17 Mad Libs: Using regular expressions

When I was a wee lad, we used to play at Mad Libs for hours and hours. This was before computers, mind you, before televisions or radio or even paper! No, scratch that, we had paper. Anyway, point is we only had Mad Libs to play, and we loved it! And now you must play!

In this chapter, we'll write a program called mad.py that will read a file given as a positional argument and find all the placeholders in angle brackets, like `<verb>` or `<adjective>`. For each placeholder, we'll prompt the user for the part of speech being requested, like "Give me a verb" and "Give me an adjective." (Notice that you'll need to use the correct article, just as in chapter 2.) Each value from the user will then replace the placeholder in the text, so if the user says "drive" for the verb, then `<verb>` in the text will be replaced with `drive`. When all the placeholders have been replaced with inputs from the user, we'll print out the new text.

There is a 17_mad_libs/inputs directory with some sample files you can use, but I also encourage you to create your own. For instance, here is a version of the "fox" text:

```
$ cd 17_mad_libs
$ cat inputs/fox.txt
The quick <adjective> <noun> jumps <preposition> the lazy <noun>.
```

When the program is run with this file as the input, it will ask for each of the placeholders and then print the silliness:

```
$ ./mad.py inputs/fox.txt
Give me an adjective: surly
Give me a noun: car
Give me a preposition: under
Give me a noun: bicycle
The quick surly car jumps under the lazy bicycle.
```

By default, this is an interactive program that will use the `input()` prompt to ask the user for their answers, but for testing purposes we will have an `-i` or `--inputs` option so the test suite can pass in all the answers and bypass the interactive `input()` calls:

```
$ ./mad.py inputs/fox.txt -i surly car under bicycle
The quick surly car jumps under the lazy bicycle.
```

In this exercise, you will

- Learn to use `sys.exit()` to halt your program and indicate an error status
- Learn about greedy matching with regular expressions
- Use `re.findall()` to find all matches for a regex
- Use `re.sub()` to replace found patterns with new text
- Explore ways to write the solution without using regular expressions

17.1 Writing mad.py

To start off, create the program mad.py in the 17_mad_libs directory using new.py or by copying template/template.py to 17_mad_libs/mad.py. You would also do well to define the positional `file` argument as a readable text file using `type=argparse.FileType('rt')`. The `-i` or `--inputs` option should use `nargs='*'` to define a `list` of zero or more `str` values.

After this, your program should be able to produce a usage statement when given no arguments or the `-h` or `--help` flag:

```
$ ./mad.py -h
usage: mad.py [-h] [-i [input [input ...]]] FILE

Mad Libs

positional arguments:
  FILE                  Input file

optional arguments:
  -h, --help            show this help message and exit
  -i [input [input ...]], --inputs [input [input ...]]
                        Inputs (for testing) (default: None)
```

If the given `file` argument does not exist, the program should error out:

```
$ ./mad.py blargh
usage: mad.py [-h] [-i [str [str ...]]] FILE
mad.py: error: argument FILE: can't open 'blargh': \
[Errno 2] No such file or directory: 'blargh'
```

If the text of the file contains no `<>` placeholders, the program should print a message and exit with an error value (something other than `0`). Note that this error does not need to print a usage statement, so you don't have to use `parser.error()` as in previous exercises:

```
$ cat no_blanks.txt
This text has no placeholders.
$ ./mad.py no_blanks.txt
"no_blanks.txt" has no placeholders.
```

Figure 17.1 shows a string diagram to help you visualize the program.

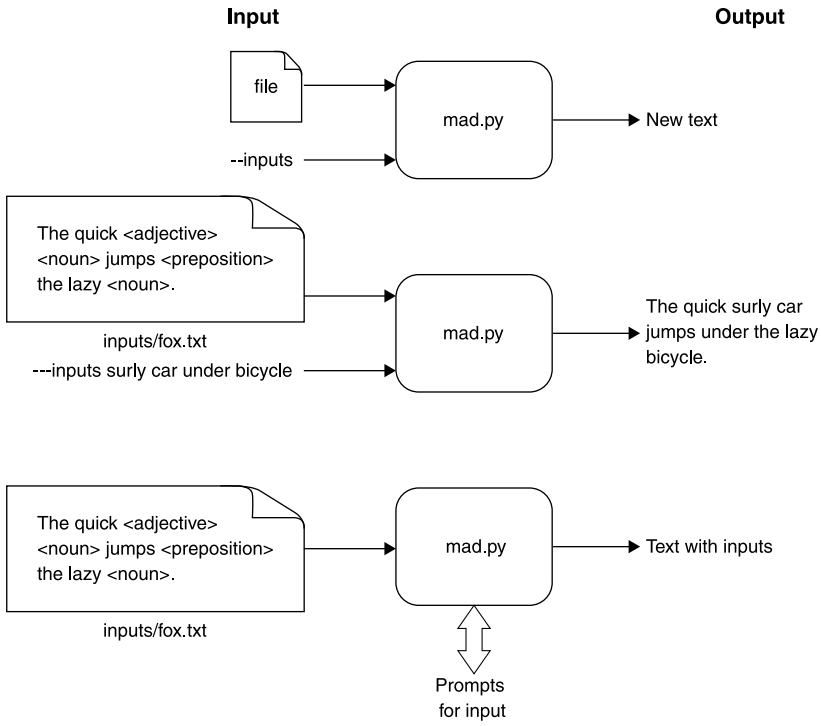


Figure 17.1 The Mad Libs program must have an input file. It may also have a list of strings for the substitutions or it will interactively ask the user for the values.

17.1.1 Using regular expressions to find the pointy bits

We've talked before about the possible dangers of reading an entire file into memory. Because we'll be parsing the text to find all the <…> bits in this program, we'll really need to read the whole file at once. We can do this by chaining the appropriate functions like so:

```
>>> text = open('inputs/fox.txt').read().rstrip()
>>> text
'The quick <adjective> <noun> jumps <preposition> the lazy <noun>.'
```

We're looking for patterns of text inside angle brackets, so let's use a regular expression. We can find a literal < character like so (see figure 17.2):

```
>>> import re
>>> re.search('<', text)
<re.Match object; span=(10, 11), match='<'>
```

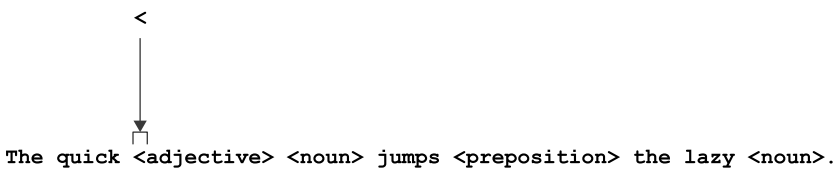


Figure 17.2 Matching a literal less-than sign

Now let's find that bracket's mate. The . in a regular expression means "anything," and we can add a + after it to mean "one or more." I'll capture the match so it's easier to see:

```
>>> match = re.search('(<.+>)', text)
>>> match.group(1)
'<adjective> <noun> jumps <preposition> the lazy <noun>'
```

As shown in figure 17.3, that matched all the way to the end of the string instead of stopping at the first available >. It's common when you use * or + for zero, one, or more for the regex engine to be "greedy" on the *or more* part. The pattern matches beyond where we wanted, but it is technically matching exactly what we described. Remember that . means *anything*, and a right angle bracket (or greater-than sign) is

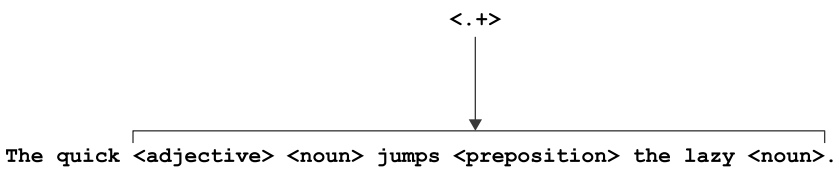


Figure 17.3 The plus sign to match one or more is a greedy match, matching as many characters as possible.

"anything." It matches as many characters as possible until it finds the last right angle bracket to stop at, which is why this pattern is called "greedy."

We can make the regex "non-greedy" by changing `+` to `+?` so that it matches the shortest possible string (see figure 17.4):

```
>>> re.search('<.+?>', text)
<re.Match object; span=(10, 21), match='<adjective>'>
```

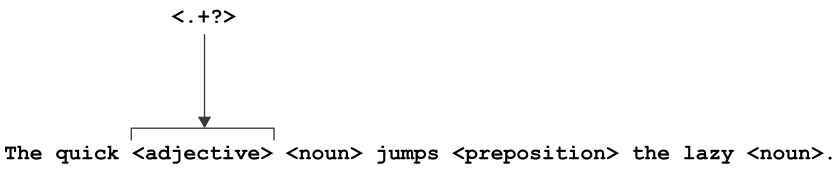


Figure 17.4 The question mark after the plus sign makes the regex stop at the shortest possible match.

Rather than using `.` for "anything," it would be more accurate to say that we want to match one or more of anything *that is not either of the angle brackets*. The character class `[<>]` would match either bracket. We can negate (or complement) the class by putting a caret (`^`) as the first character, so we have `[^<>]` (see figure 17.5). That will match anything that is not a left or right angle bracket:

```
>>> re.search('<[^<>]+>', text)
<re.Match object; span=(10, 21), match='<adjective>'>
```

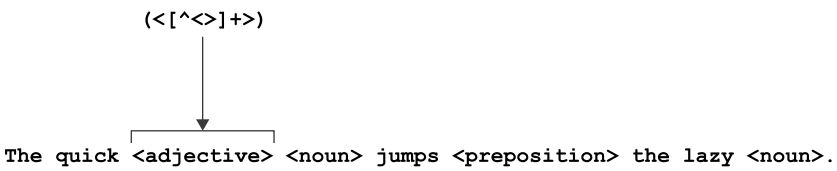


Figure 17.5 A negated character class to match anything other than the angle brackets

Why do we have both brackets inside the negated class? Wouldn't the right bracket be enough? Well, I'm guarding against *unbalanced* brackets. With only the right bracket, it would match this text (see figure 17.6):

```
>>> re.search('<[^>]+>', 'foo <<bar> baz')
<re.Match object; span=(4, 10), match='<<bar>'>
```

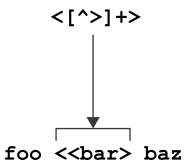


Figure 17.6 This regex leaves open the possibility of matching unbalanced brackets.

But with *both* brackets in the negated class, it finds the correct, balanced pair (see figure 17.7):

```
>>> re.search('<[^<>]+>', 'foo <<bar> baz')
<re.Match object; span=(5, 10), match='<bar>'>
```

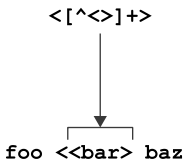


Figure 17.7 This regex finds the correctly balanced brackets and contained text.

We'll add two sets of parentheses `()`. The first will capture the *entire* placeholder pattern (see figure 17.8):

```
>>> match = re.search('(<([^<>]+)>)', text)
>>> match.groups()
('<adjective>', 'adjective')
```

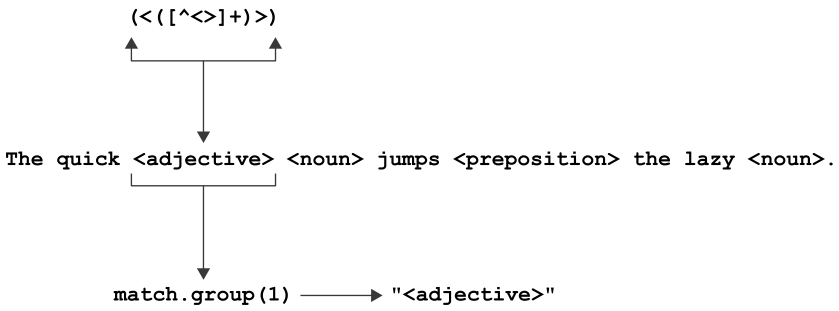


Figure 17.8 The outer parentheses capture the brackets and text.

The other is for the string *inside* the `<>` (see figure 17.9):

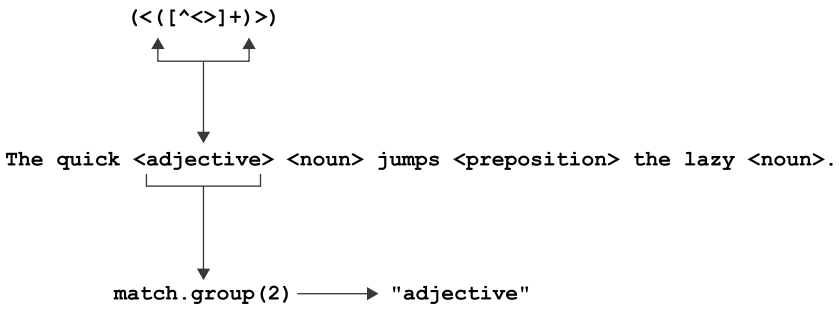


Figure 17.9 The inner parentheses capture just the text.

There is a very handy function called `re.findall()` that will return all matching text groups as a `list` of `tuple` values:

```
>>> from pprint import pprint
>>> matches = re.findall('(<([^<>]+)>)', text)
>>> pprint(matches)
[('<adjective>', 'adjective'),
 ('<noun>', 'noun'),
 ('<preposition>', 'preposition'),
 ('<noun>', 'noun')]
```

Note that the capture groups are returned in the order of their opening parentheses, so the entire placeholder is the first member of each `tuple`, and the contained text is the second. We can iterate over this `list`, *unpacking* each `tuple` into variables (see figure 17.10):

```
>>> for placeholder, name in matches:
...     print(f'Give me {name}')
...
Give me adjective
Give me noun
Give me preposition
Give me noun
```

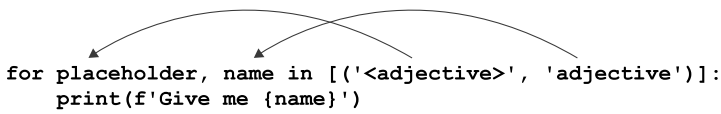


Figure 17.10 Since the list contains 2-tuples, we can unpack them into two variables in the `for` loop.

You should insert the correct article ("a" or "an," as you did in chapter 2) to use as the prompt for `input()`.

17.1.2 Halting and printing errors

If we find there are no placeholders in the text, we need to print an error message. It's common to print error messages to `STDERR` (standard error), and the `print()` function allows us to specify a `file` argument. We'll use `sys.stderr`, just as we did in chapter 9. To do that, we need to import that module:

```
import sys
```

You may recall that `sys.stderr` is like an already open file handle, so there's no need to `open()` it:

```
print('This is an error!', file=sys.stderr)
```

If there really are no placeholders, we should exit the program with an error value to indicate to the operating system that the program failed to run properly. The normal exit value for a program is `0`, as in "zero errors," so we need to exit with some `int` value that is *not* `0`. I always use `1`:

```
sys.exit(1)
```

One of the tests checks whether your program can detect missing placeholders and if your program exits correctly.

You can also call `sys.exit()` with a string value, in which case the string will be printed to `sys.stderr` and the program will exit with the value `1`:

```
sys.exit('This will kill your program and print an error message!')
```

17.1.3 Getting the values

For each one of the parts of speech in the text, we need a value that will come either from the `--inputs` argument or directly from the user. If we have nothing for `--inputs`, we can use the `input()` function to get an answer from the user.

The `input()` function takes a `str` value to use as a prompt:

```
>>> value = input('Give me an adjective: ')
Give me an adjective: blue
```

And it returns a `str` value of whatever the user typed before pressing the Return key:

```
>>> value
'blue'
```

If, however, we have values for the inputs, we can use those and not bother with the `input()` function. I'm only making you handle the `--inputs` option for testing purposes. You can safely assume that you will always have the same number of inputs as you have placeholders (see figure 17.11).

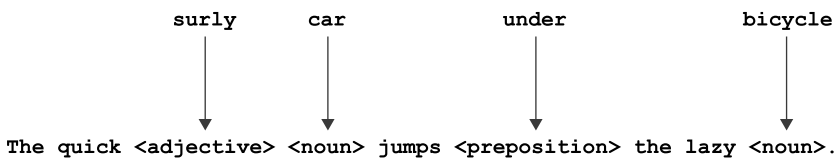


Figure 17.11 If given inputs from the command line, they will match up with the placeholders in the text.

For instance, you might have the following as the `--inputs` option to your program for the fox.txt example:

```
>>> inputs = ['surly', 'car', 'under', 'bicycle']
```

You need to remove and return the first string, "surly," from `inputs`. The `list.pop()` method is what you need, but it wants to remove the *last* element by default:

```
>>> inputs.pop()
'bicycle'
```

The `list.pop()` method takes an optional argument to indicate the index of the element you want to remove. Can you figure out how to make that work? Be sure to read `help(list.pop)` if you're stuck.

17.1.4 Substituting the text

When you have values for each of the placeholders, you will need to substitute them into the text. I suggest you look into the `re.sub()` (substitute) function, which will replace any text matching a given regular expression with some other value. I definitely recommend you read `help(re.sub)`:

```
sub(pattern, repl, string, count=0, flags=0)
    Return the string obtained by replacing the leftmost
    non-overlapping occurrences of the pattern in string by the
    replacement repl.
```

I don't want to give away the ending, but you will need to use a pattern similar to the preceding to replace each `<placeholder>` with each `value`.

Note that it's not a requirement that you use the `re.sub()` function to solve this. I challenge you, in fact, to try writing a solution that does not use the `re` module at all. Now go write the program, and use the tests to guide you!

17.2 Solution

Are you getting more comfortable with regular expressions? I know they are complicated, but really understanding them will help you more than you might expect.

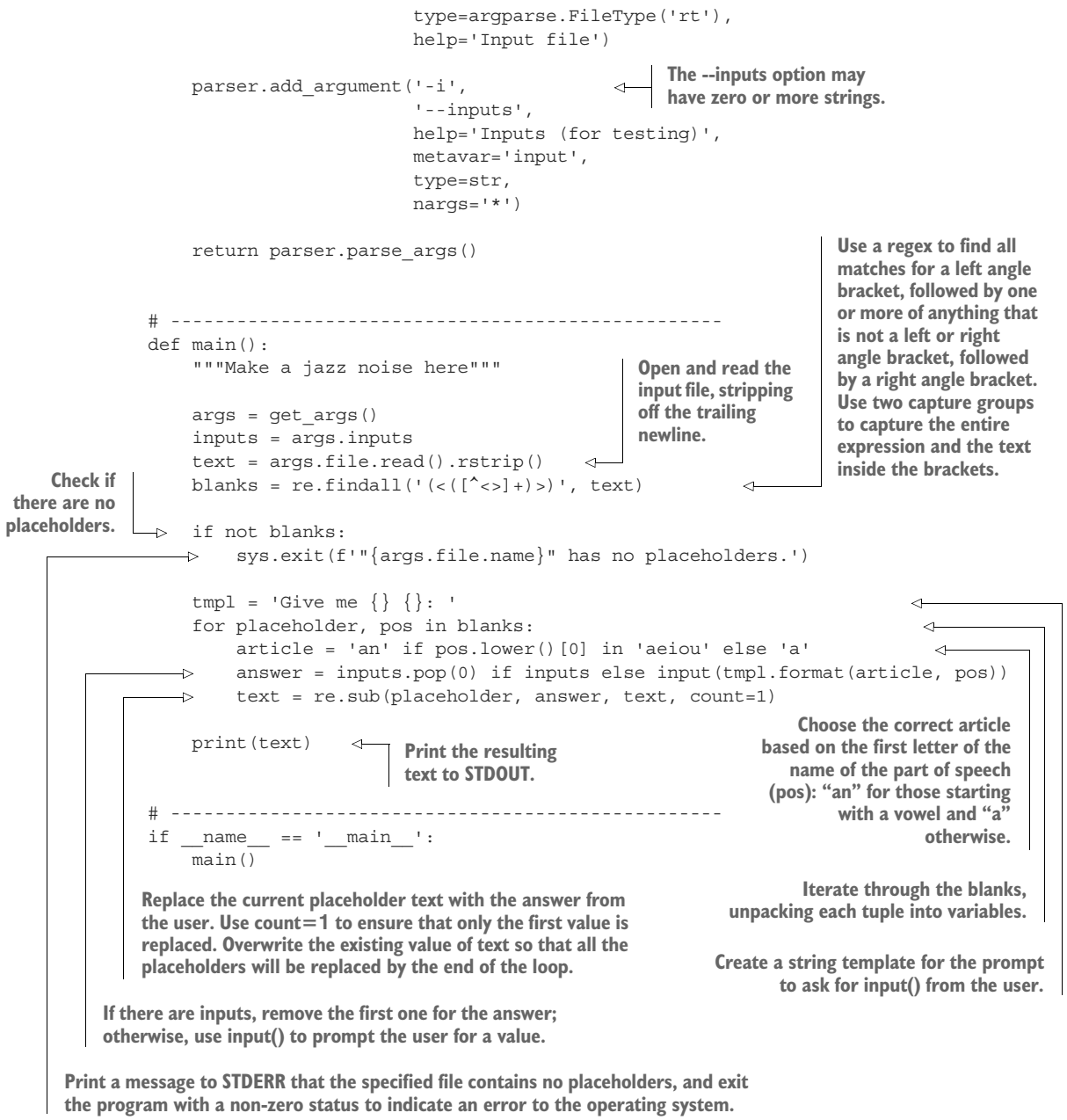
```
#!/usr/bin/env python3
"""Mad Libs"""

import argparse
import re
import sys


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

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

    parser.add_argument('file',                    <-- The file argument should be a readable text file.
                        metavar='FILE',
```



17.3 Discussion

We start off by defining our arguments well. The input `file` should be declared using `type=argparse.FileType('rt')` so that `argparse` will verify that the argument is a readable text file. The `--inputs` are optional, so we can use `nargs='*'` to indicate

zero or more strings. If no inputs are provided, the default value will be `None`, so be sure you don't assume it's a `list` and try doing list operations on a `None`.

17.3.1 Substituting with regular expressions

There is a subtle bug waiting for you in using `re.sub()`. Suppose we have replaced the first `<adjective>` with "blue" so that we have this:

```
>>> text = 'The quick blue <noun> jumps <preposition> the lazy <noun>.'
```

Now we want to replace `<noun>` with "dog," so we try this:

```
>>> text = re.sub('<noun>', 'dog', text)
```

Let's check on the value of `text` now:

```
>>> text
'The quick blue dog jumps <preposition> the lazy dog.'
```

Since there were two instances of the string `<noun>`, both got replaced with "dog," as shown in figure 17.12.

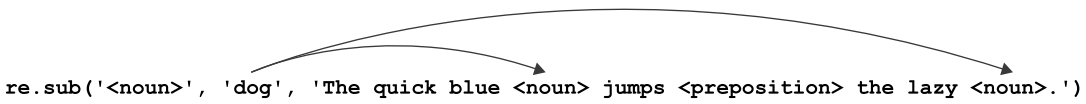


Figure 17.12 The `re.sub()` function will replace all matches.

We must use `count=1` to ensure that only the first occurrence is changed (see figure 17.13):

```
>>> text = 'The quick blue <noun> jumps <preposition> the lazy <noun>.'
>>> text = re.sub('<noun>', 'dog', text, count=1)
>>> text
'The quick blue dog jumps <preposition> the lazy <noun>.'
```

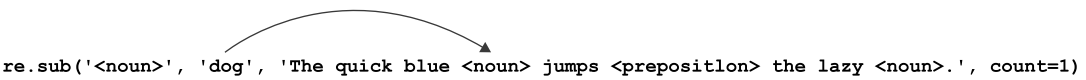


Figure 17.13 Use the `count` option to `re.sub()` to limit the number of replacements.

Now we can keep moving on to replace the other placeholders.

17.3.2 Finding the placeholders without regular expressions

I trust the explanation of the regex solution earlier in the chapter was sufficient. I find that solution fairly elegant, but it is certainly possible to solve this without using regexes. Here is how I might solve it manually.

First I need a way to search the text for <...>. I start off by writing a test that helps me imagine what I might give to my function and what I might expect in return for both good and bad values.

I decide to return `None` when the pattern is missing and to return a tuple of `(start, stop)` indices when the pattern is present:

```
def test_find_brackets():
    """Test for finding angle brackets"""
    assert find_brackets('') is None
    assert find_brackets('<>') is None
    assert find_brackets('<x>') == (0, 2)
    assert find_brackets('foo <bar> baz') == (4, 8)
```

There is no text, so it should return None.

There are angle brackets, but they lack any text inside, so this should return None.

The pattern should be found at the beginning of a string.

The pattern should be found further into the string.

Now I need to write the code that will satisfy that test. Here is what I wrote:

```
def find_brackets(text):
    """Find angle brackets"""
    start = text.index('<') if '<' in text else -1
    stop = text.index('>') if start >= 0 and '>' in text[start + 2:] else -1
    return (start, stop) if start >= 0 and stop >= 0 else None
```

Find the index of the left bracket if one is found in the text.

Find the index of the right bracket if one is found starting two positions after the left.

If both brackets were found, return a tuple of their start and stop positions; otherwise, return None.

This function works well enough to pass the given tests, but it is not quite correct because it will return a region that contains unbalanced brackets:

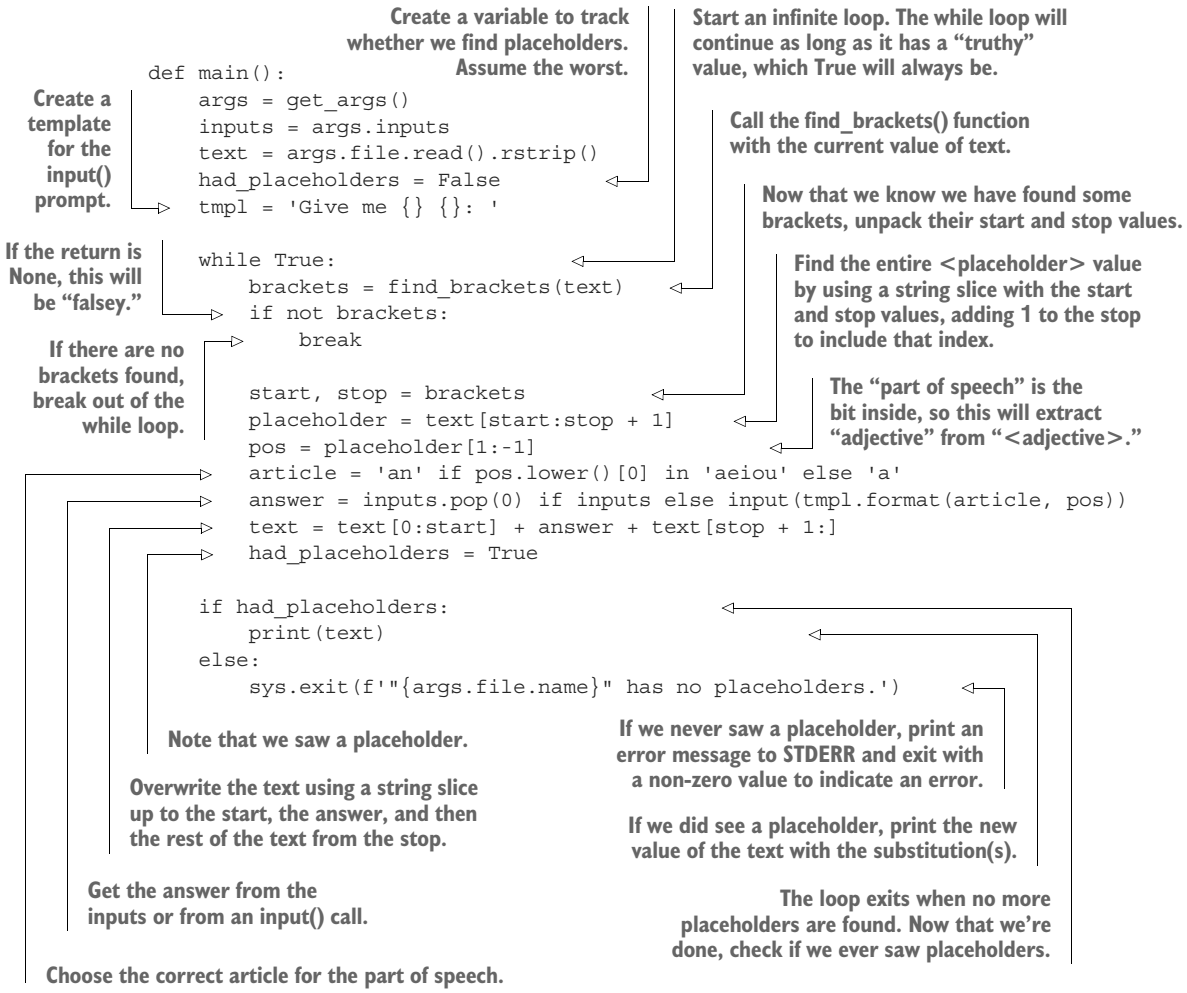
```
>>> text = 'foo <<bar> baz'
>>> find_brackets(text)
[4, 9]
>>> text[4:10]
'<<bar>'
```

That may seem unlikely, but I chose angle brackets to make you think of HTML tags like `<head>` and `<img>`. HTML is notorious for being incorrect, maybe because it was hand generated by a human who messed up a tag or because some tool that generated the HTML had a bug. The point is that most web browsers have to be fairly relaxed in parsing HTML, and it would not be unexpected to see a malformed tag like `<<head>` instead of the correct `<head>`.

The regex version, on the other hand, specifically guards against matching unbalanced brackets by using the class `[^<>]` to define text that cannot contain any angle brackets. I could write a version of `find_brackets()` that finds only balanced brackets, but, honestly, it's just not worth it. This function points out that one of the strengths of the regex engine is that it can find a partial match (the first left bracket), see that it's unable to make a complete match, and start over (at the next left bracket). Writing this myself would be tedious and, frankly, not that interesting.

Still, this function works for all the given test inputs. Note that it only returns one set of brackets at a time. I will alter the text after I find each set of brackets, which will likely change the start and stop positions of any following brackets, so it's best to handle one set at a time.

Here is how I would incorporate it into the `main()` function:




```
def main():
    args = get_args()
    inputs = args.inputs
    text = args.file.read().rstrip()
    had_placeholders = False
    tmpl = 'Give me {} {}: '

    while True:
        brackets = find_brackets(text)
        if not brackets:
            break

        start, stop = brackets
        placeholder = text[start:stop + 1]
        pos = placeholder[1:-1]
        article = 'an' if pos.lower()[0] in 'aeiou' else 'a'
        answer = inputs.pop(0) if inputs else input(tmpl.format(article, pos))
        text = text[0:start] + answer + text[stop + 1:]
        had_placeholders = True

    if had_placeholders:
        print(text)
    else:
        sys.exit(f'"{args.file.name}" has no placeholders.')
```

17.4 Going further

- Extend your code to find all the HTML tags enclosed in <...> and </...> in a web page you download from the internet.
- Write a program that will look for unbalanced open/close pairs for parentheses `()`, square brackets `[]`, and curly brackets `{}`. Create input files that have balanced and unbalanced text, and write tests that verify your program identifies both.

Summary

- Regular expressions are almost like functions where we *describe* the patterns we want to find. The regex engine will do the work of trying to find the patterns, handling mismatches and starting over to find the pattern in the text.
- Regex patterns with `*` or `+` are "greedy" in that they match as many characters as possible. Adding a `?` after them makes them "non-greedy" so that they match as *few* characters as possible.
- The `re.findall()` function will return a `list` of all the matching strings or capture groups for a given pattern.
- The `re.sub()` function will substitute a pattern in some text with new text.
- You can halt your program at any time using the `sys.exit()` function. If it's given no arguments, the default exit value will be `0` to indicate no errors. If you wish to indicate there was an error, use any non-zero value such as `1`. Or use a string value, which will be printed to `STDERR`, and a non-zero exit value will be used automatically.