1. 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!

We'll write a program called `mad.py` that will read a file given as a positional argument and find all the placeholders noted 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 "Crow's Nest.") Each value from the user will then replace the placeholder in the text, so if the user says "drive" for "verb," then `<verb>` in the text will be replace with `drive`. When all the placeholders have been replaced with inputs from the user, print out the new text.

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

```
$ 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, you will have an option for `-i` or `--inputs` 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:

- Print 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 substitute found patterns with new text.
- Explore ways to write without using regular expressions.

1.1. Writing `mad.py`

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

First modify your `mad.py` until it produces the following usage 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)
```

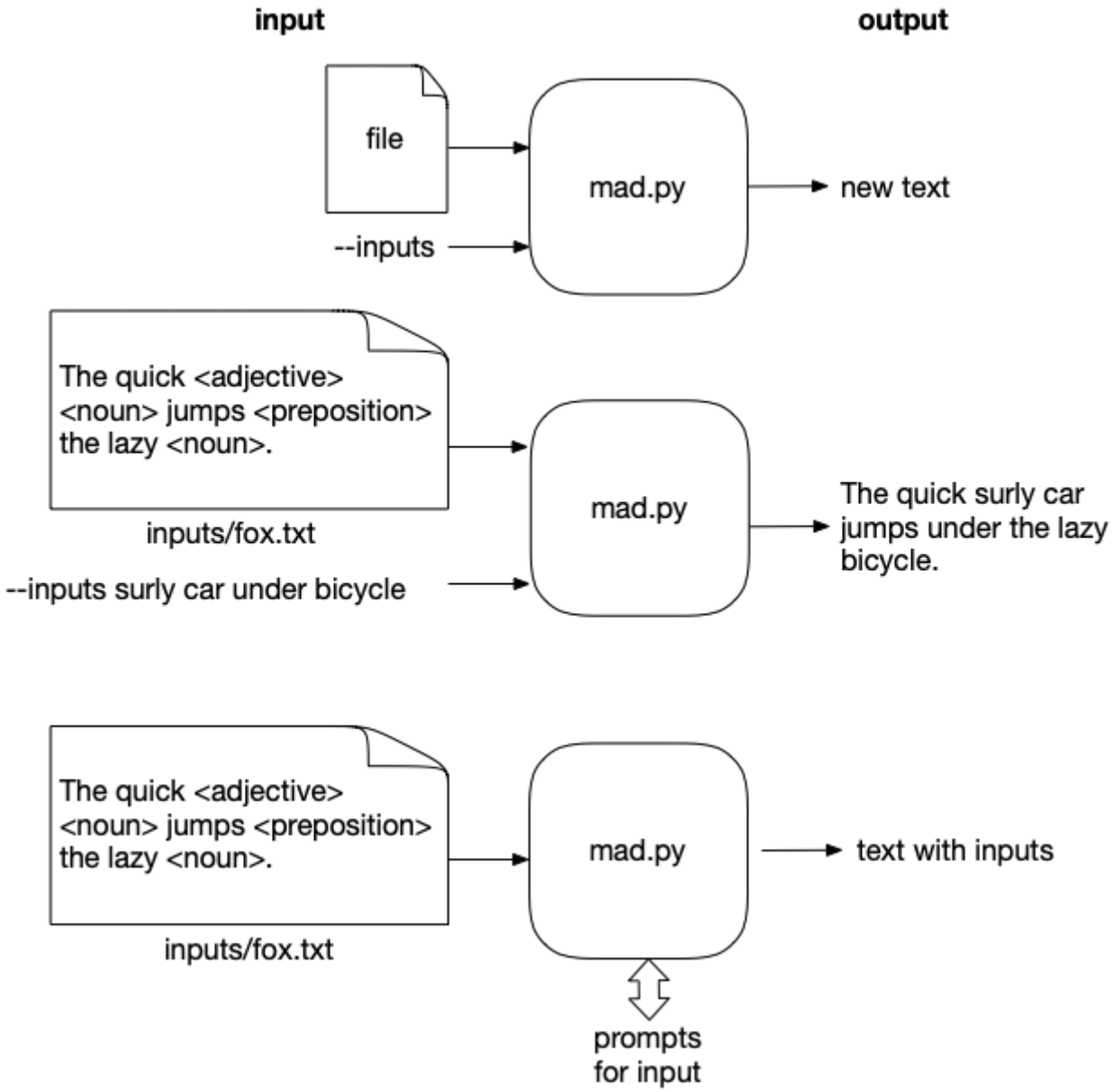
If the given `file` argument does not exist, the program should error out:

```
$ ./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, it should print a message and *exit with an error value* (something other than `0`). Note this does not need to print a usage, 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.
```

Here is a string diagram to help you visualize the program:



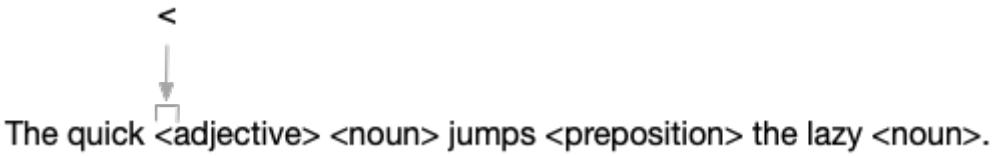
1.1.1. Using regular expressions to find the pointy bits

The first thing we need to do is `read()` the input file:

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

We need to find all the `<···>` bits, so let's use a regular expression. We can find a literal `<` character like so:

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



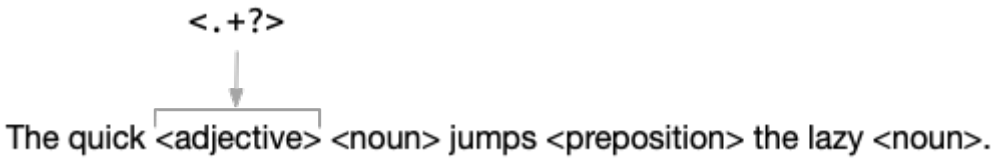
Now let's find that bracket's mate. The `.` means "anything," and we can add a `+` after it to mean "one or more". I'll capture the match so it's easier to see:

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

Hmm, that matched all the way to the end of the string instead of stopping at the first available `>`. It's common when we use `*` or `+` for zero/one or more that the regex engine is "greedy" on the *or more* part. The pattern matches beyond where we want them to, but they are technically matching exactly what we describe. Remember that `.` means *anything*, and a right angle bracket is anything. It matches as many characters as possible until it finds the last right angle to stop which is why this pattern is called "greedy."

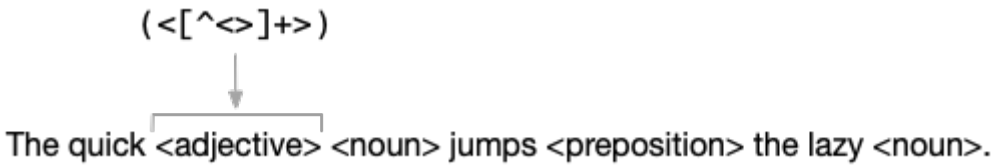
We can make the regex "non-greedy" by changing `+` to `+?` so that it matches the shortest possible string:

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



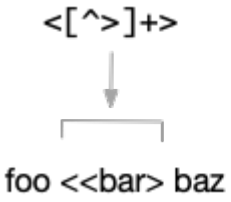
Rather than using `.` for "anything," it would be more accurate to say that we want to match one or more of anything *that is not either of the angle brackets*. The character class `[<>]` would match either bracket. We can negate (or complement) the class by putting a caret (`^`) as the first character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

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

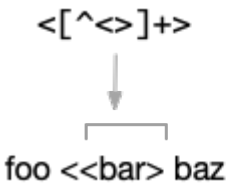


Why do we have both brackets inside the negated class? Wouldn't the right bracket be enough? Well, I'm guarding against *unbalanced* brackets. With only the right bracket, it would match this text:

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



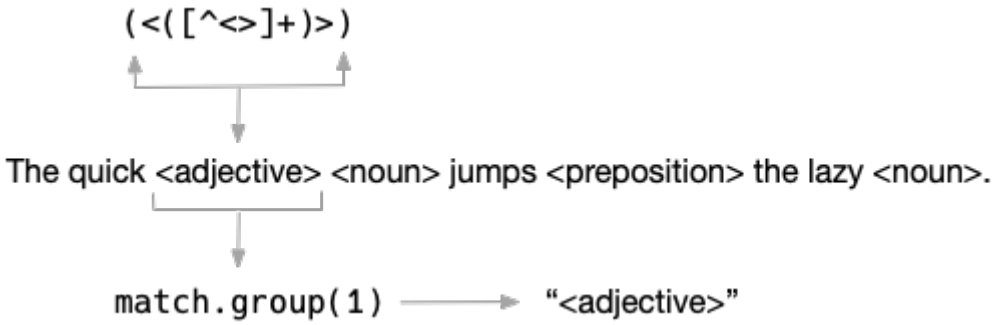
But with *both* brackets in the negated class, it finds the correct, balanced pair:



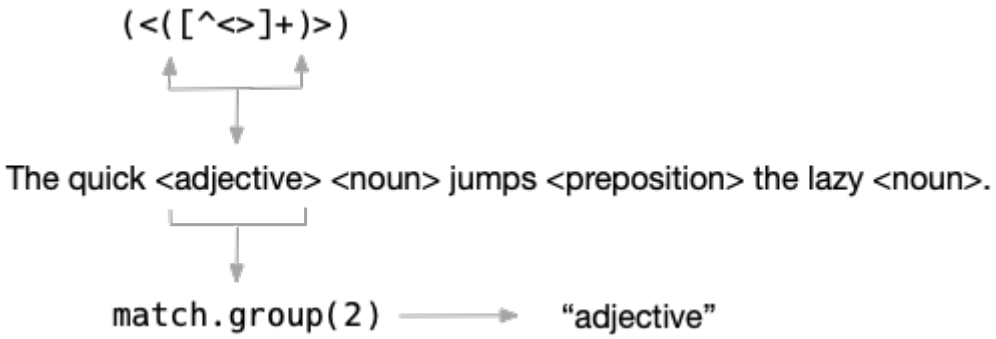

```
>>> re.search('<[^<>]+>', 'foo <<bar> baz')
<re.Match object; span=(5, 10), match='<bar>'>
```

We'll add two sets of parens `()`, one to capture the *entire* placeholder pattern:

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



And another for the string *inside* the `<>`:



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:

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

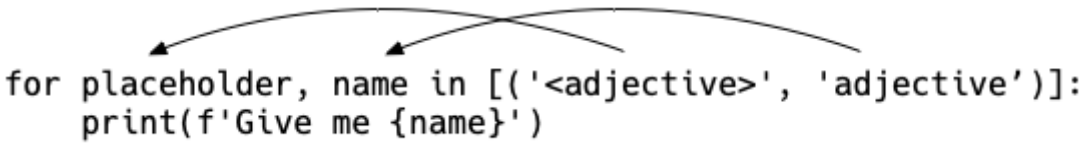


Figure 17. 1. Since the list contains 2-tuples, we can unpack them into two variables in the `for` loop.

You should insert the correct article ("a" or "an", see the "Crow’s Nest" exercise) to use as the prompt for `input()`.

1.1.2. Halting and printing errors

If you find there are no placeholders in the text, you need to print an error message. It's common to print *error* message 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 the "Abuse" chapter. To do that, be sure you import that module:

```
1 import sys
```

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

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

If there really are no placeholders, then we should exit the program *with an error value* to indicate to the operating system that our 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`:

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

One of the tests checks if your program can detect missing placeholders and if your program exits correctly.[1]

1.1.3. Getting the values

For each one of those parts of speech, you need a value that will come either from the `--inputs` argument or directly from the user. If we have nothing for `--inputs`, then you can use the `input()` function to get some answer from the user. The function takes a `str` value to use as a prompt:

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

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

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

If, however, you have values for the inputs, use those and do not bother with the `input()` function. I'm only making you handle this option for testing purposes, so you can assume you will always have the correct number of inputs for the number of placeholders in the text provided in the same order as the placeholders they should replace.

So assume this:

```
>>> 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.

1.1.4. 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()` function that will *substitute* text matching a given regular expression for some given text. I would 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 one above to replace each `<placeholder>` with each `value`.

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

1.2. Solution

```
#!/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',                          ①
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        help='Input file')

    parser.add_argument('-i',                            ②
                        '--inputs',
                        help='Inputs (for testing)',
                        metavar='input',
                        type=str,
                        nargs='*')

    return parser.parse_args()


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

    args = get_args()
    inputs = args.inputs
    text = args.file.read().rstrip()                     ③
    blanks = re.findall('(<([^<>]+)>)', text)            ④

    if not blanks:                                       ⑤
        print(f'"{args.file.name}" has no placeholders.', file=sys.stderr) ⑥
        sys.exit(1)                                      ⑦

    tmpl = 'Give me {} {}: '                                 ⑧
    for placeholder, pos in blanks:                          ⑨
        article = 'an' if pos.lower()[0] in 'aeiou' else 'a' ⑩
        answer = inputs.pop(0) if inputs else input(tmpl.format(article, pos)) ⑪
```

```
49         text = re.sub(placeholder, answer, text, count=1)    ⑫
50
51     print(text) ⑬
52
53
54 # --------------------------------------------------
55 if __name__ == '__main__':
56     main()
```

① The `file` argument should be a readable file.

② The `--inputs` option may have zero or more strings.

③ Read the input file, stripping off the trailing newline.

④ Use a regex to find all the matches for a left angle bracket followed by one or more of anything that is not a left or right angle bracket followed by a right angle bracket. Use two capture groups to capture the entire expression and the text inside the brackets.

⑤ Chec it there are no placeholders.

⑥ Print a message to `STDERR` that the given file name contains no placeholders.

⑦ Exit the program with a non-zero status to indicate an error to the operating system.

⑧ Create a string template for the prompt to ask for `input()` from the user.

⑨ Iterate through the `blanks`, unpacking each `tuple` into variables.

⑩ Choose the correct article based on the first letter of the name of the part of speech (`pos`), "an" for those starting with a vowel and "a" otherwise.

⑪ If there are inputs, remove the first one for the `answer`, otherwise use the `input()` to prompt the user for a value.

⑫ Replace the current `placeholder` text with the `answer` from the user. Use `count=1` to ensure that only the first value is replaced. Overwrite the existing value of `text` so that all the placeholders will be replaced by the end of the loop.

⑬ Print the resulting text to `STDOUT`.

1.3. Discussion

1.3.1. Defining the arguments

If you define the `file` with `type=argparse.FileType('r')`, then `argparse` will verify that the value is a file, creating an error and usage if it is not, and then will `open()` it for you. Quite the time saver. I also define `--inputs` with `nargs='*'` so that I can get any number of strings as a `list`. If nothing is 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`.

1.3.2. 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>.'
```

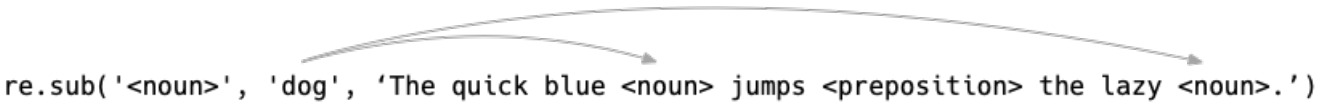
Now we want to replace `<noun>` with "dog," and so we try this:

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

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

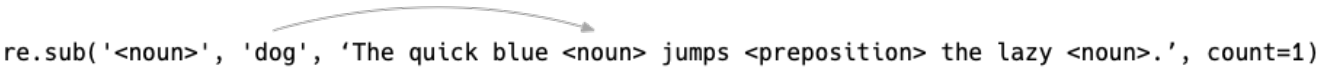
```
>>> text
'The quick blue dog jumps <preposition> the lazy dog.'
```

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



We must use `count=1` to ensure that only the first occurence is changed:

```
>>> 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>.'
```



And now we can keep moving to replace the other placeholders.

1.3.3. Finding the placeholders without regular expressions

I trust the explanation of the regex solution in the introduction 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 decided 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('') == None                ①
    assert find_brackets('<>') == None              ②
    assert find_brackets('<x>') == (0, 2)           ③
    assert find_brackets('foo <bar> baz') == (4, 8) ④
```

① There is no text, so it should return `None`.

② There are angle brackets but they lack any text inside, so this should return `None`.

③ The pattern should be found at the beginning of a string.

④ The pattern should be found further into the string.

Now 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                          ①
    stop = text.index('>') if start >= 0 and '>' in text[start + 2:] else -1 ②
    return (start, stop) if start >= 0 and stop >= 0 else None               ③
```

① Find the index of the left bracket if one is found in the text.

② Find the index of the right bracket if one is found starting two positions after the left.

③ 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 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 internal 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. This is because 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                ①
    tmpl = 'Give me {} {}: '                ②

    while True:                             ③
        brackets = find_brackets(text)      ④
        if not brackets:                    ⑤
            break                           ⑥

        start, stop = brackets              ⑦
        placeholder = text[start:stop + 1] ⑧
        pos = placeholder[1:-1]             ⑨
        article = 'an' if pos.lower()[0] in 'aeiou' else 'a' ⑩
        answer = inputs.pop(0) if inputs else input(tmpl.format(article, pos)) ⑪
        text = text[0:start] + answer + text[stop + 1:]      ⑫
        had_placeholders = True             ⑬

    if had_placeholders:                    ⑭
        print(text)                         ⑮
    else:
        print(f'"{args.file.name}" has no placeholders.', file=sys.stderr) ⑯
        sys.exit(1)                         ⑰
```

① Create a variable to track whether we find placeholders. Assume the worst.

② Create a template for the `input()` prompt.

③ Start an infinite loop. The `while` will continue as long as it has a "truthy" value as `True` will always be.

④ Call the `find_brackets()` function with the current value of `text`.

⑤ If the return is `None`, then this will be "falsey."

⑥ If there are no brackets found, use `break` to exit the infinite `while` loop.

⑦ Now that we know `brackets` is not `None`, unpack the `start` and `stop` values.

⑧ Find the entire `<placeholder>` value by using a string slice with the `start` and `stop` values, adding 1 to the `stop` to include that index.

⑨ The "part of speech" is the bit inside, so this will extract `adjective` from `<adjective>`.

⑩ Choose the correct article for the part of speech.

⑪ Get the `answer` from the `inputs` or from an `input()` call.

⑫ Overwrite the `text` using a string slice up to the `start`, the `answer`, and then the rest of the `text` from the `stop`.

⑬ Note that we saw a placeholder.

⑭ We exit the loop when we no longer find placeholders. Check if we ever saw one.

⑮ If we did see a placeholder, print the new value of the `text`.

⑯ If we never saw a placeholder, print an error message to `STDERR`.

⑰ Exit with a non-zero value to indicate an error.

1.4. Review

- 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 "not 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 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`.

1.5. 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.

[1] In Perl, there is a function called `die` which prints a message to `STDERR` and then halts the program with a non-zero value. If you like that idea, I released a module called `dire` that has a `die` function which you can use!