



Exercise 49: Making Sentences

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

```
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from ex48 import lexicon
>>> lexicon.scan("go north")
[('verb', 'go'), ('direction', 'north')]
>>> lexicon.scan("kill the princess")
[('verb', 'kill'), ('stop', 'the'), ('noun', 'princess')]
>>> lexicon.scan("eat the bear")
[('verb', 'eat'), ('stop', 'the'), ('noun', 'bear')]
>>> lexicon.scan("open the door and smack the bear in the nose")
[('error', 'open'), ('stop', 'the'), ('error', '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 scanned words. 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.
5. A Sentence object to put the results in.

We will be putting these functions in a module named `ex48.parser` 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.

The Sentence Grammar

Before you can write the code you need to understand how a basic English

sentence grammar works. In our parser we want to produce a `Sentence` object that has three attributes:

`Sentence.subject`

This is the subject of any sentence, but could default to "player" most of the time since a sentence of, "run north" is implying "player run north". This will be a noun.

`Sentence.verb`

This is the action of the sentence. In "run north" it would be "run". This will be a verb.

`Sentence.object`

This is another noun that refers to what the verb is done on. In our game we separate out directions which would also be objects. In "run north" the word "north" would be the object. In "hit bear" the word "bear" would be the object.

Our parser then has to use the functions we described and given a scanned sentence, convert it into an `List` of `Sentence` objects to match the input.

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.

The Parser Code

If you want an extra challenge, stop right now and try to write this based on just my description. If you get stuck you can come back and see how I did it, but trying to implement the parser yourself is good practice. I will now walk through the code so you can enter it into your `ex48/parser.py`. We start the parser with the exception we need for a parsing error:

```
1 class ParserError(Exception):
2     pass
```

This is how you make your own `ParserError` exception class you can throw. Next we need the `Sentence` object we'll create:

```
1 class Sentence(object):
2
3     def __init__(self, subject, verb, obj):
4         # remember we take ('noun', 'princess') tuples and convert them
5         self.subject = subject[1]
6         self.verb = verb[1]
7         self.object = obj[1]
```

There's nothing special about this code so far. You're just making simple classes.

In our description of the problem we need a function that can peek at a list of words and return what type of word it is:

```

1 | def peek(word_list):
2 |     if word_list:
3 |         word = word_list[0]
4 |         return word[0]
5 |     else:
6 |         return None

```

We need this function because we'll have to make decisions about what kind of sentence we're dealing with based on what the next word is, and then we can call another function to consume that word and carry on.

To consume a word we use a the `match` function, which confirms that the expected word is the right type, takes it off the list, and returns the word.

```

1 | def match(word_list, expecting):
2 |     if word_list:
3 |         word = word_list.pop(0)
4 |
5 |         if word[0] == expecting:
6 |             return word
7 |         else:
8 |             return None
9 |     else:
10 |         return None

```

Again, fairly simple but make sure you understand this code but also *why* I'm doing it this way. I need to peek at words in the list to decide what kind of sentence I'm dealing with, and then I need to match those words to create my `Sentence`.

The last thing I need is a way to skip words that aren't useful to the `Sentence`. These are the words labeled "stop words" (type 'stop') that are words like "the", "and", and "a".

```

1 | def skip(word_list, word_type):
2 |     while peek(word_list) == word_type:
3 |         match(word_list, word_type)

```

Remember that `skip` doesn't skip one word, it skips as many words of that type as it finds. This makes it so if someone types, "scream at the bear" you get "scream" and "bear."

That's our basic set of parsing functions, and with that we can actually parse just about any text we want. Our parser is very simple though, so the remaining functions are short.

First we can handle parsing a verb:

```

1 | def parse_verb(word_list):
2 |     skip(word_list, 'stop')
3 |
4 |     if peek(word_list) == 'verb':
5 |         return match(word_list, 'verb')
6 |     else:
7 |         raise ParserError("Expected a verb next.")

```

We skip any stop words, then peek ahead to make sure the next word is a "verb" type. If it's not then raise the `ParserError` to say why. If it is a "verb" then match it, which takes it off the list. A similar function handles sentence objects:

```

1 def parse_object(word_list):
2     skip(word_list, 'stop')
3     next_word = peek(word_list)
4
5     if next_word == 'noun':
6         return match(word_list, 'noun')
7     elif next_word == 'direction':
8         return match(word_list, 'direction')
9     else:
10        raise ParserError("Expected a noun or direction next.")

```

Again, skip the stop words, peek ahead, and decide if the sentence is correct based on what's there. In the `parse_object` function though we need to handle both "noun" and "direction" words as possible objects. Subjects are then similar again, but since we want to handle the implied "player" noun, we have to use `peek`:

```

1 def parse_subject(word_list):
2     skip(word_list, 'stop')
3     next_word = peek(word_list)
4
5     if next_word == 'noun':
6         return match(word_list, 'noun')
7     elif next_word == 'verb':
8         return ('noun', 'player')
9     else:
10        raise ParserError("Expected a verb next.")

```

With that all out of the way and ready, our final `parse_sentence` function is very simple:

```

1 def parse_sentence(word_list):
2     subj = parse_subject(word_list)
3     verb = parse_verb(word_list)
4     obj = parse_object(word_list)
5
6     return Sentence(subj, verb, obj)

```

Playing With The Parser

To see how this works, you can play with it like this:

```

Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from ex48.parser import *
>>> x = parse_sentence([('verb', 'run'), ('direction', 'north')])
>>> x.subject
'player'
>>> x.verb
'run'
>>> x.object
'north'
>>> x = parse_sentence([('noun', 'bear'), ('verb', 'eat'), ('stop', 'the'), ('noun',
'honey')])
>>> x.subject
'bear'
>>> x.verb
'eat'
>>> x.object
'honey'

```

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 the parser 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

Q: 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`.

Video

Purchase The Videos For \$29.59

For just \$29.59 you can get access to all the videos for Learn Python The Hard Way, **plus** a PDF of the book and no more popups all in this one location. For \$29.59 you get:

- All 52 videos, 1 per exercise, almost 2G of video.
- A PDF of the book.
- Email help from the author.
- See a list of everything you get before you buy.

When you buy the videos they will immediately show up **right here** without any hassles.


Buying Is Easy

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

Full Name

Email Address

☐    Pay With Credit Card (by Stripe™)

☐  Use your PayPal™ account.

Already Paid? Reactivate Your Purchase Right Now!

Buy Learn Python The Hard Way, 3rd Edition

Zed Shaw PDF + Videos + Updates \$29.95	Amazon Paper + DVD \$22.74	Amazon Kindle \$17.27	B&N Paper + DVD \$22.96	B&N Nook (No Video) \$17.27
	InformIT eBook + Paper \$43.19	Interested In Ruby? Ruby is also a great language. Learn Ruby The Hard Way		