⚠ **Warning**

I will be closing off free access to my Python 2 (https://learnpythonthehardway.org/book/) and Python 3 (https://learnpythonthehardway.org/python3/") at Midnight on July 8th, 2017. If you need this book then please purchase it (http://bit.ly/buypython). If you need a free book then I recommend you read my Learn Ruby The Hard Way (https://learnrubythehardway.org/book/) book instead. You can read more about my decision at the blog (https://blog.learncodethehardway.com/2017/07/07/learn-python-3-the-hard-way-officially-released/) and you can email me at help@learncodethehardway.org if you need help with this. My sincere apologies to anyone that is inconvenienced by this decision.

Exercise 48: Advanced User Input

In past games you handled the user's input by simply expecting set strings. If the user typed "run", and exactly "run", then the game worked. If they typed in similar phrases like "run fast" it would fail. What we need is a device that lets users type phrases in various ways and then convert that into something the computer understands. 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 list of allowed words called a "lexicon":

- 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 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 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 = ('verb', 'go')
second_word = ('direction', 'north')
third_word = ('direction', 'west')
sentence = [first_word, second_word, third_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`, 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:

```
Python 2.7.11 (default, May 25 2016, 05:27:56)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.29)] on darwin
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:

```python
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.

A Test First Challenge

Test first is a programming tactic where you write an automated test that pretends the code works, *then* you write the code to make the test actually work. This method works when you can't visualize how the code is implemented, but you can imagine how you have to work with it. For example, if you know how you need to use a new class in another module, but you don't quite know how to implement that class yet, then write the test first.

You are going to take a test I give you and use it to write the code that makes it work. To do this exercise you're going to follow this procedure:

1. Create one small part of the test I give you.

2. Make sure it runs and *fails* so you know that the test is actually confirming a feature works.
3. Go to your source file `lexicon.py` and write the code that makes this test pass.
4. Repeat until you have implemented everything in the test.

When you get to 3 it's also good to combine our other method of writing code:

1. Make the "skeleton" function or class that you need.
2. Write comments inside describing how that function works.
3. Write the code that does what the comments describe.
4. Remove any comments that just repeat the code.

This method of writing code is called "psuedo code" and works well if you don't know how to implement something, but you can describe it in your own words.

Combining the "test first" with the "psuedo code" tactics we have this simple process for programming:

1. Write a bit of test that fails.
2. Write the skeleton function/module/class the test needs.
3. Fill the skeleton with comments in your own words explaining how it works.
4. Replace the comments with code until the test passes.
5. Repeat.

In this exercise you will practice this method of working by making a test I give you run against the `lexicon.py` module.

What You Should Test

Here is the test case `tests/lexicon_tests.py` that you should use, but *don't type this in yet*:

```
from nose.tools import *
from ex48 import lexicon


def test_directions():
    assert_equal(lexicon.scan("north"), [('direction', 'north')])
    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')])

```

```python

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')])
```

You will want to create a new project using the project skeleton just like you did in Exercise 47. Then you'll need to create this test case and the `lexicon.py` file it will use. Look at the top of the test case to see how it's being imported to figure out where it goes.

Next, follow the procedure I gave you and write a little bit of the test case at a time. For example, here's how I'd do it:

1. Write the import at the top. Get that to work.

2. Create an empty version of the first test case `test_directions` . Make sure that runs.
3. Write the first line of the `test_directions` test case. Make it fail.
4. Go to the `lexicon.py` file, create an empty `scan` function.
5. Run the test, make sure `scan` is at least running, even though it fails.
6. Fill in psuedo code comments for how `scan` should work to make `test_directions` pass.
7. Write the code that matches the comments until `test_directions` passes.
8. Go back to `test_directions` and write the rest of the lines.
9. Back to `scan` in `lexicon.py` and work on it to make this new test code pass.
10. Once you've done that you have your first passing test, and you move on to the next test.

As long as you keep following this procedure one little chunk at a time you can successfully turn a large problem into smaller solvable problems. It's like climbing a mountain by turning it into a bunch of little hills.

Study Drills

1. Improve the unit test to make sure you test 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-except` 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.

Gitter LCodeTHW Community Chat

Zed sometimes hangs out in the Learn Code The Hard Way community chat rooms at http://bit.ly/lcthwchat (http://bit.ly/lcthwchat) If you have a quick question or just want to hang out with other people working on the books then join in.

Buy The Python 2 Course

When you buy directly from the author, Zed A. Shaw, you'll get a professional quality PDF and hours of HD Video, all DRM-free and yours to download.

$ 29. 99

BUY THE PYTHON 2 COURSE (HTTPS://SHOP.LEARNCODETHEHARDWAY.ORG/ACCESS/BUY/2/)

PRE-ORDER THE PYTHON 3 COURSE INSTEAD (HTTPS://SHOP.LEARNCODETHEHARDWAY.ORG/ACCESS/BUY/9/)

OR, YOU CAN READ LEARN PYTHON THE HARD WAY FOR FREE (HTTPS://LEARNPYTHONTHEHARDWAY.ORG/BOOK/) RIGHT HERE, VIDEO LECTURES NOT INCLUDED.

Other Buying Options

BUY ON AMAZON (HTTP://BIT.LY/AMZNLPTHW)

BUY A HARD COPY FROM THE PUBLISHER (HTTP://BIT.LY/INFORMITLPTHW)

BUY A HARD COPY FROM BARNES & NOBLE (HTTP://BIT.LY/BNLPTHW)

Previous (ex47.html)

Next (ex49.html)

HOME (/) **ABOUT (HTTPS://LEARNCODETHEHARDWAY.ORG/ABOUT/)** **CONTACT (HTTPS://LEARNCODETHEHARDWAY.ORG/CONTACT/)**

(https://twitter.com/lzsthw)