Exercise 48: Advanced User Input

Your game probably was coming along great, but I bet how you handled what the user typed was becoming tedious. Each room needed its own very exact set of phrases that only worked if your player typed them perfectly. What you'd rather have is a device that lets users type phrases in various ways. For example, we'd like to have all of these phrases work the same:

- open door
- open the door
- go THROUGH the door
- punch bear
- Punch The Bear in the FACE

It should be alright for a user to write something a lot like English for your game, and have your game figure out what it means. To do this, we're going to write a module that does just that. This module will have a few classes that work together to handle user input and convert it into something your game can work with reliably.

In a simple version of English the following elements:

- Words separated by spaces.
- Sentences composed of the words.
- Grammar that structures the sentences into meaning.

That means the best place to start is figuring out how to get words from

the user and what kinds of words those are.

Our Game Lexicon

In our game we have to create a lexicon of words:

- Direction words: north, south, east, west, down, up, left, right, back.
- Verbs: go, stop, kill, eat.
- Stop words: the, in, of, from, at, it
- Nouns: door, bear, princess, cabinet.
- Numbers: any string of 0 through 9 characters.

When we get to nouns, we have a slight problem since each room could have a different set of nouns, but let's just pick this small set to work with for now and improve it later.

Breaking Up a Sentence

Once we have our lexicon of words we need a way to break up sentences so that we can figure out what they are. In our case, we've defined a sentence as "words separated by spaces," so we really just need to do this:

```
stuff = raw_input('> ')
words = stuff.split()
```

That's really all we'll worry about for now, but this will work really well for quite a while.

Lexicon Tuples

Once we know how to break up a sentence into words, we just have to go through the list of words and figure out what "type" they are. To do that we're going to use a handy little Python structure called a "tuple." A tuple is nothing more than a list that you can't modify. It's created by putting data inside two `()` with a comma, like a list:

```
first_word = ('direction', 'north')
second_word = ('verb', 'go')
sentence = [first_word, second_word]
```

This creates a pair (TYPE, WORD) that lets you look at the word and do things with it.

This is just an example, but that's basically the end result. You want to take raw input from the user, carve it into words with `split`, then analyze those words to identify their type, and finally make a sentence out of them.

Scanning Input

Now you are ready to write your scanner. This scanner will take a string of raw input from a user and return a sentence that's composed of a list of tuples with the (TOKEN, WORD) pairings. If a word isn't part of the lexicon then it should still return the WORD but set the TOKEN to an error token. These error tokens will tell users they messed up.

Here's where it gets fun. I'm not going to tell you how to do this. Instead I'm going to write a `unit test` and you are going to write the scanner so that the unit test works.

Exceptions and Numbers

There is one tiny thing I will help you with first, and that's converting numbers. In order to do this though, we're going to cheat and use exceptions. An exception is an error that you get from some function you may have run. What happens is your function "raises" an exception when it encounters an error, then you have to handle that exception. For example, if you type this into Python:

```
~/projects/simplegame $ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> int("hell")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10:
'hell'
>>
```

That `ValueError` is an exception that the `int()` function threw because what you handed `int()` is not a number. The `int()` function could have returned a value to tell you it had an error, but since it only returns integers, it'd have a hard time doing that. It can't return -1 since that's a number. Instead of trying to figure out what to return when there's an error, the `int()` function raises the `ValueError` exception and you deal with it.

You deal with an exception by using the `try` and `except` keywords:

```
def convert_number(s):
    try:
        return int(s)
    except ValueError:
        return None
```

You put the code you want to "try" inside the `try` block, and then you put the code to run for the error inside the `except`. In this case, we want to "try" to call `int()` on something that might be a number. If that has an error, then we "catch" it and return `None`.

In your scanner that you write, you should use this function to test if something is a number. You should also do it as the last thing you check for before declaring that word an error word.

What You Should Test

Here are the files from `tests/lexicon_tests.py` that you should use:

```
    from nose.tools import *
    from ex47 import lexicon
1
2
3   def test_directions():
4       assert_equal(lexicon.scan("north"), [('direction',
5   'north')])
6       result = lexicon.scan("north south east")
```

```
    assert_equal(result, [('direction', 'north'),
                          ('direction', 'south'),
                          ('direction', 'east')])

def test_verbs():
    assert_equal(lexicon.scan("go"), [('verb', 'go')])
    result = lexicon.scan("go kill eat")
    assert_equal(result, [('verb', 'go'),
                          ('verb', 'kill'),
                          ('verb', 'eat')])


def test_stops():
    assert_equal(lexicon.scan("the"), [('stop', 'the')])
    result = lexicon.scan("the in of")
    assert_equal(result, [('stop', 'the'),
                          ('stop', 'in'),
                          ('stop', 'of')])


def test_nouns():
    assert_equal(lexicon.scan("bear"), [('noun',
'bear')])
    result = lexicon.scan("bear princess")
    assert_equal(result, [('noun', 'bear'),
                          ('noun', 'princess')])

def test_numbers():
    assert_equal(lexicon.scan("1234"), [('number',
1234)])
    result = lexicon.scan("3 91234")
    assert_equal(result, [('number', 3),
                          ('number', 91234)])


def test_errors():
    assert_equal(lexicon.scan("ASDFADFASDF"), [('error',
'ASDFADFASDF')])
    result = lexicon.scan("bear IAS princess")
    assert_equal(result, [('noun', 'bear'),
                          ('error', 'IAS'),
                          ('noun', 'princess')])
```

Remember that you will want to make a new project with your skeleton, type in this test case (do not copy-paste!) and write your scanner so that the test runs. Focus on the details and make sure everything works right.

Design Hints

Focus on getting one test working at a time. Keep this simple and just put all the words in your lexicon in lists that are in your `lexicon.py` module. Do not modify the input list of words, but instead make your own new list with your lexicon tuples in it. Also, use the `in` keyword with these lexicon lists to check if a word is in the lexicon.

Study Drills

1. Improve the unit test to make sure you cover more of the lexicon.
2. Add to the lexicon and then update the unit test.
3. Make sure your scanner handles user input in any capitalization and case. Update the test to make sure this actually works.
4. Find another way to convert the number.
5. My solution was 37 lines long. Is yours longer? Shorter?

Common Student Questions

Why do I keep getting `ImportErrors`?

Import errors are caused by usually four things. 1. You didn't make a `__init__.py` in a directory that has modules in it. 2. you are in the wrong directory. 3. You are importing the wrong module because you spelled it wrong. 4. Your `PYTHONPATH` isn't set to `.` so you can't load modules from your current directory.

What's the difference between `try-except` and `if-else`?

The `try-expect` construct is only used for handling exceptions that modules can throw. It should *never* be used as an alternative to `if-else`.

Is there a way to keep the game running while the user is waiting to type?

I'm assuming you want to have a monster attack users if they don't react quick enough. It is possible but it involves modules and techniques that are outside of this book's domain.

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.

Already Paid? Reactivate Your Purchase Right Now!

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.

Buy Learn Python The Hard Way, 3rd Edition