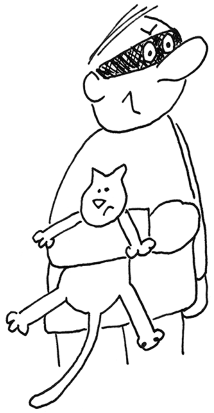
12 Ransom: Randomly capitalizing text

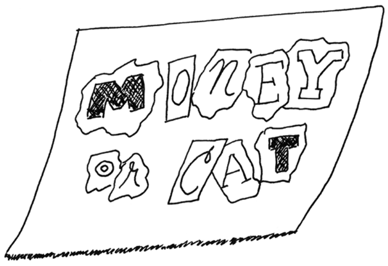


All this hard work writing code is getting on my nerves. I'm ready to turn to a life of crime! I've kidnapped (cat-napped?) the neighbor's cat, and I want to send them a ransom note. In the good old days, I'd cut letters from magazines and paste them onto a piece of paper to spell out my demands. That sounds like too much work. Instead, I'm going to write a Python program called ransom.py that will encode text into randomly capitalized letters:

```
$ ./ransom.py 'give us 2 million dollars or the cat
     gets it!'
gIVe US 2 milLION DoLlArs or ThE cAt GEts It!
```

As you can see, my diabolical program accepts the heinous input text as a positional argument. Since this program uses the `random` module, I want to accept an `-s` or `--seed` option so I can replicate the vile output:

```
$ ./ransom.py --seed 3 'give us 2 million dollars or the cat gets it!'
giVE uS 2 MILlioN dolLaRS OR tHe cAt GETS It!
```



The dastardly positional argument might name a vicious file, in which case that should be read for the demoniac input text:

```
$ ./ransom.py --seed 2 ../inputs/fox.txt
the qUIck BROWN fOX JUmps ovEr ThE LAZY DOg.
```

If the unlawful program is run with no arguments, it should print a short, infernal usage statement:

```
$ ./ransom.py
usage: ransom.py [-h] [-s int] text
ransom.py: error: the following arguments are required: text
```

If the nefarious program is run with `-h` or `--help` flags, it should print a longer, fiendish usage:

```
$ ./ransom.py -h
usage: ransom.py [-h] [-s int] text

Ransom Note

positional arguments:
  text                 Input text or file

optional arguments:
  -h, --help           show this help message and exit
  -s int, --seed int   Random seed (default: None)
```

Figure 12.1 shows a noxious string diagram to visualize the inputs and outputs.

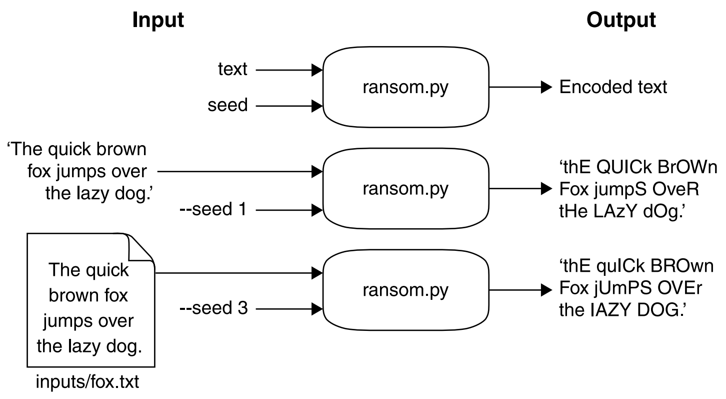

Input
Output
text
seed
ransom.py
Encoded text
'The quick brown fox jumps over the lazy dog.'
--seed 1
ransom.py
'thE QUICk BrOWn Fox jumpS OveR tHe LAzY dOg.'
The quick brown fox jumps over the lazy dog.
inputs/fox.txt
--seed 3
ransom.py
'thE quICk BROwn Fox jUmPS OVEr the IAZY DOG.'

Figure 12.1 The awful program will transform input text into a ransom note by randomly capitalizing letters.

In this chapter, you will

- Learn how to use the `random` module to figuratively "flip a coin" to decide between two choices
- Explore ways to generate new strings from an existing one, incorporating random decisions
- Study the similarities of `for` loops, list comprehensions, and the `map()` function

12.1 Writing ransom.py

I suggest starting with new.py or copying the template/template.py file to create ransom.py in the 12_ransom directory. This program, like several before it, accepts a required, positional string for the `text` and an optional integer (default `None`) for the `--seed`. Also, as in previous exercises, the `text` argument may name a file that should be read for the `text` value.

To start out, use this for your `main()` code:

```
def main():
    args = get_args()
    random.seed(args.seed)
    print(args.text)
```

Get the processed command-line arguments.

Set the random.seed() with the value from the user. The default is None, which is the same as not setting it.

Start off by echoing back the input.

If you run this program, it should echo the input from the command line:

```
$ ./ransom.py 'your money or your life!'
your money or your life!
```

Or the text from an input file:

```
$ ./ransom.py ../inputs/fox.txt
The quick brown fox jumps over the lazy dog.
```

The important thing when writing a program is to take baby steps. You should run your program *after every change*, checking manually and with the tests to see if you are progressing.

Once you have this working, it's time to think about how to randomly capitalize this awful message.

12.1.1 Mutating the text

You've seen before that you can't directly modify a `str` value:

```
>>> text = 'your money or your life!'
>>> text[0] = 'Y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

So how can we randomly change the case of some of the letters?

I suggest that instead of thinking about how to change many letters, you should think about how to change *one* letter. That is, given a single letter, how can you randomly return the upper- or lowercase version of the letter? Let's create a dummy `choose()` function that accepts a single character. For now, we'll have the function return the character unchanged:

```
def choose(char):
    return char
```

Here's a test for it:

```
def test_choose():
    state = random.getstate()
    random.seed(1)
    assert choose('a') == 'a'
    assert choose('b') == 'b'
    assert choose('c') == 'C'
    assert choose('d') == 'd'
    random.setstate(state)
```

The state of the random module is global to the program. Any change we make here could affect unknown parts of the program, so we save our current state.

Set the random seed to a known value. This is a global change to our program. Any other calls to functions from the random module will be affected!

The choose() function is given a series of letters, and we use the assert statement to test if the value returned by the function is the expected letter.

Reset the global state to the original value.

Random seeds

Have you wondered how I knew what would be the result of `choose()` for a given random seed? Well, I confess that I wrote the function, then set the seed, and ran it with the given inputs. I recorded the results as the assertions you see. In the future, these results should still be the same. If they are not, I've changed something and probably broken my program.

12.1.2 Flipping a coin

We need to `choose()` between returning the upper- or lowercase version of the character you are given. It's a *binary* choice, meaning we have two options, so we can use the analogy of flipping a coin. Heads or tails? Or, for our purposes, 0 or 1:

```
>>> import random
>>> random.choice([0, 1])
1
```

Or `True` or `False` if you prefer:

```
>>> random.choice([False, True])
True
```

Think about using an `if` expression where you return the uppercase answer when the `0` or `False` option is selected and the lowercase version otherwise. My entire `choose()` function is this one line.

12.1.3 Creating a new string

Now we need to apply our `choose()` function to each character in the input string. I hope this is starting to feel like a familiar tactic. I encourage you to start by mimicking the first approach from chapter 8 where we used a `for` loop to iterate through each character of the input text and replace all the vowels with a single vowel. In this program, we can iterate through the characters of text and use them as the argument to the `choose()` function. The result will be a new `list` (or `str`) of the transformed characters. Once you can pass the test with a `for` loop, try to rewrite it as a list comprehension, and then a `map()`.

Now off you go! Write the program, pass the tests.

12.2 Solution

We're going to explore many ways to process all the characters in the input text. We'll start off with a `for` loop that builds up a new list, and I hope to convince you that a list comprehension is a better way to do this. Finally, I'll show you how to use `map()` to create a very terse (perhaps even elegant) solution.

```
#!/usr/bin/env python3
"""Ransom note"""

import argparse
import os
import random


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

    parser = argparse.ArgumentParser(
        description='Ransom Note',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    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
```

The text argument is a positional string value.

The --seed option is an integer that defaults to None.

Process the command-line arguments into the args variable.

If the args.text is a file, use the contents of that as the new args.text value.

Return the arguments to the caller.

Set the random.seed() to the given args.seed value. The default is None, which is the same as not setting it. That means the program will appear random when no seed is given but will be testable when we do provide a seed value.

Create an empty list to hold the new ransom message.

Use a for loop to iterate through each character of args.text.

Append the chosen letter to the ransom list.

```
    random.seed(args.seed)
    ransom = []
    for char in args.text:
        ransom.append(choose(char))

    print(''.join(ransom))
```

Join the ransom list on the empty string to create a new string to print.

Define a function to randomly return the upper- or lowercase version of a given character.

```
# --------------------------------------------------
def choose(char):
    """Randomly choose an upper or lowercase letter to return"""

    return char.upper() if random.choice([0, 1]) else char.lower()
```

Use random.choice() to select either 0 or 1, which, in the Boolean context of the if expression, evaluates to False or True, respectively.

Save the current state of the random module.

Define a test_choose() function that will be run by Pytest. The function takes no arguments.

Set the random.seed() to a known value for the purposes of the test.

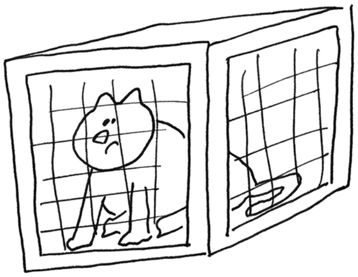
```
# --------------------------------------------------
def test_choose():
    """Test choose"""

    state = random.getstate()
    random.seed(1)
    assert choose('a') == 'a'
    assert choose('b') == 'b'
    assert choose('c') == 'C'
    assert choose('d') == 'd'
    random.setstate(state)
```

Use the assert statement to verify that we get the expected result from the choose() for a known argument.

Reset the random module's state so that our changes won't affect any other part of the program.

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

12.3 Discussion



I like this problem because there are so many interesting ways to solve it. I know, I know, Python likes there to be "one obvious way" to solve it, but let's explore, shall we? There's nothing in `get_args()` that we haven't seen several times by now, so let's skip that.

12.3.1 Iterating through elements in a sequence

Assume that we have the following cruel message:

```
>>> text = '2 million dollars or the cat sleeps with the fishes!'
```

I want to randomly upper- and lowercase the letters. As suggested in the earlier description of the problem, we can use a `for` loop to iterate over each character.

One way to print an uppercase version of the `text` is to print an uppercase version *of each letter*:

```
for char in text:
    print(char.upper(), end='')
```

That would give me “2 MILLION DOLLARS OR THE CAT SLEEPS WITH THE FISHES!” Now, instead of always printing `char.upper()`, I can randomly choose between `char.upper()` and `char.lower()`. For that, I’ll use `random.choice()` to choose between two values like `True` and `False` or `0` and `1`:

```
>>> import random
>>> random.choice([True, False])
False
>>> random.choice([0, 1])
0
>>> random.choice(['blue', 'green'])
'blue'
```

Following the first solution from chapter 8, I created a new `list` to hold the ransom message and added these random choices:

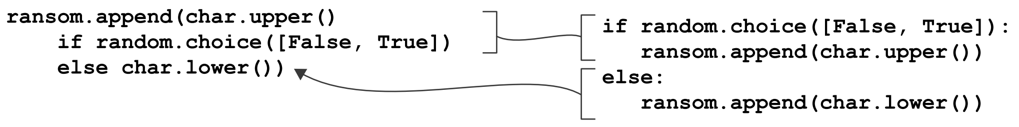
```
ransom = []
for char in text:
    if random.choice([False, True]):
        ransom.append(char.upper())
    else:
        ransom.append(char.lower())
```

Then I joined the new characters on the empty string to print a new string:

```
print(''.join(ransom))
```

It’s far less code to write this with an `if` expression to select whether to take the upper- or lowercase character, as shown in figure 12.2:

```
ransom = []
for char in text:
    ransom.append(char.upper() if random.choice([False, True]) else char.lower())
```



```
ransom.append(char.upper()
    if random.choice([False, True])
    else char.lower())
```

```
if random.choice([False, True]):
   ransom.append(char.upper())
else:
   ransom.append(char.lower())
```

Figure 12.2 A binary if/else branch is more succinctly written using an `if` expression.

You don't have to use actual Boolean values (`False` and `True`). You could use `0` and `1` instead:

```
ransom = []
for char in text:
    ransom.append(char.upper() if random.choice([0, 1]) else char.lower())
```

When numbers are evaluated *in a Boolean context* (that is, in a place where Python expects to see a Boolean value), `0` is considered `False`, and every other number is `True`.

12.3.2 Writing a function to choose the letter

The `if` expression is a bit of code that could be put into a function. I find it hard to read inside the `ransom.append()`.

By putting it into a function, I can give it a descriptive name and write a test for it:

```
def choose(char):
    """Randomly choose an upper or lowercase letter to return"""

    return char.upper() if random.choice([0, 1]) else char.lower()
```

Now I can run the `test_choose()` function to test that my function does what I think. This code is much easier to read:

```
ransom = []
for char in text:
    ransom.append(choose(char))
```

12.3.3 Another way to write list.append()

The solution in section 12.2 creates an empty `list`, to which I `list.append()` the return from `choose()`. Another way to write `list.append()` is to use the `+=` operator to add the right-hand value (the element to add) to the left-hand side (the list), as in figure 12.3.

```
def main():
    args = get_args()
    random.seed(args.seed)

    ransom = []
    for char in args.text:
        ransom += choose(char)

    print(''.join(ransom))
```

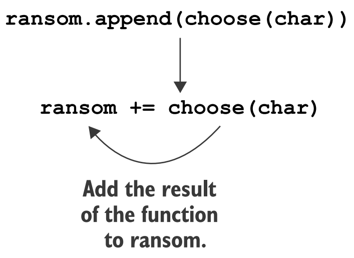

ransom.append(choose(char))
ransom += choose(char)
Add the result of the function to ransom.

Figure 12.3 The `+=` operator is another way to write `list.append()`.

This is the same syntax for concatenating a character to a string or adding a number to another number.

12.3.4 Using a str instead of a list

The two previous solutions require that the lists be joined on the empty string to make a new string to print. We could, instead, start off with an empty string and build that up, one character at a time, using the `+=` operator:

```
def main():
    args = get_args()
    random.seed(args.seed)

    ransom = ''
    for char in args.text:
        ransom += choose(char)

    print(ransom)
```

As we just noted, the `+=` operator is another way to append an element to a list. Python often treats strings and lists interchangeably, often implicitly, for better or worse.

12.3.5 Using a list comprehension

The previous patterns all initialize an empty `str` or `list` and then build it up with a `for` loop. I'd like to convince you that it's almost always better to express this using a list comprehension, because its entire raison d'être is to return a new list. We can condense our three lines of code to just one:

```
def main():
    args = get_args()
    random.seed(args.seed)
    ransom = [choose(char) for char in args.text]
    print(''.join(ransom))
```

Or you can skip creating the `ransom` variable altogether. As a general rule, I only assign a value to a variable if I use it more than once or if I feel it makes my code more readable:

```
def main():
    args = get_args()
    random.seed(args.seed)
    print(''.join([choose(char) for char in args.text]))
```

A `for` loop is really for iterating through some sequence and producing *side effects*, like printing values or handling lines in a file. If your goal is to create a new `list`, a list comprehension is probably the best tool. Any code that would go into the body of the `for` loop to process an element is better placed in a function with a test.

12.3.6 Using a map() function

I've mentioned before that `map()` is just like a list comprehension, though usually with less typing. Both approaches generate a new `list` from some iterable, as shown in figure 12.4. In this case, the resulting list from `map()` is created by applying the `choose()` function to each character of `args.text`:

```
def main():
    args = get_args()
    random.seed(args.seed)
    ransom = map(choose, args.text)
    print(''.join(ransom))
```

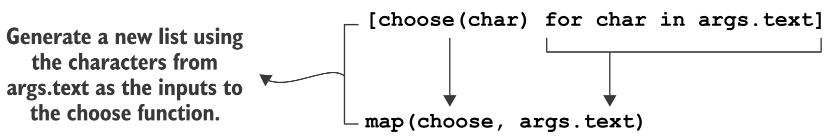

Generate a new list using the characters from args.text as the inputs to the choose function.
[choose(char) for char in args.text]
map(choose, args.text)

Figure 12.4 The ideas of the list comprehension can be expressed more succinctly with `map()`.

Or, again, you could leave out the `ransom` assignment and use the `list` that comes back from `map()` directly:

```
def main():
    args = get_args()
    random.seed(args.seed)
    print(''.join(map(choose, args.text)))
```

12.4 Comparing methods

It may seem silly to spend so much time working through so many ways to solve what is essentially a trivial problem, but one of the goals of this book is to explore the various ideas available in Python. The first solution in section 12.2 is a very imperative solution that a C or Java programmer would probably write. The version using a list comprehension is very idiomatic to Python—it is "Pythonic," as Pythonistas would say. The `map()` solution would look very familiar to someone coming from a purely functional language like Haskell.

All these approaches accomplish the same goal, but they embody different aesthetics and programming paradigms. My preferred solution would be the last one, using `map()`, but you should choose an approach that makes the most sense to you.

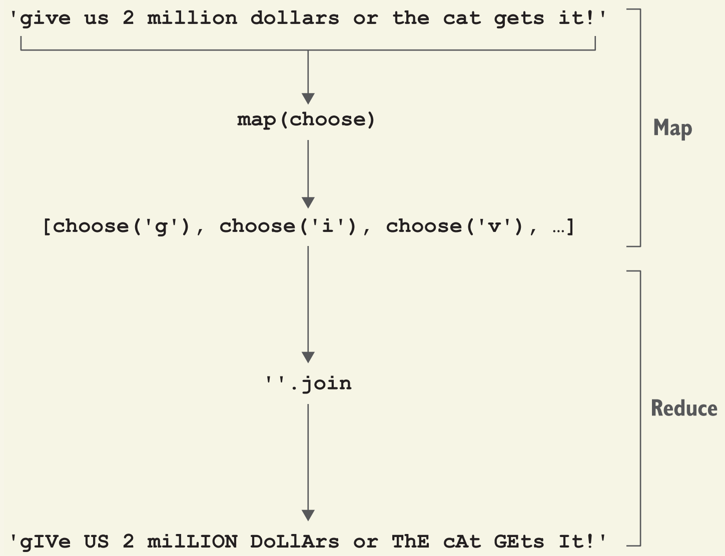
MapReduce

In 2004, Google released a paper on their "MapReduce" algorithm. The "map" phase applies some transformation to all the elements in a collection, such as all the pages of the internet that need to be indexed for searching. These operations can happen in *parallel*, meaning you can use many machines to process the pages separately from each other and in any order. The "reduce" phase then brings all the processed elements back together, maybe to put the results into a unified database.

In our ransom.py program, the "map" part selected a randomized case for the given letter, and the "reduce" part was putting all those bits back together into a new string. Conceivably, `map()` could make use of multiple processors to run the functions *in parallel* as opposed to *sequentially* (like with a `for` loop), possibly cutting the time to produce the results.

The ideas of map/reduce can be found in many places, from indexing the internet to our ransom program.

Learning about MapReduce was, to me, a bit like learning the name of a new bird. I never even noticed that bird before, but, once I was told its name, I saw it everywhere. Once you understand this pattern, you'll begin to see it in many places.


'give us 2 million dollars or the cat gets it!'
map(choose)
[choose('g'), choose('i'), choose('v'), …]
''.join
'gIVe US 2 milLION DoLlArs or ThE cAt GEts It!'
Map
Reduce

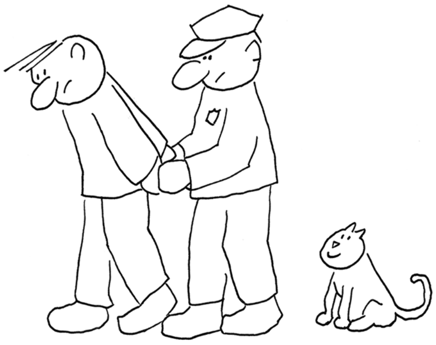
12.5 Going further

Write a version of ransom.py that represents letters in other ways by combining ASCII characters, such as the following. Feel free to make up your own substitutions. Be sure to update your tests.

```
A    4       K    |<
B    |3      L    |_
```

```
C   (       M   |\/|
D   |)      N   |\|
E   3       P   |`
F   |=      S   5
G   (-      T   +
H   |-|     V   \/
I   1       W   \/\/
J   _|
```

Summary



- Whenever you have lots of things to process, try to think about how you'd process just one of them.
- Write a test that helps you imagine how you'd like to use the function to process one item. What will you pass in, and what do you expect back?
- Write your function to pass your test. Be sure to think about what you'll do with both good and bad input.
- To apply your function to each element in your input, use a `for` loop, a list comprehension, or a `map()`.