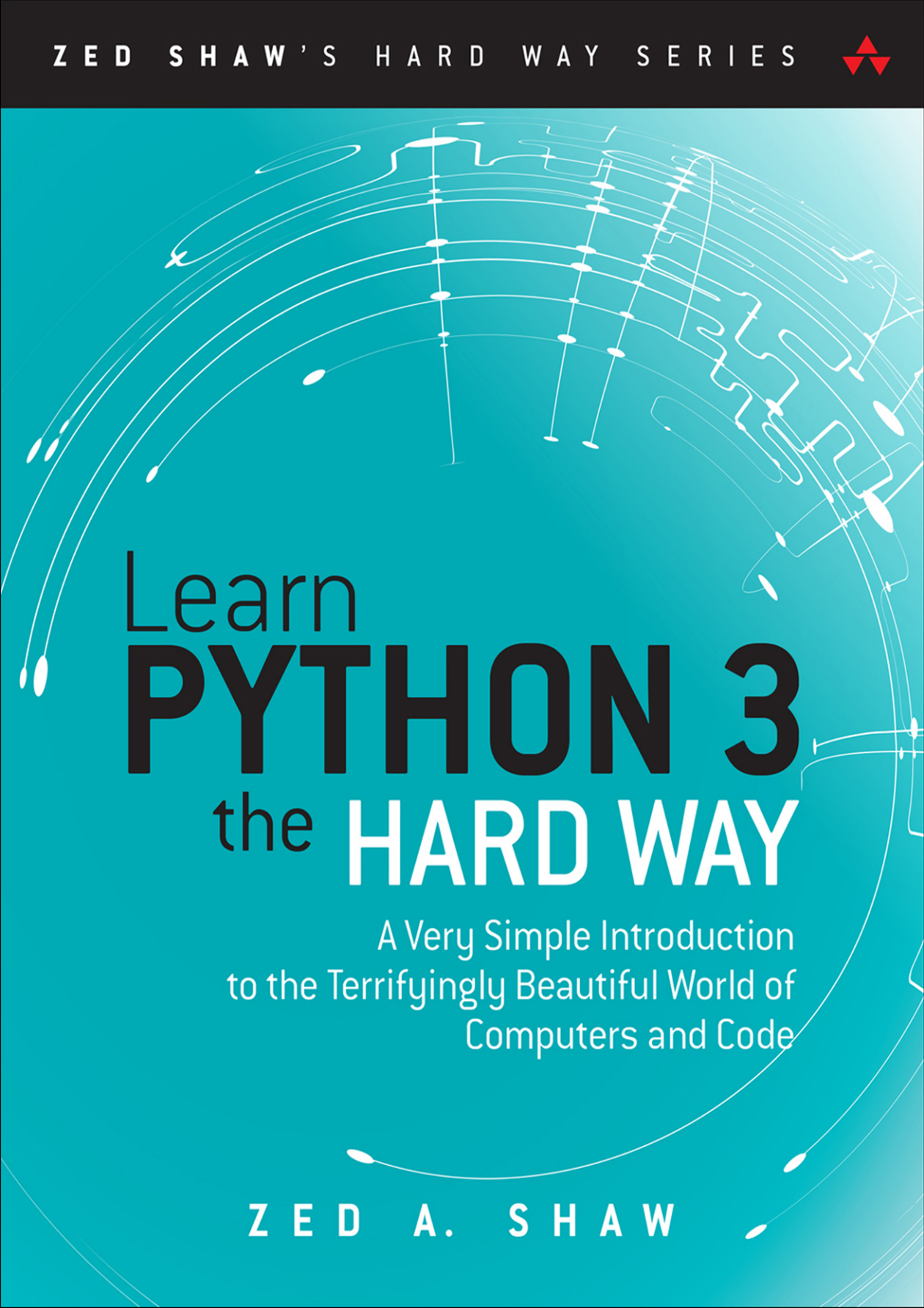
ZED SHAW'S HARD WAY SERIES



Learn PYTHON 3 the HARD WAY

A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code

ZED A. SHAW

LEARN PYTHON 3 THE HARD WAY

EXERCISE 22

What Do You Know So Far?

There won't be any code in this exercise, so there's no *What You Should See* or *Study Drills* sections. In fact, this exercise is like one giant *Study Drills*. I'm going to have you do a review of what you have learned so far.

First, go back through every exercise you have done so far and write down every word and symbol (another name for "character") that you have used. Make sure your list of symbols is complete.

Next to each word or symbol, write its name and what it does. If you can't find a name for a symbol in this book, then look for it online. If you do not know what a word or symbol does, then read about it again and try using it in some code.

You may run into a few things you can't find out or know, so just keep those on the list and be ready to look them up when you find them.

Once you have your list, spend a few days rewriting the list and double-checking that it's correct. This may get boring, but push through and really nail it down.

Once you have memorized the list and what they do, then step it up by writing out tables of symbols, their names, and what they do *from memory*. If you hit some you can't recall from memory, go back and memorize them again.

WARNING! The most important thing when doing this exercise is, "There is no failure, only trying."

What You Are Learning

It's important when you are doing a boring, mindless memorization exercise like this to know why. It helps you focus on a goal and know the purpose of all your efforts.

In this exercise you are learning the names of symbols so that you can read source code more easily. It's similar to learning the alphabet and basic words of English, except this Python alphabet has extra symbols you might not know.

Just take it slow and do not hurt your brain. It's best to take 15 minutes at a time with your list and then take a break. Giving your brain a rest will help you learn faster with less frustration.

This page intentionally left blank

EXERCISE 23

Strings, Bytes, and Character Encodings

To do this exercise you'll need to *download* a text file that I've written named languages.txt (https://learnpythonthehardway.org/python3/languages.txt). This file was created with a list of human languages to demonstrate a few interesting concepts:

1. How modern computers store human languages for display and processing and how Python 3 calls this `strings`
2. How you must "encode" and "decode" Python's strings into a type called `bytes`
3. How to handle errors in your string and byte handling
4. How to read code and find out what it means even if you've never seen it before

In addition to that you'll also get a brief glimpse of the Python 3 `if-statement` and `lists` for processing a list of things. You don't have to master this code or understand these concepts right away. You'll get plenty of practice in later exercises. For now your job is to get a taste of the future and learn the four topics in the preceding list.

WARNING! This exercise is hard! There's a lot of information in it that you need to understand, and it's information that goes deep into computers. This exercise is complex because Python's strings are complex and difficult to use. I recommend you take this exercise *painfully* slow. Write down every word you don't understand, and look it up or research it. Take a paragraph at a time if you must. You can continue with other exercises while you study this one, so don't get stuck here. Just chip away at it for as long as it takes.

Initial Research

I'm going to teach you how to research a piece of code to expose its secrets. You'll need the languages.txt file for this code to work, so make sure you download it first. The `languages.txt` file simply contains a list of human language names that are encoded in UTF-8.

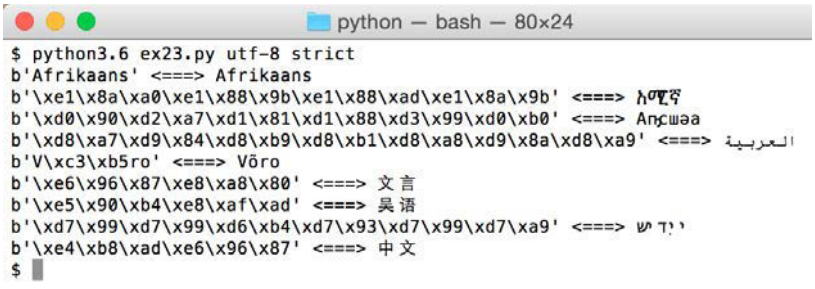
ex23.py

```
1   import sys
2   script, input_encoding, error = sys.argv
3
4
5   def main(language_file, encoding, errors):
6       line = language_file.readline()
7
```

```
 8        if line:
 9            print_line(line, encoding, errors)
10            return main(language_file, encoding, errors)
11
12
13    def print_line(line, encoding, errors):
14        next_lang = line.strip()
15        raw_bytes = next_lang.encode(encoding, errors=errors)
16        cooked_string = raw_bytes.decode(encoding, errors=errors)
17
18        print(raw_bytes, "<===>", cooked_string)
19
20
21    languages = open("languages.txt", encoding="utf-8")
22
23    main(languages, input_encoding, error)
```

Simply write down a list of each thing you've never seen before. There may be quite a few things that are new, so scan the file a few times.

Once you have that you'll want to run this Python script to play with it. Here are some example commands I used to test it:



WARNING! You'll notice I'm using images here to show you what you should see. After extensive testing it turns out that so many people have their computers configured to not display UTF-8 that I had to use images so you'll know what to expect. Even my own typesetting system (LaTeX) couldn't handle these encodings, forcing me to use images instead. If you don't see this then your Terminal is most likely not able to display UTF-8 and you should try to fix that.

These examples use the `utf-8`, `utf-16`, and `big5` encodings to demonstrate the conversion and the types of errors you can get. Each of these names are called a "codec" in Python 3, but you use the

parameter “encoding.” At the end of this exercise there’s a list of the available encodings if you want to try more. I’ll cover what all of this output means shortly. You’re only trying to get an idea of how this works so we can talk about it.

After you’ve run it a few times, go through your list of symbols and make a guess as to what they do. When you’ve written down your guesses try looking the symbols up online to see if you can confirm your hypotheses. Don’t worry if you have no idea how to search for them. Just give it a try.

Switches, Conventions, and Encodings

Before I can get into what this code means, you need to learn some basics about how data is stored in a computer. Modern computers are incredibly complex, but at their cores they are like a huge array of light switches. Computers use electricity to flip switches on or off. These switches can represent 1 for on, or 0 for off. In the old days there were all kinds of weird computers that did more than just 1 or 0, but these days it’s just 1s and 0s. One represents energy, electricity, on, power, substance. Zero represents off, done, gone, power down, the lack of energy. We call these 1s and 0s “bits.”

Now, a computer that only lets you work with 1 and 0 would be both horribly inefficient and incredibly annoying. Computers take these 1s and 0s and use them to encode larger numbers. At the small end a computer will use 8 of these 1s and 0s to encode 256 numbers (0-255). What does “encode” mean, though? It’s nothing more than an agreed-upon standard for how a sequence of bits should represent a number. It’s a convention humans picked or stumbled on that says that `00000000` would be 0, `11111111` would be 255, and `00001111` would be 15. There were even huge wars in the early history of computers on nothing more than the order of these bits because they were simply conventions we all had to agree on.

Today we call a “byte” a sequence of 8 bits (1s and 0s). In the old days everyone had their own convention for a byte, so you’ll still run into people who think that this term should be flexible and handle sequences of 9 bits, 7 bits, or 6 bits, but now we just say it’s 8 bits. That’s our convention, and that convention defines our encoding for a byte. There are futher conventions for encoding large numbers using 16, 32, 64, and even more bits if you get into really big math. There’s entire standards groups who do nothing but argue about these conventions, then implement them as encodings that eventually turn switches on and off.

Once you have bytes you can start to store and display text by deciding on another convention for how a number maps to a letter. In the early days of computing there were many conventions that mapped 8 or 7 bits (or less or more) onto lists of characters kept inside a computer. The most popular convention ended up being American Standard Code for Information Interchange, or ASCII. This standard maps a number to a letter. The number 90 is Z, which in bits is `1011010`, which gets mapped to the ASCII table inside the computer.

You can try this out in Python right now:

```
>>> 0b1011010
90
>>> ord('Z')
90
>>> chr(90)
'Z'
>>>
```

First, I write the number 90 in binary, then I get the number based on the letter `'Z'`, then I convert the number to the letter `'Z'`. Don't worry about needing to remember this though. I think I've had to do it twice the entire time I've used Python.

Once we have the ASCII convention for encoding a character using 8 bits (a byte), we can then "string" them together to make a word. If I want to write my name, "Zed A. Shaw," I just use a sequence of bytes: `[90, 101, 100, 32, 65, 46, 32, 83, 104, 97, 119]`. Most of the early text in computers was nothing more than sequences of bytes, stored in memory, that a computer used to display text to a person. Again, this is just a sequence of conventions that turned switches on and off.

The problem with ASCII is that it only encodes English and maybe a few other similar languages. Remember that a byte can hold 256 numbers (0-255, or 00000000-11111111). Turns out, there's a *lot* more characters than 256 used throughout the world's languages. Different countries created their own encoding conventions for their languages, and that mostly worked, but many encodings could only handle one language. That meant if you want to put the title of an American English book in the middle of a Thai sentence you were kind of in trouble. You'd need one encoding for Thai and one for English.

To solve this problem a group of people created Unicode. It sounds like "encode," and it is meant to be a "universal encoding" of all human languages. The solution Unicode provides is like the ASCII table, but it's huge by comparison. You can use 32 bits to encode a Unicode character, and that is more characters than we could possibly find. A 32-bit number means we can store 4,294,967,295 characters (2^32), which is enough space for every possible human language and probably a lot of alien ones too. Right now we use the extra space for important things like poop and smile emojis.

We now have a convention for encoding any characters we want, but 32 bits is 4 bytes (32/8 == 4) which means there is so much wasted space in most text we want to encode. We can also use 16 bits (2 bytes), but still there's going to be wasted space in most text. The solution is to use a clever convention to encode most common characters using 8 bits, and then "escape" into larger numbers when we need to encode more characters. That means we have one more convention that is nothing more than a compression encoding, making it possible for most common characters to use 8 bits and then escape out into 16 or 32 bits as needed.

The convention for encoding text in Python is called "UTF-8", which means "Unicode Transformation Format 8 Bits." It is a convention for encoding Unicode characters into sequences of bytes, which are sequences of bits, which turn sequences of switches on and off. You can also use other conventions (encodings), but UTF-8 is the current standard.