Exercise 49: Making Sentences

What we should be able to get from our little game lexicon scanner is a list that looks like this:

```
>>> from ex48 import lexicon
>>> print lexicon.scan("go north")
[('verb', 'go'), ('direction', 'north')]
>>> print lexicon.scan("kill the princess")
[('verb', 'kill'), ('stop', 'the'), ('noun',
'princess')]
>>> print lexicon.scan("eat the bear")
[('verb', 'eat'), ('stop', 'the'), ('noun', 'bear')]
>>> print lexicon.scan("open the door and smack the
bear in the nose")
[('error', 'open'), ('stop', 'the'), ('noun', 'door'),
('error', 'and'),
('error', 'smack'), ('stop', 'the'), ('noun', 'bear'),
('stop', 'in'),
('stop', 'the'), ('error', 'nose')]
>>>
```

Now let us turn this into something the game can work with, which would be some kind of Sentence class.

If you remember grade school, a sentence can be a simple structure like:

> Subject Verb Object

Obviously it gets more complex than that, and you probably did many days of annoying sentence graphs for English class. What we want is to turn the above lists of tuples into a nice Sentence object that has subject, verb, and object.

Match and Peek

To do this we need four tools:

1. A way to loop through the list of tuples. That's easy.
2. A way to "match" different types of tuples that we expect in our Subject Verb Object setup.
3. A way to "peek" at a potential tuple so we can make some decisions.
4. A way to "skip" things we do not care about, like stop words.

We will be putting these functions in a file named `ex48/parser.py` in order to test it. We use the `peek` function to say look at the next element in our tuple list, and then match to take one off and work with it. Let's take a look at a first `peek` function:

```
def peek(word_list):
    if word_list:
        word = word_list[0]
        return word[0]
    else:
        return None
```

Very easy. Now for the `match` function:

```
def match(word_list, expecting):
    if word_list:
        word = word_list.pop(0)

        if word[0] == expecting:
            return word
        else:
            return None
    else:
        return None
```

Again, very easy, and finally our `skip` function:

```
def skip(word_list, word_type):
    while peek(word_list) == word_type:
        match(word_list, word_type)
```

By now you should be able to figure out what these do. Make sure you understand them.

The Sentence Grammar

With our tools we can now begin to build `Sentence` objects from our list of tuples. What we do is a process of:

1. Identify the next word with `peek`.
2. If that word fits in our grammar, we call a function to handle that part of the grammar, say `parse_subject`.
3. If it doesn't, we `raise` an error, which you will learn about in this lesson.
4. When we're all done, we should have a `Sentence` object to work with in our game.

The best way to demonstrate this is to give you the code to read, but here's where this exercise is different from the previous one: You will write the test for the parser code I give you. Rather than giving you the test so you can write the code, I will give you the code, and you have to write the test.

Here's the code that I wrote for parsing simple sentences by using the `ex48.lexicon` module:

```python
class ParserError(Exception):
    pass


class Sentence(object):

    def __init__(self, subject, verb, object):
        # remember we take ('noun','princess') tuples and convert them
        self.subject = subject[1]
        self.verb = verb[1]
        self.object = object[1]


def peek(word_list):
    if word_list:
        word = word_list[0]
        return word[0]
    else:
        return None


def match(word_list, expecting):
    if word_list:
        word = word_list.pop(0)

        if word[0] == expecting:
            return word
        else:
            return None
    else:
        return None


def skip(word_list, word_type):
    while peek(word_list) == word_type:
        match(word_list, word_type)


def parse_verb(word_list):
    skip(word_list, 'stop')

    if peek(word_list) == 'verb':
        return match(word_list, 'verb')
    else:
        raise ParserError("Expected a verb next.")


def parse_object(word_list):
    skip(word_list, 'stop')
    next = peek(word_list)

    if next == 'noun':
        return match(word_list, 'noun')
    if next == 'direction':
        return match(word_list, 'direction')
    else:
        raise ParserError("Expected a noun or direction
next.")


def parse_subject(word_list, subj):
    verb = parse_verb(word_list)
    obj = parse_object(word_list)

    return Sentence(subj, verb, obj)


def parse_sentence(word_list):
    skip(word_list, 'stop')

    start = peek(word_list)

    if start == 'noun':
        subj = match(word_list, 'noun')
        return parse_subject(word_list, subj)
    elif start == 'verb':
        # assume the subject is the player then
        return parse_subject(word_list, ('noun',
'player'))
        else:
            raise ParserError("Must start with subject,
object, or verb not: %s" % start)
```

A Word On Exceptions

You briefly learned about exceptions but not how to raise them. This code demonstrates how to do that with the `ParserError` at the top. Notice that it uses classes to give it the type of `Exception`. Also notice the use of the `raise` keyword to raise the exception.

In your tests, you will want to work with these exceptions, which I'll show you how to do.

What You Should Test

For Exercise 49, write a complete test that confirms everything in this code is working. Put the test in `tests/parser_tests.py` similar to the test file from the last exercise. That includes making exceptions happen by giving it bad sentences.

Check for an exception by using the function `assert_raises` from the nose documentation. Learn how to use this so you can write a test that is *expected* to fail, which is very important in testing. Learn about this function (and others) by reading the nose documentation.

When you are done, you should know how this bit of code works and how to write a test for other people's code even if they do not want you to. Trust me, it's a very handy skill to have.

Study Drills

1. Change the `parse_` methods and try to put them into a class rather than

be just methods. Which design do you like better?
2. Make the parser more error-resistant so that you can avoid annoying your users if they type words your lexicon doesn't understand.
3. Improve the grammar by handling more things like numbers.
4. Think about how you might use this Sentence class in your game to do more fun things with a user's input.

Common Student Questions

I can't seem to make `assert_raises` work right.

Make sure you are writing `assert_raises(exception, callable, parameters)` and *not* writing `assert_raises(exception, callable(parameters))`. Notice how the second form is calling the function then passing the result to `assert_raises`, which is *wrong*. You have to pass the function to call *and* its arguments to `assert_raises` instead.

Buying is easy. Just fill out the form below and we'll get started.

Full Name

Email Address

[✓] VISA MasterCard Pay With Credit Card (by Stripe™)

[ ] PayPal Use your PayPal™ account.

Buy Learn Python The Hard Way, 3rd Edition