) ③

    print(text)
```

① 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.

② Use the compound `if` expression to select the correct character.

③ 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'
```

And "O" if the argument is an uppercase vowel:

```
>>> new_char('A')
'O'
```

Otherwise it should return the given character:

```
>>> new_char('b')
'b'
```

We can use 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!'
```

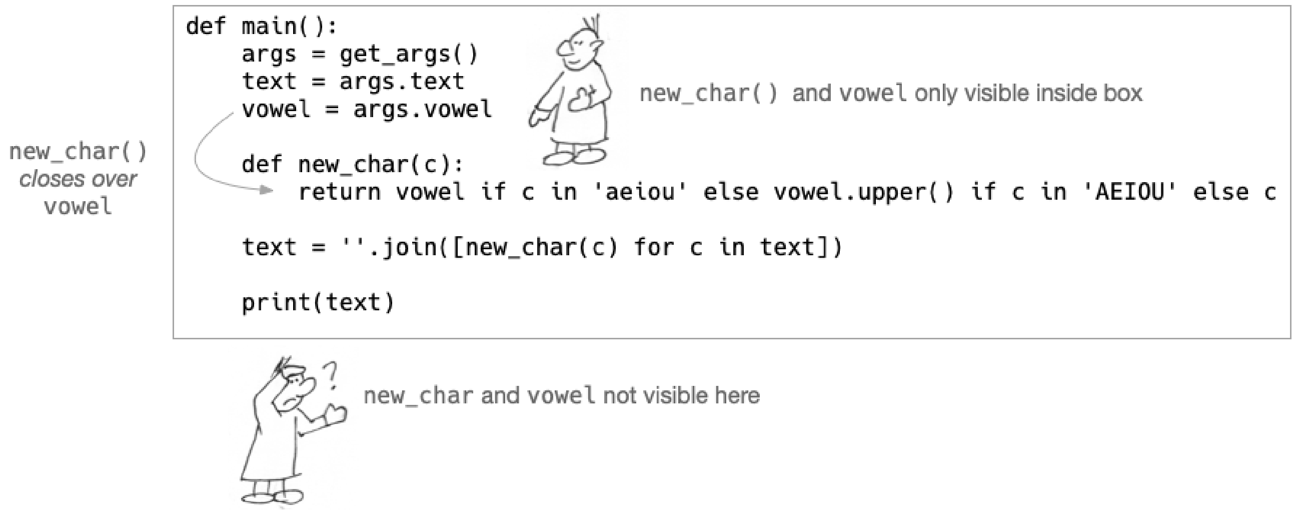
A note about the fact that the `new_char()` function is declared *inside* the `main()` function. Yes, you can do that! The function is then only "visible" inside the `main()` function. The reason I do this is because I want to reference the `vowel` variable inside the function without actually passing it as an argument.

As an example, I define a `foo()` function that has a `bar()` function inside it. I can call `foo()` and it will call `bar()`, but from outside of `foo()` the `bar()` function does not exist ("is not visible" or "is not in scope"):

```
>>> def foo():
...     def bar():
...         print('This is bar')
...     bar()
...
>>> foo()
This is bar
>>> bar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bar' is not defined
```

I did this because I actually created a special type of function with `new_char()` called a "closure" because it is "closing" around the `vowel`. That is, I wanted to use reference the `vowel` variable inside the `new_char()` function without explicitly passing it as an argument. If I declare the function in the same scope as `vowel`, I can do this.

If I had defined `new_char()` outside of `main()`, the `vowel` would not be visible to `new_char()` because it only exists inside the `main()` function. If we draw a box around the `main()` function, that is where the `new_char()` function can be seen. Outside of that, the function is consider undefined.


new_char() closes over vowel
new_char() and vowel only visible inside box
new_char and vowel not visible here

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

    print(text)
```

Figure 8. 1. Visibility of the `new_char()` *function is limited just to the* `main()` *function. Code outside of* `main()` *cannot see or call* `new_char()`*. This also makes it difficult to unit test the function!*

There are many reasons why to write a function like this, even if it has just one line. I think of functions as *units* of code that describe some concept — ideally just *one* per function! We can formalize our understand of our functions with assertions:

```
>>> assert all([new_char(v) == 'O' for v in 'AEIOU'])
>>> assert all([new_char(v) == 'o' for v in 'aeiou'])
```

In two lines of code, I’ve just tested all 10 vowels by using list comprehensions plus the `all()` function. Let’s take a moment to understand `all()` and `any()`. If you read `help(all)`, you’ll see "Return True if bool(x) is True for all values x in the iterable." So all the values need to the `True` for the entire expression to be `True`:

```
>>> all([True, True, True])
True
```

It’s the same as putting `and` in between all the values:

```
>>> True and True and True
True
```

If any value is `False`, then the whole expression is `False`:

```
>>> all([True, False, True])
False
```

The `any()` function returns `True` if *any* of the values are `True`:

```
>>> any([True, False, True])
True
```

It’s the same as putting `or` in between all the values [1]:

```
>>> True or False or True
True
```

And `False` if there are no `True` values:

```
>>> any([False, False, False])
False
```

We can check the values manually:

```
>>> [new_char(v) == 'O' for v in 'AEIOU']
[True, True, True, True, True]
```

So then `all()` should be `True` for the entire expression:

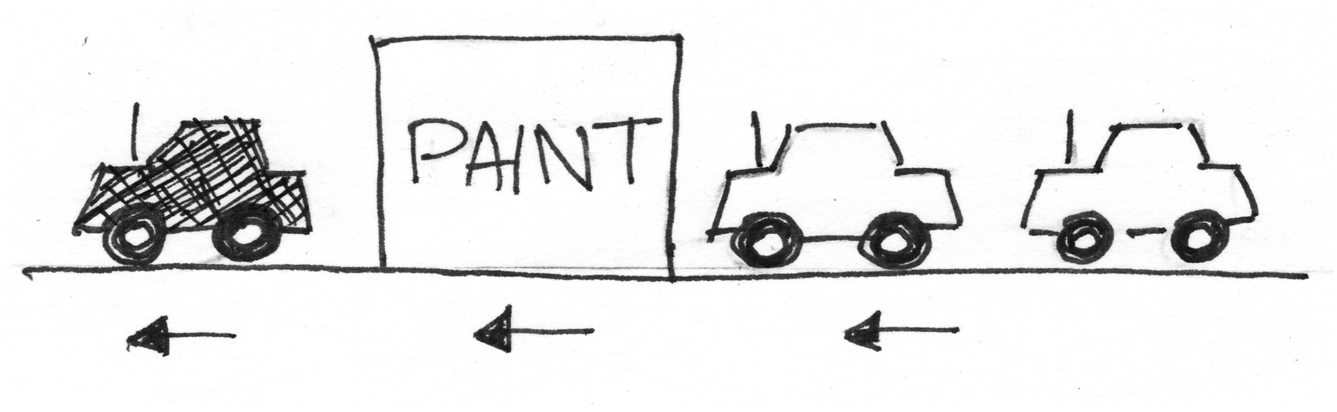
```
>>> all([new_char(v) == 'O' for v in 'AEIOU'])
True
```

We can move these into formal `test_` functions that can be run with `pytest`. This is the idea of *unit testing* where I personally think of functions as units. Some people have other ideas about what is the best *unit* to test. I recommend you read further about testing to understand the range of opinions.

1.4.6. Method 6: The `map()` function

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

1. A function
2. An iterable like a list, lazy function, or 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. I can create a function to "paint" my cars by adding the string "blue" to the beginning:

```
>>> list(map(lambda car: 'blue ' + car, ['BMW', 'Alfa Romeo', 'Chrysler']))
['blue BMW', 'blue Alfa Romeo', 'blue Chrysler']
```

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

Think about regular named functions like `add1()` that adds `1` to a value:

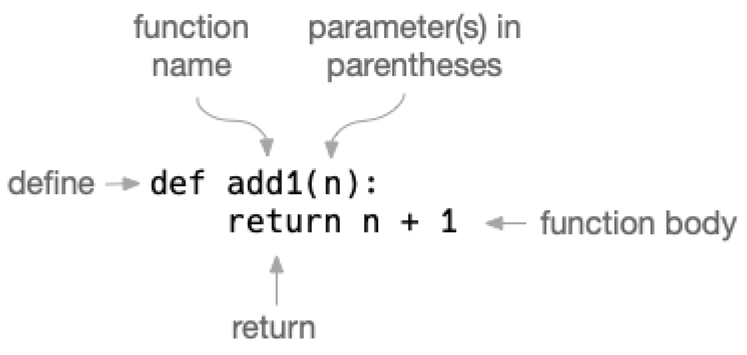
```
def add1(n):
    return n + 1
```

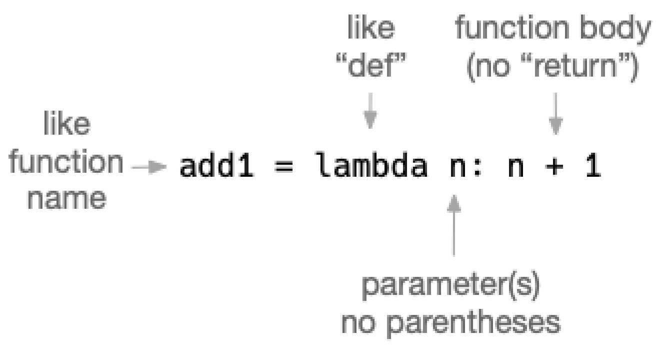
It works as expected:

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

Now compare to the `lambda` version. We can assign it to the variable `add1` which then kinda sorta acts like the function's name:

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





It works exactly the same as using `def` to define a function the normal way:

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

The function body for a `lambda` pretty much needs to fit on one line, and they don't have `return` at the end. In both versions, the argument to the function is `n`. In the usual `def add(n)`, the argument is defined in the parentheses just after the function name. In the `lambda n` version, there is no function name and no parentheses around the function's parameter, `n`.

There is no difference in how you can use them. They are both functions:

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

If you are comfortable with using `add1()` in a list comprehension:

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

Then it's a short step to using the `map()` function. Note that `map()` is a lazy function like the `range()` function we looked earlier. It won't actually create the values until you actually need them as opposed 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, I need to use the list() function:

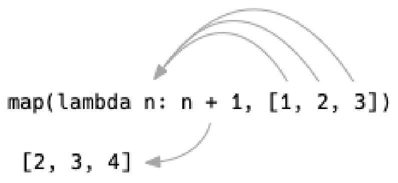
```
>>> list(map(add1, [1, 2, 3]))
[2, 3, 4]
```

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

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

Which looks very similar to the lambda code:

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



So here is how I could use a map():

```
def main():
    args = get_args()
    vowel = args.vowel
    text = map(                                        ①
        lambda c: vowel if c in 'aeiou' else vowel.upper() ②
        if c in 'AEIOU' else c, args.text)             ③

    print(''.join(text)) ④
```

① The map() function wants a function for the first argument and a list for the second.

② Use lambda to create an *anonymous* function that accepts character, c.

③ args.text is the second argument to map(). Because map() expects this argument to be a list, it will automatically coerce it to a list (which is what we want).

④ The map() returned a new list into the text variable, so 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 as argument, which is wicked cool. Later we'll use another HOF called filter().

1.4.7. Method 7: Using map() with a defined function

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

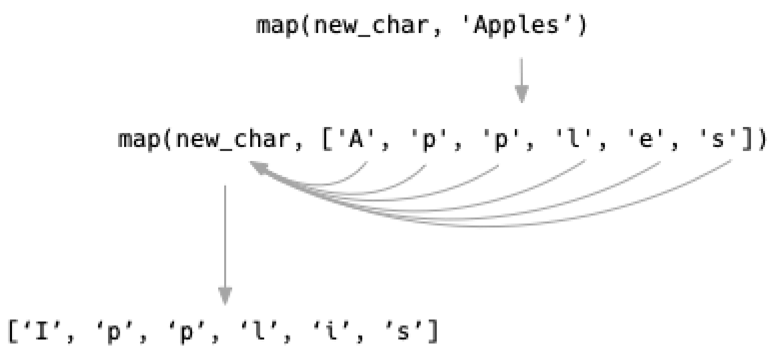
```
def main():
    args = get_args()
    vowel = args.vowel

    def new_char(c): ①
        return vowel if c in 'aeiou' else vowel.upper() if c in 'AEIOU' else c

    print(''.join(map(new_char, args.text))) ②
```

① Define a function that will return the proper character.

② Use `map()` to apply `new_char()` to all the characters in `text`. The result is a `list` of characters which we can use `str.join()` to create a new string for `print()`.

Notice that `map()` uses `new_char` *without parentheses* as the first argument. If you added the parens, you'd be *calling* the function and would see this error:

```
>>> text = ''.join(map(new_char(), text))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: new_char() missing 1 required positional argument: 'c'
```


map(new_char, 'Apples')
map(new_char, ['A', 'p', 'p', 'l', 'e', 's'])
['I', 'p', 'p', 'l', 'i', 's']

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

1.4.8. Method 8: Using regular expressions

A "regular expression" is a term from linguistics and computer science. It is a way of describing a "regular language" [2]. Essentially regular expressions give us a way to *describe patterns of text*, which is truly, outstandingly useful.

To use regular expressions, you must `import re` in your code. Here I will show how we can use the "substitute" function `re.sub()` to find any of the lower- and uppercase vowels and replace them with the given `vowel`:

```
def main():
    args = get_args()
    text = args.text
    vowel = args.vowel
    text = re.sub('[aeiou]', vowel, text)         ①
    text = re.sub('[AEIOU]', vowel.upper(), text) ②
    print(text)
```

① Substitute any of the lowercase vowels for the given vowel (which is lowercase because of the restrictions in `get_args()`).

② Substitute any of the uppercase vowels for the uppercased vowel.

Regular expressions (also called "regexes") are a separate domain-specific language (DSL). That is, 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 them.

The `re.sub()` function will *substitute* all instances of text matching a given pattern for a new string. 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 our `vowel` provided by the user. The third argument is the string we want to change, which is the `text` from the user.

```
>>> import re
>>> text = 'Apples and Bananas!'
>>> vowel = 'o'
>>> re.sub('[aeiou]', vowel, text)
'Applos ond Bononos!'
```

Note that `re.sub()` returns a new string, and the original `text` remains unchanged by the operation:

```
>>> text
'Apples and Bananas!'
```

That almost worked, but it missed the uppercase vowel "A". I could overwrite the `text` in two steps to get both lower- and uppercase:

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

Or do it in one step just like the `str.replace()` method 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 with this solution to all the others is how we use regular expressions to describe what we were looking for and didn't have to write the code to actually find the characters! This is more along the lines of *declarative* programming. We declare what we want, and the computer does the grunt work!

1.5. Refactoring with tests

There are many ways to solve this problem. The most important step is to get your program to work properly. Tests let you know when you've reached that point. From there, you can explore other ways to solve the problem and keep using the tests to ensure you still have a correct program. Tests actually provide great freedom to be creative. Always be thinking about tests you can write for your own programs so that, when you change them later, they will always keep working!

1.6. Review

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.

- 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 short-hand way to write a `for` loop inside `[]` to create a new `list`.
- Functions can be defined inside other functions. Their visibility is then limited to the enclosing function. * Function 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.

1.7. Going Further

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

[1] Here is a lesson in logic brought to you by my youngest who used to try to get out of flossing his teeth. When he would come out of the bathroom, I would ask "Did you brush and floss your teeth?" He would invariably reply "Yes." I would then ask "Did you brush your teeth?" "Yes," he would say. "Did you floss your teeth?" I would ask? "No," he would usually say. So, he was using `or` while I was using `and`.

[2] https://en.wikipedia.org/wiki/Regular_language