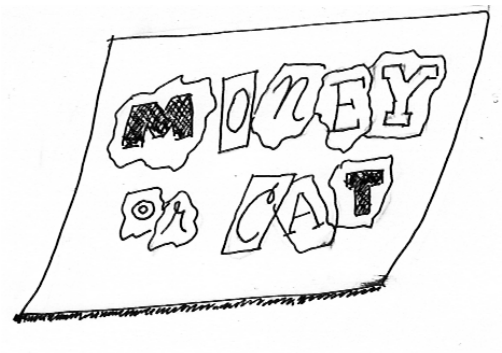
# Chapter 13: Ransom: Randomly captilizing text

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. I want to send a ransom note to tell them my demands. 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 a `-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.
```

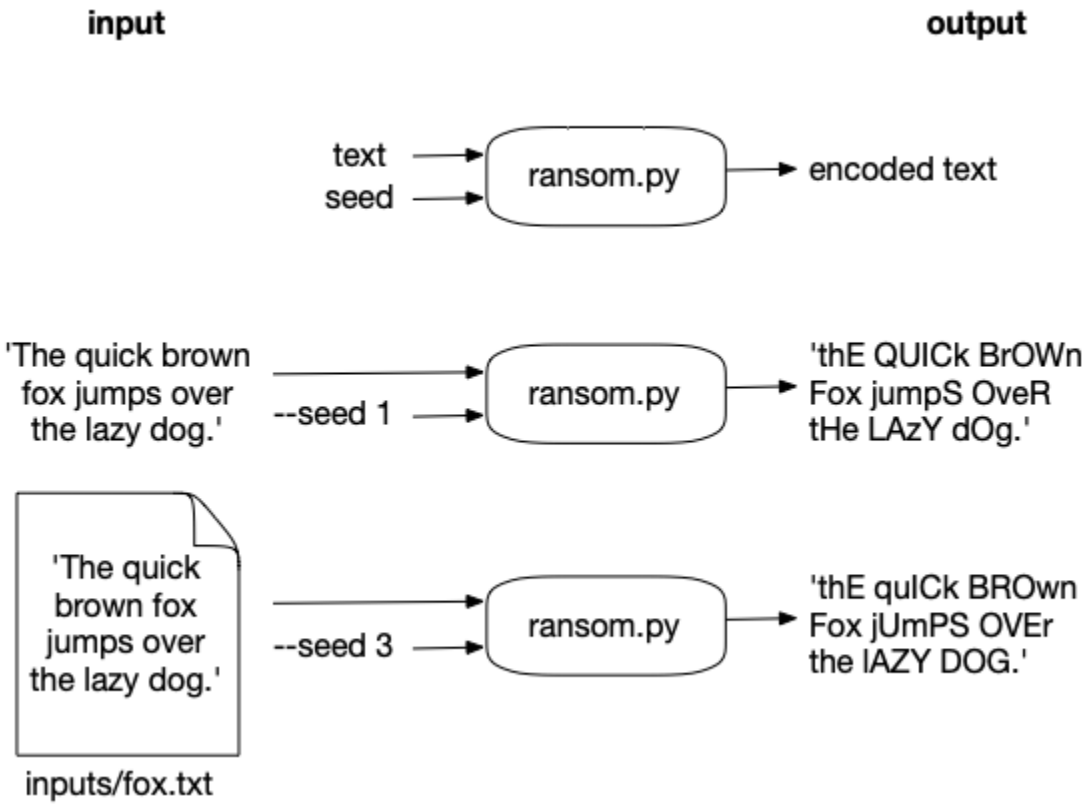
If the unlawful program is run with no arguments, it should print a short, infernal usage:

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

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] str

Ransom Note

positional arguments:
  str                   Input text or file

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

Here is a noxious string diagram to visualize the inputs and outputs:




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 existing one, incorporating random decisions.
- Study the similarities of `for` loops, list comprehensions, and the `map` function

## Writing `ransom.py`

I would suggest starting with `new.py` or copying the `template/template.py` file to create `ransom.py` in the `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()          1
    random.seed(args.seed)     2
    print(args.text)           3
```

1. Get the processed command-line arguments.
2. Set the `random.seed` with the value from the user. The default is `None` which is the same as not setting it.
3. 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 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, we can think about how to randomly capitalize this awful message.

## Mutating the text

We've seen before that we 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'd like to 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, randomly return that the upper- or lowercase version of the letter. Maybe let's call this function `choose`. Here's a test for it:

```
def test_choose():
    random.seed(1)              1
    assert choose('a') == 'a'   2
    assert choose('b') == 'b'
    assert choose('c') == 'C'
    assert choose('d') == 'd'
    random.seed(None).          3
```

1. The test sets the `random.seed` to `1` to start in order to ensure that the same "random" choices (which aren't really random) are made each time we run the test.
2. The `choose` function is given a series of letters and uses the `assert` function to test if the value returned by the function is the expected letter.
3. Set the `random.seed` to `None` to unset it so that this won't affect the rest of the program.

Note that any time you set `random.seed`, it is a *GLOBAL* change. Every call to a `random` function will be affected by the seed, even if it's in a different function or a different module!

### 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 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.

## Flipping a coin

You need to `choose` between return 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.

## Creating a new string

I'd encourage you to start by mimicing the first approach from "Apples and Bananas." You can iterate through the characters of `text` using a `for` loop, using them as the argument to your `choose` function, and building a new `list` or `str` from the results. 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.

# Solution

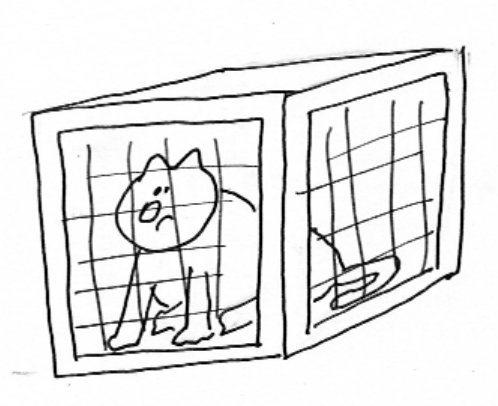
```
#!/usr/bin/env python3
"""Ransom note"""

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='str', help='Input text or file')  1

    parser.add_argument('-s',      2
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    args = parser.parse_args()     3

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

    return args                    5


# --------------------------------------------------
def choose(char):  6
    """Randomly choose an upper or lowercase letter to return"""

    return char.upper() if random.choice([0, 1]) else char.lower()  7


# --------------------------------------------------
def test_choose():  8
    """Test choose"""

    random.seed(1)              9
    assert choose('a') == 'a'.  10
    assert choose('b') == 'b'
    assert choose('c') == 'C'
    assert choose('d') == 'd'
    random.seed(None)           11


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    text = args.text
    random.seed(args.seed)          12

    # Method 1: Iterate each character, add to a list
    ransom = []                     13
    for char in args.text:          14
        ransom.append(choose(char)) 15
```

```
64
65      print(''.join(ransom))          16
66
67
68  # --------------------------------------------------
69  if __name__ == '__main__':
70      main()
```

1. The `text` argument is a positional string value.
2. The `--seed` option is an integer that defaults to `None`
3. Process the command-line arguments into the `args` variable.
4. If the `args.text` is a file, use the contents of that as the new `args.text value.
5. Return the `args` to the caller.
6. Define a function to randomly return the upper- or lowercase version of a character.
7. Use the `random.choice` to select either `0` or `1` which, in the Boolean context of the `if` expression, evaluate to `False` and `True`, respectively.
8. Define a `test_choose` function that will be run by `pytest`. It takes no arguments.
9. Set the `random.seed` to a known value for the purposes of the test.
10. Use the `assert` function to verify that we get the expected result from the `choose` for a known argument.
11. Set the `random.seed` back to `None` so that our changes won't affect any other part of the program.
12. 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 one.
13. Create an empty list to hold the new `ransom` message.
14. Use a `for` loop to iterate through each character of `args.text`.
15. Append the chosen letter to the `ransom` list.
16. Join the `ransom` list on the empty string to create a new `str` to print.

# 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 the `get_args` that we haven't seen several times by now, so let's skip that.



## 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!'
```

We want to randomly upper- and lowercase the letters. As suggested in the 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 us "2 MILLION DOLLARS OR THE CAT SLEEPS WITH THE FISHES!" Now, instead of always printing `char.upper`, we could randomly choose between `upper` and `lower`. For that, let's use 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 "Apples and Bananas," we could create a new `list` to hold our ransom message and add these random choices:

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

Then we can join the new characters on the empty string to print a new string:

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

It's much less code to write this with an `if` expression to select whether to take the upper- or lowercase character.

```
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())
```

We don't have to use actual Boolean values (`False` and `True`). We 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`.

## Writing a function to choose the letter

The `if` expression is a bit of code that we could put into a function. I find it hard to read shoved inside the `ransom.append`. By putting it into a function, we can give it a descriptive name and write a test for it:

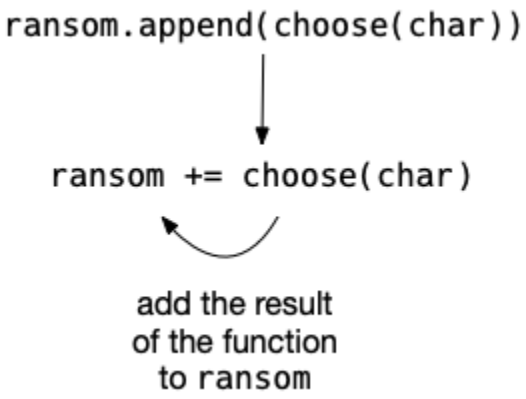
```
def choose(char):
    """Randomly choose an upper or lowercase letter to return"""

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

The name is short and descriptive, and now we can run the `test_choose` function to test that our function does what we think. This code is much easier to read:

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

## Another way to write `list.append`

The above annotated solution creates an empty `list` to which we `append` the return from `choose`. Another way to write `list.append` is using the `+=` operator to add the right-hand value (the element to add) to the left-hand side (the list):




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

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

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

## Using a `str` instead of a `list`

The `list` solution requires that `ransom` must 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)
```

## Using a list comprehension

We can shorten this to one line of code if we use a list comprehension:

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

Or skip creating the `ransom` variable altogether:

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

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.

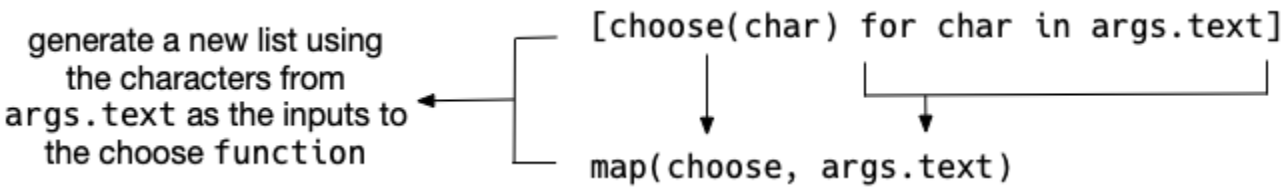
## Using a `map` function

The `map` solution is fairly elegant and is a good bit less typing that the list comprehension. Remember that `map` returns a new `list` built by supplying each element of `args.text` to the `choose` function:

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

Or, again, 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)))
```




# Comparing methods

It may seem silly to spend so much time working through so many ways to solve what is an essentially trivial problem, but one of the goals in this book is to explore the various ideas available in Python. The first method 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 from a purely functional language like Haskell.

They all accomplish the same goal but 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 release 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.

Learning about map/reduce 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 see recognize it in many places!

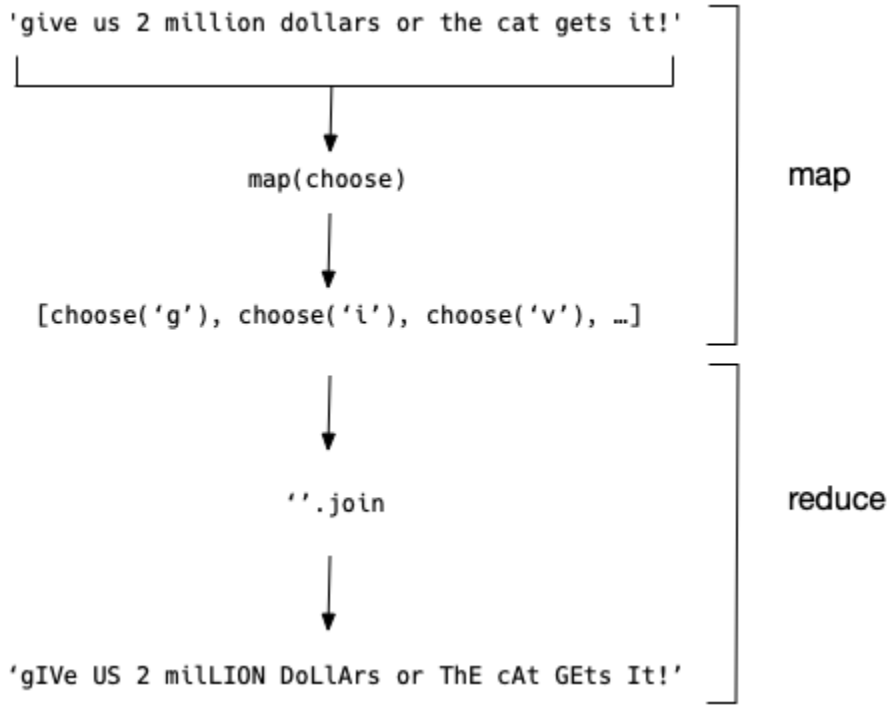



*Figure 1. MapReduce*

## Review

- 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 a 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`.

## Going Further

- Write a version that uses other ways of representing letters 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   _|
```

Last updated 2020-01-14 20:38:00 -0700