Chapter 11: Telephone: Randomly mutating strings

> *"What we have here is a failure to communicate." — Captain*

Now that we've played with randomness, let's apply the idea to randomly mutating a string. This is interesting because strings are actually *immutable* in Python, so we'll have to figure out a way around that. To explore these ideas, we'll write a version of the game of "Telephone." In that game, a secret message is whispered through a line or circle of people. Each time the message is transmitted, it's usually changed in some way. When the last person receives the message, they say it out loud to compare it to the original message.

We will write a program called `telephone.py` that will mimic this game. It will mutate some `text` by some varying amount and then print "You said: " plus the original text followed by "I heard: " with the modified message. The text may come from the command line:

```
$ ./telephone.py 'The quick brown fox jumps over the lazy dog.'
You said: "The quick brown fox jumps over the lazy dog."
I heard : "TheMquick brown fox jumps ovMr t:e lamy dog."
```

Or from a file:

```
$ ./telephone.py ../inputs/fox.txt
You said: "The quick brown fox jumps over the lazy dog."
I heard : "The quick]b'own fox jumps ovek the la[y dog."
```

In this exercise, you will learn to:

- Round numbers
- Use the `string` module
- Modify strings and lists to introduce random point mutations

More briefing

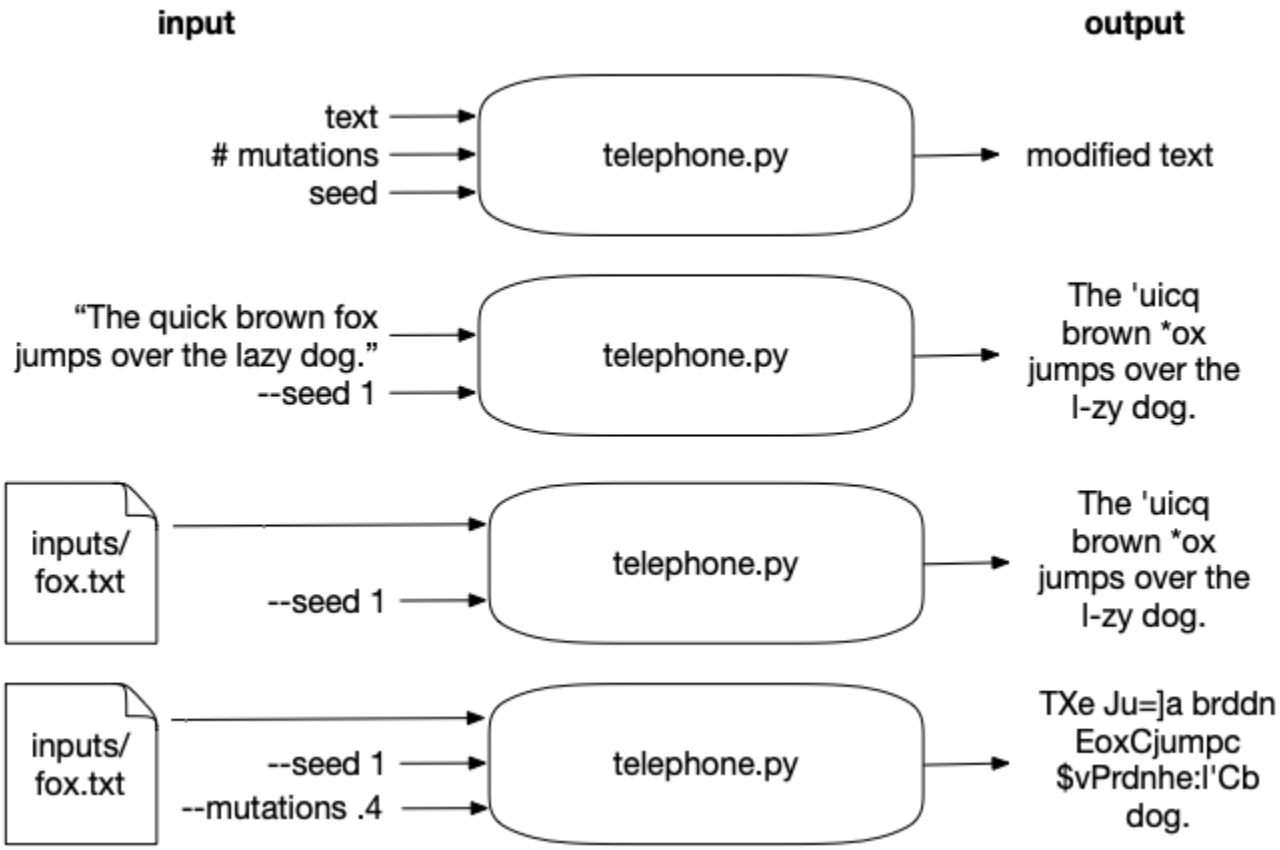
The program should accept a `-m` or `--mutations` option which should be a number between 0 and 1 with a default value of `0.1` (10%). This will be a percentage of the number of letters that should be altered. For instance, `.5` means that 50% of the letters should be changed:

```
$ ./telephone.py ../inputs/fox.txt -m .5
You said: "The quick brown fox jumps over the lazy dog."
I heard : "F#eYquJsY ZrHnna"o. Muz/$ Nver t/Relazy dA!."
```

Because we are using the `random` module, we'll accept an `int` value for the `-s` or `--seed` option so that we can reproduce our "random" selections:

```
$ ./telephone.py ../inputs/fox.txt -s 1
You said: "The quick brown fox jumps over the lazy dog."
I heard : "The 'uicq brown *ox jumps over the l-zy dog."
```

Here is a string diagram of the program:



Writing `telephone.py`

I recommend you start by with `new.py telephone.py` to create the program in the `telephone` directory. You could also copy the `template/template.py` to `telephone/telephone.py`. Modify the `get_args` function until your `-h` output matches the following:

```
$ ./telephone.py -h
usage: telephone.py [-h] [-s int] [-m float] str

Telephone

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)
  -m float, --mutations float
                        Percent mutations (default: 0.1)
```

Now run the test suite. You should pass at least the first two tests (the `telephone.py` program exists and prints something like a "usage" when run with `-h` or `--help`).

The next two tests check that your `--seed` and `--mutations` options both reject non-numeric values. This should happen automatically if you define these parameter using `type` of `int` and `float`, respectively. Your program should behave like this:

```
$ ./telephone.py -s blargh foo
usage: telephone.py [-h] [-s int] [-m float] str
telephone.py: error: argument -s/--seed: invalid int value: 'blargh'
$ ./telephone.py -m blargh foobar
usage: telephone.py [-h] [-s str] [-m float] str
telephone.py: error: argument -m/--mutations: invalid float value: 'blargh'
```

The next test checks if the program rejects `--mutations` outside of the range 0-1 (where both bounds are inclusive). This is not a check that you can easily describe to `argparse`, so I suggest that you look at how we handled the validation of the arguments in the `abuse.py` program. In the `get_args` function of that program, we manually checked the value of the arguments and used the `parser.error` function to throw an error. Your program should do this:

```
$ ./telephone.py -m -1 foobar
usage: telephone.py [-h] [-s str] [-m float] str
telephone.py: error: --mutations "-1.0" must be between 0 and 1
```

Note that a `--mutations` value of `0` is acceptable, in which case we will print the input text back without modifications. This is another program that accepts the input text either from the command line directly or from a file. I suggest you look at the "Howler" solution where, inside the `get_args` function, I suggest that you use `os.path.isfile` to detect if the text argument is a file. If it is a file, read the contents of the file for the `text` value.

Once you have taken care of all the program parameters, start off your `main` function with setting the `random.seed` and echoing back the given text:

```
def main():
    args = get_args()
    random.seed(args.seed)
    print(f'You said: "{args.text}"')
    print(f'I heard : "{args.text}"')
```

Your program should handle command-line text:

```
$ ./telephone.py 'The quick brown fox jumps over the lazy dog.'
You said: "The quick brown fox jumps over the lazy dog."
I heard : "The quick brown fox jumps over the lazy dog."
```

Or an input file:

```
$ ./telephone.py ../inputs/fox.txt
You said: "The quick brown fox jumps over the lazy dog."
I heard : "The quick brown fox jumps over the lazy dog."
```

At this point, your code should pass up to `test_for_echo`. The next tests start asking you to mutate the input, so let's discuss how to do that.

Calculating the number of mutations

The number of letters that need to be changed can be calculated by mutiplying the length of the input text by the `args.mutations` value. If we want to change 20% of the characters in "[t]he quick brown fox..." string, we'll find that is not a whole number:

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
>>> mutations = .20
>>> len(text) * mutations
8.8
```

We can use the `round` function to give us the nearest integer value. Read `help(round)` to understand how to round floating point numbers to a specific number of digits:

```
>>> round(len(text) * mutations)
9
```

Note that you could also convert a `float` to an `int` by using the `int` function, but this truncates the fractional part of the number rather than rounding it:

```
>>> int(len(text) * mutations)
8
```

You will need this value for later:

```
>>> num_mutations = round(len(text) * mutations)
>>> assert num_mutations == 9
```

The mutation space

When we change a character, what will we change it to? For this, I'd like to introduce you to the `string` module. I'd encourage you to take a look at the documentation by importing the module and reading the `help`:

```
>>> import string
>>> help(string)
```

We can, for instance, get all the lowercase ASCII letters. Note that this is not a method call as there are no parentheses `()` at the end:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

Note that this returned a `str`:

```
>>> type(string.ascii_lowercase)
<class 'str'>
```

For our program, I would like to use `string.ascii_letters` and `string.punctuation`. How do we concatenate two strings together? We can use the `+` operator. These are the characters we will draw from when replacing:

```
>>> alpha = string.ascii_letters + string.punctuation
>>> alpha
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Selecting the characters to mutate

There are at least two approaches we could take to choosing which characters to change.

Non-deterministic selection

One way would be to mimic solution 1 in "Apples". We could iterate `for char in text` and flip a coin to decide whether to take the original character or randomly select a new character:

```
new_text = ''  1
for char in args.text:  2
    new_text += random.choice(alpha) if random.random() <= args.mutations else char  3
print(new_text)  4
```

1. Initialize `new_text` as an empty string.
2. Iterate through each character in the text.
3. Use `random.random` to generate a value between 0 and 1. If that value is less than or equal to the `args.mutation` value, then randomly choose from the `alpha`; otherwise, use the character.
4. Print the resulting `new_text`.

We used the `random.choice` function in `abuse.py` to randomly select *one* value from a list of choices. Here we select a character from the `alpha` if the `random.random` value falls within the range of the `args.mutation` value (which we know is also a `float`).

The problem with this approach is that is is *non-deterministic*. That is, by the end of the `for` loop, we are not guaranteed of having made the correct number of changes, which we calculated above as `num_mutations`. If the mutation value was 40%, you could expect that you'd end up changing about 40% of the string because a random value between 0 and 1 should be less than or equal to 0.4 about 40% of the time. However, since the decision whether to change each character is random, you might end up with more or less mutation than this, especially with a short text. Since we can't be certain — since we can't determine beforehand — that we will end up with exactly 40% mutations, this is *non-deterministic*.

Still, this is a really useful technique that you should note. Imagine you have an input file with millions to potentially billions of lines of text and you want to randomly sample approximately 10% of the lines. The above approach would be a reasonably fast and accurate way to select which lines to take and which to discard. A larger sample size helps to get closer to the desired number of mutations.

Randomly sampling characters

A deterministic approach to the million-line file would require first reading the entire input to count the number of lines. Then you would have to choose which lines to take, and then you would go back through the file a second time to take those lines. This approach would take *much* longer than the method described above. Depending on how large the input

file is, how the program is written, and how much memory your computer has, the program could possibly even crash your computer!

Because our input is rather small, we will use this algorithm because it has the advantage of being exact and testable. Rather than lines of text, though, we'll consider indexes of characters. We've seen the `str.replace` method (in "Apples") that allows us to change, for instance, all instances of one string to another:

```
>>> 'foo'.replace('o', 'a')
'faa'
```

We can't use that method here, however, because we only want to change some characters. Instead we need to use the indexes of the characters in the text. We can use the `random.sample` function to select the indexes for us.

The first argument to `random.sample` needs to be something like a `list`. We should give it a `range` of numbers up to the length of our `text`. So, if our `text` is 44 characters long:

```
>>> text
'The quick brown fox jumps over the lazy dog.'
>>> len(text)
44
```

We can use the `range` function to make a `list` of numbers up to 44. Note that, in the REPL, we see that `range` is a lazy function:

```
>>> range(len(text))
range(0, 44)
```

It won't actually produce the 44 values until we force it, which we can do in the REPL using the `list` function. You rarely have to do this in your actual programs, however:

```
>>> list(range(len(text)))
```

We calculated above that the `num_mutations` for altering 20% of `text` is 9. Here are the indexes that need to be changed:

```
>>> indexes = random.sample(range(len(text)), num_mutations)
>>> indexes
[13, 6, 31, 1, 24, 27, 0, 28, 17]
```

I suggest you use a `for` loop to iterate through each of these index values:

```
>>> for i in indexes:
...     print(f'{i:2} {text[i]}')
...
13 w
 6 i
31 t
 1 h
24 s
27 v
 0 T
28 e
17 o
```

You should replace the character at that position with a randomly selected character from `alpha`.

```
>>> for i in indexes:
...     print(f'{i:2} {text[i]} -> {random.choice(alpha)}')
...
13 w -> b
 6 i -> W
31 t -> B
 1 h -> #
24 s -> d
27 v -> :
 0 T -> C
28 e -> %
17 o -> ,
```

I will introduce one other twist — we don't want the replacement value to ever be the same as the character it is replacing. Can you figure out how to get a subset of `alpha` that *does not* include the character at the position?

Mutating a string

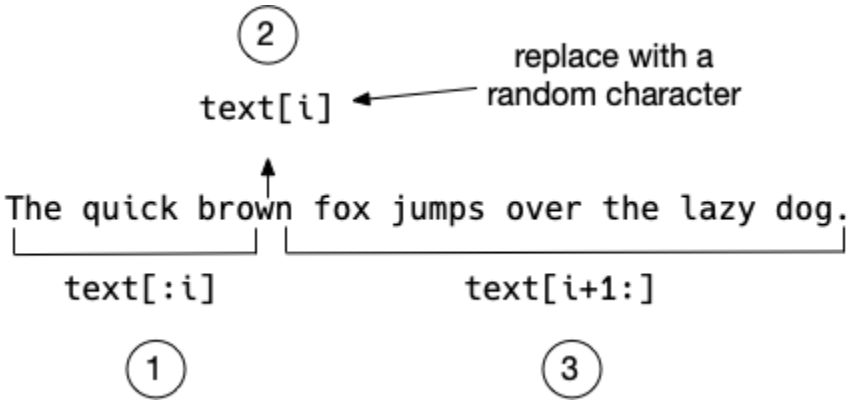
Python `str` variables are *immutable*, meaning we cannot directly modify them. For instance, we want to change the character `'w'` at position `13` to a `'b'`. It would be handy to directly modify `text[13]`, but that will create an exception:

```
>>> text[13] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

The only way to modify the `str` value `text` is to overwrite with a new `str`. So, you need to create a new `str` with

1. the part of `text` before a given index
2. the randomly selected value from `alpha`
3. the part of `text` after a given index

For 1 and 3, you can use string *slices*.



For example, if the index `i` is 13, the slice before it is:

```
>>> text[:13]
'The quick bro'
```

And the part after:

```
>>> text[14:]
'n fox jumps over the lazy dog.'
```

Using the 1, 2, and 3 above, your `for` loop should be:

```
for i in index:
    text = 1 + 2 + 3
```

Can you figure that out?

Time to write

OK, the lesson is over. You have to go write this now. Use the tests. Solve them one at a time. You can do this.

Solution

```
#!/usr/bin/env python3
"""Telephone"""

import argparse
import os
import random
import string  1


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

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

    parser.add_argument('text', metavar='str', help='Input text or file')  2

    parser.add_argument('-s',                           3
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    parser.add_argument('-m',                           4
                        '--mutations',
                        help='Percent mutations',
                        metavar='float',
                        type=float,
                        default=0.1)

    args = parser.parse_args()                          5

    if not 0 <= args.mutations <= 1:                    6
        parser.error(f'--mutations "{args.mutations}" must be between 0 and 1')

    if os.path.isfile(args.text):                       7
        args.text = open(args.text).read().rstrip()

    return args                                         8


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

    args = get_args()
    text = args.text
    random.seed(args.seed)                              9
    alpha = string.ascii_letters + string.punctuation   10
    len_text = len(text)                                11
    num_mutations = round(args.mutations * len_text)    12
    new_text = text                                     13

    for i in random.sample(range(len_text), num_mutations):       14
        new_char = random.choice(alpha.replace(new_text[i], ''))  15
        new_text = new_text[:i] + new_char + new_text[i + 1:]     16

    print(f'You said: "{text}"\nI heard : "{new_text}"')          17


```

```
64  # --------------------------------------------------
65  if __name__ == '__main__':
66      main()
```

1. We need to import the `string` module here.
2. The program takes one positional argument for the `text`. This could be either a string of text or a file which needs to be read.
3. The `--seed` parameter is an `int` value with a default of `None`.
4. The `--mutations` parameter must be a `float` with a default of `0.1`.
5. Here we gather the `args` from the command line. If `argparse` detects failures such as non-numeric values for `seed` or `mutations`, then the program dies here and the user sees an error message. If this call succeeds, then `argparse` has validated the arguments and converted the values, e.g., `args.mutations` is actually a `float` value.
6. If `args.mutations` is *not* in the acceptable range of 0-1, then use `parser.error` to halt the program and print the given message. Note the use of feedback to echo the bad `args.mutation` value to the user.
7. If `args.text` names an existing file, then read that file for the contents, overwriting the original value of `args.text`.
8. Return `args` to the caller (line 49).
9. Be sure to set the `random.seed` to the value provided by the user. Remember that the default value for `args.seed` is `None`, which is the same as not setting the seed.
10. Set `alpha` to be the characters we'll use for replacements.
11. Since I use the `len(text)` more than once, I put it into a variable.
12. Figure the `num_mutations`.
13. Make copy of `text`.
14. Use `random.sample` to get `num_mutations` indexes to change. That function returns a `list` that we can iterate using the `for` loop.
15. Select a `new_char` using `random.choice` to select from a string we create by replacing in `alpha` the current character (`text[i]`) with nothing. This ensures that our new character cannot be the same as the one we are replacing.
16. Overwrite the `text` by concatenating the slice before the current index, the `new_char`, and the slice after the current index.
17. Print the `text`.

Discussion

Defining the arguments

There's nothing in `get_args` that we haven't seen before. The `--seed` argument is an `int` that we will pass to the `random.seed` function so as to control the randomness for testing. The `default` should be `None` so that we can call `random.seed(args.seed)` where `None` is the same as not setting it. (Note that we could also define the seed as a `str` value as both are acceptable seeds.) The `--mutations` is a `float` with a reasonable default, and I test that `args.mutations` is in the proper range and use `parser.error` if it is not. As in other programs, I test if the `text` argument is a file and read the contents if it is.

Mutating a string

Now we saw earlier that we can't just change the `text`:

```
>>> text[13] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

So we're going to have to create a *new* string using the text before and after `i` which we can get with string slices using `text[start:stop]`. If you leave out `start`, Python starts at `0` (the beginning of the string), and if you leave out `stop` then it goes to the end, so `text[:]` is a copy of the entire string.

If `i` is `13`, then bit before `i` is:

```
>>> i = 13
>>> text[:i]
'The quick bro'
```

And the bit after `i + 1`:

```
>>> text[i+1:]
'n fox jumps over the lazy dog.'
```

Now for what to put in the middle. I noted that we should use `random.choice` to select a character from `alpha`, which is the combination of all the ASCII letters and punctuation *without* the current character. I can use the `str.replace` method to get rid of the current letter:

```
>>> alpha = string.ascii_letters + string.punctuation
>>> alpha.replace(text[i], '')
'abcdefghijklmnopqrstuvxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

And use that to get a new letter that won't include what it's replacing:

```
>>> new_char = random.choice(alpha.replace(text[i], ''))
>>> new_char
'E'
```

There are many ways to join strings together into new strings, and the `+` operator is perhaps the simplest:

```
>>> text = text[:i] + new_char + text[i+1:]
>>> text
'The quick broEn fox jumps over the lazy dog.'
```

We do this for each index in the `random.sample` of indexes, each time overwriting `text`. After the `for` loop is done, we have mutated all the positions of the input string and can `print` it.

Using a `list` instead of a `str`

Strings are immutable, but lists are not. We see above that a move like `text[13] = 'b'` creates an exception, but we can change `text` into a list and directly modify it with the same syntax:

```
>>> text = list(text)
>>> text[13] = 'b'
```

And then we can turn that `list` back into a `str` by joining it on the empty string:

```
>>> ''.join(text)
'The quick brobn fox jumps over the lazy dog.'
```

Here is a version of `main` that uses this approach:

```
def main():
    args = get_args()
    text = args.text
    random.seed(args.seed)
    alpha = string.ascii_letters + string.punctuation
    len_text = len(text)
    num_mutations = round(args.mutations * len_text)
    new_text = list(text)  1

    for i in random.sample(range(len_text), num_mutations):
        new_text[i] = random.choice(alpha.replace(new_text[i], ''))  2

    print('You said: "{}"\nI heard : "{}"'.format(text, ''.join(new_text)))  3
```

1. Set the `new_text` to be a `list` of characters from the `text` value.
2. Now you can directly modify a value in `new_text`.
3. Join `new_list` on the empty string to make a new `str`.

There's no particular advantage of one way over the other, but I would personally choose the second method because I don't like messing around with slicing strings. To me, modifying a `list` in-place makes much more sense than repeatedly chopping up and piecing together a `str`.

Mutations in DNA

For what it's worth, this is (kind of, sort of) how DNA changes over time. The machinery to copy DNA makes mistakes, and mutations randomly occur. Many times the change has no deleterious affect on the organism. Our example only changes characters to other characters, what are called "point mutations" or "single nucleotide variations" (SNV) or "single nucleotide polymorphisms" (SNP) in biology, but we could write a version that would also randomly delete or insert new characters which are called "in-dels" (insertion-deletions) in biology. Mutations (that don't result in the demise of the organism) occur at a fairly standard rate, so counting the number of mutations between a conserved region of any two organisms can allow an estimate of how long ago they diverged from a common ancestor!

Review

- A string cannot be directly modified, but the variable containing the string can be repeatedly overwritten with new values.
- Lists can be directly modified, so it can sometimes help to use `list` on a string to turn it into a `list`, modify that, then use `''.join` to change it back to a `str`.

- The `string` module has handy functions for dealing with strings.

Going Further

- Apply the mutations to randomly selected words instead of the whole string.
- Add insertions and deletions in addition to mutations; maybe create arguments for the percentage of each and choose to add or delete characters at the indicated frequency.
- Add an option for `-o` or `--output` that names a file to write the output. The default should be to print to `STDOUT`.
- Add an flag to limit the replacements to character values only (no punctuation).
- Add tests to `test.py` for every new feature and ensure your program works properly.

Last updated 2020-01-14 20:37:53 -0700