10 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. We'll have to figure out a way around that.

To explore these ideas, we'll write a version of the game of Telephone where a secret message is whispered through a line or circle of people. Each time the message is transmitted, it's usually changed in some unpredictable way. The last person to receive the message will say it out loud to compare it to the original message. Often the results are nonsensical and possibly comical.

We will write a program called telephone.py that will mimic this game. It will print "You said: " and the original text, followed by "I heard: " with a modified version of the message. As in chapter 5, the input 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 it may come 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."
```

The program should accept an `-m` or `--mutations` option, which should be a floating-point number between 0 and 1 with a default value of `0.1` (10%). This will be the 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 pseudo-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."
```

Figure 10.1 shows a string diagram of the program.

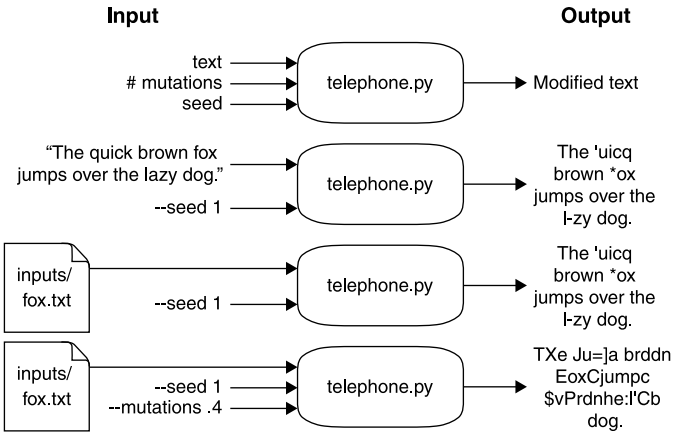


Figure 10.1 The telephone program will accept text and possibly some percentage of mutations, along with a random seed. The output will be a randomly mutated version of the input text.

In this exercise, you will learn to

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

10.1 Writing telephone.py

I recommend you use the new.py program to create a new program called telephone.py in the 10_telephone directory. You could do this from the top level of the repository like so:

```
$ ./bin/new.py 10_telephone/telephone.py
```

You could also copy template/template.py to 10_telephone/telephone.py. Modify the `get_args()` function until your `-h` output matches the following. I would recommend you use `type=float` for the mutations parameter:

```
$ ./telephone.py -h
usage: telephone.py [-h] [-s seed] [-m mutations] text

Telephone

positional arguments:
  text                  Input text or file

optional arguments:
  -h, --help            show this help message and exit
  -s seed, --seed seed  Random seed (default: None)
  -m mutations, --mutations mutations
                        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 a usage statement 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 parameters using the `int` and `float` types, respectively. That is, your program should behave like this:

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

The next test checks if the program rejects values for `--mutations` outside the range 0–1 (where both bounds are inclusive). This is not a check that you can easily describe to `argparse`, so I suggest you look at how we handled the validation of the arguments in abuse.py in chapter 9. 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. Note that a `--mutations` value of `0` is acceptable, in which case we will print out the input text without modifications. Your program should do this:

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

This is another program that accepts input text either from the command line or from a file, and I suggest you look at the solution in chapter 5. Inside the `get_args()` function, you can use `os.path.isfile()` to detect whether 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 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."
```

And it should handle 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.

10.1.1 Calculating the number of mutations

The number of letters that need to be changed can be calculated by multiplying the length of the input text by the `args.mutations` value. If we want to change 20% of the characters in "The quick brown fox…" string, we'll find that it 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 learn 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, so let's save it in a variable:

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

10.1.2 The mutation space

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

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

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

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

This returns a `str`:

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

For our program, we can use `string.ascii_letters` and `string.punctuation` to get strings of all the letters and punctuation. To concatenate the two strings together, we can use the `+` operator. We'll draw from this string to randomly select a character to replace another:

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

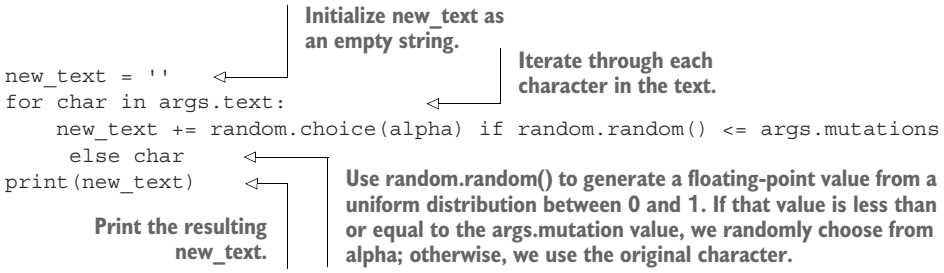
Note that even if we both use the same random seed, you and I will get different results if our letters are in a different order. To ensure our results match, we'll both need to sort the `alpha` characters so they are in a consistent order.

10.1.3 Selecting the characters to mutate

There are at least two approaches we could take to choosing which characters to change: a *deterministic approach* where the results are always guaranteed to be the same and a *non-deterministic approach* where we employ chance to get close to a target. Let's examine the latter one first.

NON-DETERMINISTIC SELECTION

One way to choose the characters to change would be to mimic method 1 in chapter 8. We could iterate through each of the characters in our text and select a random number to decide whether to keep the original character or change it to some randomly selected value. If our random number is less than or equal to our mutations setting, we should change the character:



We used the `random.choice()` function in abuse.py in chapter 9 to randomly select *one* value from a list of choices. We can use it here to select a character from `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, by the end of the `for` loop, we are not guaranteed to have made exactly the correct number of changes. That is, we calculated that we should change 9 characters out of 44 when the mutation rate is 20%. We would expect to end up changing about 20% of the characters with this code, because a random value from a uniform distribution of values between 0 and 1 should be less than or equal to 0.2 about 20% of the time. Sometimes we might end up only changing 8 characters or other times we might change 10. Because of this uncertainty, this is approach would be considered *non-deterministic*.

Still, this is a really useful technique that you should note. Imagine you have an input file with millions or potentially billions of lines of text, and you want to randomly sample approximately 10% of the lines. The preceding approach would be reasonably fast and accurate. A larger sample size will help you 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, choosing which lines to take, and then going 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!

Our input is rather small, however, so we will use this algorithm because it has the advantages of being exact and testable. Rather than focusing on lines of text, though, we'll consider indexes of characters. You've seen the `str.replace()` method (in chapter 8), which allows us to change all instances of one string to another:

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

We can't use `str.replace()` because it will change every occurrence of some character, and we only want to change individual characters. Instead we can use the `random.sample()` function to select some indexes of the characters in the text. The first argument to `random.sample()` needs to be something like a `list`. We can give it a `range()` of numbers up to the length of our `text`.

Suppose 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:

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

Note that `range()` is a lazy function. It won't actually produce the 44 values until we force it, which we can do in the REPL using the `list()` function:

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

We calculated earlier that the `num_mutations` value for altering 20% of `text` is 9. Here is one selection of indexes that could 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 each index position with a randomly selected character from `alpha`:

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

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?

10.1.4 Mutating a string

Python `str` variables are *immutable*, meaning we cannot directly modify them. For instance, suppose 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 it with a new `str`. We need to create a new `str` with the following, as shown in figure 10.2:

- The part of `text` before a given index
- The randomly selected value from `alpha`
- The part of `text` after a given index

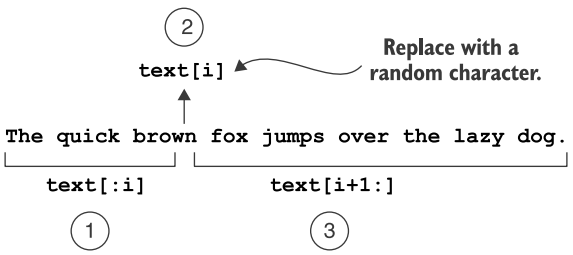


Figure 10.2 Create a new string by selecting the portion of the string up to the index, a new character, and the portion of the string after the index.

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'
```

The part after it is

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

Using the three parts listed earlier, your `for` loop should be

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

Can you figure that out?

10.1.5 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.

10.2 Solution

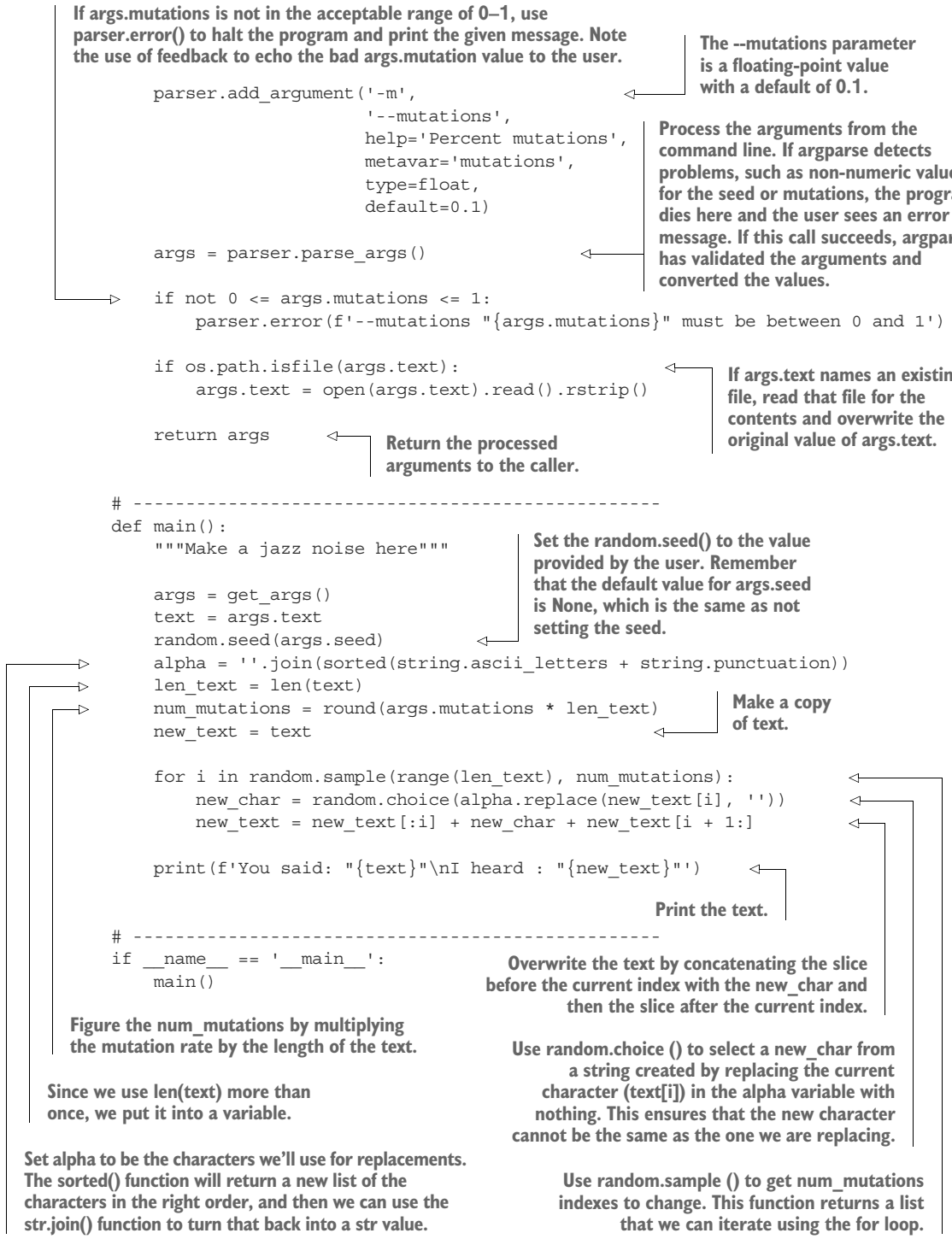
How different was your solution from mine? Let's look at one way to write a program that satisfies the tests:

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

import argparse
import os
import random
import string

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

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

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

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

Import the string module we'll need to select a random character.

Define a positional argument for the text. This could be either a string of text or a file that needs to be read.

The --seed parameter is an integer value with a default of None.


If args.mutations is not in the acceptable range of 0–1, 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.
The --mutations parameter is a floating-point value with a default of 0.1.
Process the arguments from the command line. If argparse detects problems, such as non-numeric values for the seed or mutations, the program dies here and the user sees an error message. If this call succeeds, argparse has validated the arguments and converted the values.
parser.add_argument('-m',
'--mutations',
help='Percent mutations',
metavar='mutations',
type=float,
default=0.1)
args = parser.parse_args()
if not 0 <= args.mutations <= 1:
parser.error(f'--mutations "{args.mutations}" must be between 0 and 1')
if os.path.isfile(args.text):
args.text = open(args.text).read().rstrip()
If args.text names an existing file, read that file for the contents and overwrite the original value of args.text.
return args
Return the processed arguments to the caller.
# --
def main():
"""Make a jazz noise here"""
args = get_args()
text = args.text
random.seed(args.seed)
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.
alpha = ''.join(sorted(string.ascii_letters + string.punctuation))
len_text = len(text)
num_mutations = round(args.mutations * len_text)
new_text = text
Make a copy of text.
for i in random.sample(range(len_text), num_mutations):
new_char = random.choice(alpha.replace(new_text[i], ''))
new_text = new_text[:i] + new_char + new_text[i + 1:]
print(f'You said: "{text}"\nI heard : "{new_text}"')
Print the text.
# --
if __name__ == '__main__':
main()
Overwrite the text by concatenating the slice before the current index with the new_char and then the slice after the current index.
Figure the num_mutations by multiplying the mutation rate by the length of the text.
Use random.choice () to select a new_char from a string created by replacing the current character (text[i]) in the alpha variable with nothing. This ensures that the new character cannot be the same as the one we are replacing.
Since we use len(text) more than once, we put it into a variable.
Use random.sample () to get num_mutations indexes to change. This function returns a list that we can iterate using the for loop.
Set alpha to be the characters we'll use for replacements. The sorted() function will return a new list of the characters in the right order, and then we can use the str.join() function to turn that back into a str value.

10.3 Discussion

There's nothing in `get_args()` that you 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 seed value is `None` so that we can call `random.seed(args.seed)` where `None` is the same as not setting it. The `--mutations` parameter is a `float` with a reasonable default, and we use `parser.error()` to create an error message if the value is not in the proper range. As in other programs, we test whether the `text` argument is a file and read the contents if it is.

10.3.1 Mutating a string

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

```
>>> text = 'The quick brown fox jumps over the lazy dog.'
>>> text[13] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

We 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`, it goes to the end, so `text[:]` is a copy of the entire string.

If `i` is `13`, the bit before `i` is

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

The bit after `i + 1` is

```
>>> 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 use the `str.replace()` method to get rid of the current letter:

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

Then I 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
'Q'
```

There are many ways to join strings together into new strings. The + operator is perhaps the simplest:

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

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

10.3.2 Using a list instead of a str

Strings are immutable, but lists are not. You've seen 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'
```

We can then 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 = ''.join(sorted(string.ascii_letters + string.punctuation))
    len_text = len(text)
    num_mutations = round(args.mutations * len_text)
    new_text = list(text)                                  # Initialize new_text as a list of the original text value.

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

    print('You said: "{}"\nI heard : "{}"'.format(text, ''.join(new_text)))   # Join new_list on the empty string to make a new str.
```

Now we can directly modify a value in new_text.

There's no particular advantage of one approach 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 program mimics (kind of, sort of) how DNA changes over time. The machinery to copy DNA makes mistakes, and mutations randomly occur. Often the change has no deleterious effect on the organism.

Our example only changes characters to other characters—what biologists call "point mutations," "single nucleotide variations" (SNV), or "single nucleotide polymorphisms" (SNP). We could instead write a version that would also randomly delete or insert new characters, which are called "in-dels" (insertion-deletions). 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.

10.4 Going further

- Apply the mutations to randomly selected words instead of the whole string.
- Perform 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 to. The default should be to print to `STDOUT`.
- Add a 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.

Summary

- 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 list, and then use `str.join()` to change it back to a `str`.
- The `string` module has handy definitions of various strings.