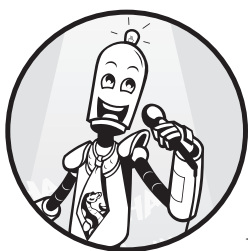# 2

PYTHON TRICKS

For our purposes, a *trick* is a way of accomplishing a task in a surprisingly fast or easy manner. In this book, you'll learn a wide variety of tricks and techniques to make your code more concise, while boosting your speed of implementation. While all technical chapters in this book show you Python tricks, this chapter addresses the low-hanging fruit: tricks you can adopt quickly and effortlessly, but with great effect on your coding productivity.

This chapter also serves as a stepping-stone for the more advanced chapters that follow. You need to understand the skills introduced in these one-liners to understand those that follow. Notably, we'll cover a range of basic Python functionality to help you write effective code, including list comprehension, file access, the map() function, the lambda function, the reduce() function, slicing, slice assignments, generator functions, and the zip() function.

If you're already an advanced programmer, you could skim over this chapter and decide which individual parts you want to study in more depth—and which ones you already understand well.

Using List Comprehension to Find Top Earners

In this section, you'll learn a beautiful, powerful, and highly efficient Python feature to create lists: list comprehension. You'll use list comprehension in many of the one-liners to come.

The Basics

Say you work in the human resources department of a large company and need to find all staff members who earn at least $100,000 per year. Your desired output is a list of tuples, each consisting of two values: the employee name and the employee's yearly salary. Here's the code you develop:

```python
employees = {'Alice' : 100000,
             'Bob' : 99817,
             'Carol' : 122908,
             'Frank' : 88123,
             'Eve' : 93121}


top_earners = []
for key, val in employees.items():
    if val >= 100000:
        top_earners.append((key,val))

print(top_earners)
# [('Alice', 100000), ('Carol', 122908)]
```

While the code is correct, there's an easier and much more concise—and therefore more readable—way of accomplishing the same result. All things being equal, the solution with *fewer lines* allows the reader to grasp the meaning of code faster.

Python offers a powerful way of creating new lists: *list comprehension*. The simple formula is as follows:

```
[ expression + context ]
```

The enclosing brackets indicate that the result is a new list. The *context* defines which list elements to select. The *expression* defines how to modify each list element before adding the result to the list. Here's an example:

```
[x * 2 for x in range(3)]
```

The bold part of the equation, `for x in range(3)`, is the context and the remaining part x * 2, is the expression. Roughly speaking, the expression doubles the values 0, 1, 2 generated by the context. Thus, the list comprehension results in the following list:

```
[0, 2, 4]
```

Both the expression and the context can be arbitrarily complicated. The expression may be a function of any variable defined in the context and may perform any computation—it can even call outside functions. The goal of the expression is to modify each list element before adding it to the new list.

The context can consist of one or many variables defined using one or many nested `for` loops. You can also restrict the context by using `if` statements. In this case, a new value will be added to the list only if the user-defined condition holds.

List comprehension is best explained by example. Study the following examples carefully and you'll get a good sense of list comprehension:

```
print([❶x ❷for x in range(5)])
# [0, 1, 2, 3, 4]
```

Expression ❶: Identity function (does not change the context variable x).
Context ❷: Context variable x takes all values returned by the range function: 0, 1, 2, 3, 4.

```
print([❶(x, y) ❷for x in range(3) for y in range(3)])
# [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

Expression ❶: Create a new tuple from the context variables x and y.
Context ❷: The context variable x iterates over all values returned by the range function (0, 1, 2), while context variable y iterates over all values returned by the range function (0, 1, 2). The two `for` loops are nested, so the context variable y repeats its iteration procedure for every single value of the context variable x. Thus, there are $3 \times 3 = 9$ combinations of context variables.

```
print([❶x ** 2 ❷for x in range(10) if x % 2 > 0])
# [1, 9, 25, 49, 81]
```

Expression ❶: Square function on the context variable x.
Context ❷: Context variable x iterates over all values returned by the range function—0, 1, 2, 3, 4, 5, 6, 7, 8, 9—but only if they are odd values; that is, x % 2 > 0.

```
print([❶x.lower() ❷for x in ['I', 'AM', 'NOT', 'SHOUTING']])
# ['i', 'am', 'not', 'shouting']
```

Expression ❶: String lowercase function on context variable x.

Context ❷: Context variable x iterates over all string values in the list: `'I'`, `'AM'`, `'NOT'`, `'SHOUTING'`.

Now, you should be able to understand the following code snippet.

The Code

Let's consider the same employee salary problem introduced earlier: given a dictionary with string keys and integer values, create a new list of (key, value) tuples so that the value associated with the key is larger than or equal to 100,000. Listing 2-1 shows the code.

```
## Data
employees = {'Alice' : 100000,
             'Bob' : 99817,
             'Carol' : 122908,
             'Frank' : 88123,
             'Eve' : 93121}


## One-Liner
top_earners = [(k, v) for k, v in employees.items() if v >= 100000]


## Result
print(top_earners)
```

Listing 2-1: One-liner solution for list comprehension

What's the output of this code snippet?

How It Works

Let's examine the one-liner:

```
top_earners = [ ❶(k, v) ❷for k, v in employees.items() if v >= 100000]
```

Expression ❶: Creates a simple (key, value) tuple for context variables k and v.

Context ❷: The dictionary method `dict.items()` ensures that context variable k iterates over all dictionary keys and that context variable v iterates over the associated values for context variable k—but only if the value of context variable v is larger than or equal to 100,000 as ensured by the `if` condition.

The result of the one-liner is as follows:

```
print(top_earners)
# [('Alice', 100000), ('Carol', 122908)]
```

This simple one-liner program introduces the important concept of *list comprehension*. We use list comprehension in multiple instances in this book, so make sure that you understand the examples in this section before moving on.

Using List Comprehension to Find Words with High Information Value

In this one-liner, you'll dive even deeper into the powerful feature of list comprehension.

The Basics

Search engines rank textual information according to its relevance to a user query. To accomplish this, search engines analyze the content of the text to be searched. All text consists of words. Some words provide a lot of information about the content of the text—and others don't. Examples for the former are words like *white, whale, Captain, Ahab* (Do you know the text?). Examples for the latter are words like *is, to, as, the, a,* or *how,* because most texts contain those words. Filtering out words that don't contribute a lot of meaning is common practice when implementing search engines. A simple heuristic is to filter out all words with three characters or less.

The Code

Our goal is to solve the following problem: given a multiline string, create a list of lists—each consisting of all the words in a line that have more than three characters. Listing 2-2 provides the data and the solution.

```
## Data
text = '''
Call me Ishmael. Some years ago - never mind how long precisely - having
little or no money in my purse, and nothing particular to interest me
on shore, I thought I would sail about a little and see the watery part
of the world. It is a way I have of driving off the spleen, and regulating
the circulation. - Moby Dick'''


## One-Liner
w = [[x for x in line.split() if len(x)>3] for line in text.split('\n')]


## Result
print(w)
```

Listing 2-2: One-liner solution to find words with high information value

What's the output of this code?

How It Works

The one-liner creates a list of lists by using two nested list comprehension expressions:

- The inner list comprehension expression `[x for x in line.split() if len(x)>3]` uses the string `split()` function to divide a given line into a sequence of words. We iterate over all words `x` and add them to the list if they have more than three characters.

- The outer list comprehension expression creates the string `line` used in the previous statement. Again, it uses the `split()` function to divide the text on the newline characters `'\n'`.

Of course, you need to get used to thinking in terms of list comprehensions, so the meaning may not come naturally to you. But after reading this book, list comprehensions will be your bread and butter—and you'll quickly read and write Pythonic code like this.

Reading a File

In this section, you'll read a file and store the result as a list of strings (one string per line). You'll also remove any leading and trailing whitespaces from the lines.

The Basics

In Python, reading a file is straightforward but usually takes a few lines of code (and one or two Google searches) to accomplish. Here's one standard way of reading a file in Python:

```python
filename = "readFileDefault.py" # this code

f = open(filename)
lines = []
for line in f:
    lines.append(line.strip())

print(lines)
"""
['filename = "readFileDefault.py" # this code',
'',
'f = open(filename)',
'lines = []',
'for line in f:',
'lines.append(line.strip())',
'',
'print(lines)']
"""
```

The code assumes that you've stored this code snippet in a file named *readFileDefault.py* in a folder. The code then opens this file, creates an empty list, `lines`, and fills the list with strings by using the `append()` operation in the `for` loop body to iterate over all the lines in the file. You also use the string method `strip()` to remove any leading or trailing whitespace (otherwise, the newline character `'\n'` would appear in the strings).

To access files on your computer, you need to know how to open and close files. You can access a file's data only after you've opened it. After closing the file, you can be sure that the data was written into the file. Python may create a buffer and wait for a while before it writes the whole buffer into the file (Figure 2-1). The reason for this is simple: file access is slow. For efficiency reasons, Python avoids writing every single bit independently. Instead, it waits until the buffer has filled with enough bytes and then flushes the whole buffer at once into the file.
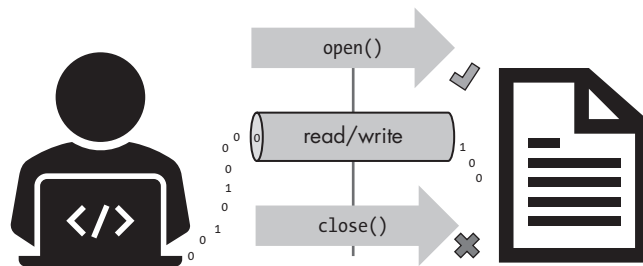

Figure 2-1: Opening and closing a file in Python

That's why it's good practice to close the file after reading it with the command `f.close()`, to ensure all the data is properly written into the file instead of residing in temporary memory. However, in a few exceptions, Python closes the file automatically: one of these exceptions occurs when the reference count drops to zero, as you'll see in the following code.

The Code

Our goal is to open a file, read all lines, strip the leading and trailing whitespace characters, and store the result in a list. Listing 2-3 provides the one-liner.

```
print([line.strip() for line in open("readFile.py")])
```

Listing 2-3: One-liner solution to read a file line by line.

Go ahead and guess the output of this code snippet before reading on.

How It Works

You use the `print()` statement to print the resulting list to the shell. You create the list by using list comprehension (see "Using List Comprehension to Find Top Earners" on page 18). In the *expression* part of the list comprehension, you use the `strip()` method of string objects.

The *context* part of the list comprehension iterates over all lines in the file.

The output of the one-liner is simply the one-liner itself (because it reads its Python source code file with the name *readFile.py*), wrapped into a string and filled into a list:

```
print([line.strip() for line in open("readFile.py")])
# ['print([line.strip() for line in open("readFile.py")])']
```

This section demonstrates that by making code shorter and more concise, you make it more readable without compromising efficiency.

Using Lambda and Map Functions

This section introduces two important Python features: the `lambda` and `map()` functions. Both functions are valuable tools in your Python toolbox. You'll use these functions to search a list of strings for occurrences of another string.

The Basics

In Chapter 1, you learned how to define a new function with the expression `def x`, followed by the content of the function. However, this is not the only way of defining a function in Python. You can also use *lambda functions* to define a simple function *with a return value* (the return value can be any object, including tuples, lists, and sets). In other words, every lambda function returns an object value to its calling environment. Note that this poses a practical restriction to lambda functions, because unlike standard functions, they are not designed to execute code *without* returning an object value to the calling environment.

NOTE *We already covered lambda functions in Chapter 1, but because it's such an important concept used throughout this book, we'll take a deeper look in this section.*

Lambda functions allow you to define a new function in a single line by using the keyword `lambda`. This is useful when you want to quickly create a function that you'll use only once and can be garbage-collected immediately afterward. Let's first study the exact syntax of lambda functions:

```
lambda arguments : return expression
```

You start the function definition with the keyword `lambda`, followed by a sequence of function arguments. When calling the function, the caller must provide these arguments. You then include a colon (`:`) and the *return expression*, which calculates the return value based on the arguments of the lambda function. The return expression calculates the function output and

can be any Python expression. Consider the following function definition as an example:

```
lambda x, y: x + y
```

The lambda function has two arguments, x and y. The return value is simply the sum of both arguments, x + y.

You typically use a lambda function when you call the function only once and can easily define it in a single line of code. One common example is using lambda with the map() function that takes as input arguments a function object f and a sequence s. The map() function then applies the function f on each element in the sequence s. Of course, you *could* define a full-fledged named function to define the function argument f. But this is often inconvenient and reduces readability—especially if the function is short and you need it only once—so it's usually best to use a lambda function here.

Before presenting the one-liner, I'll quickly introduce another small Python trick that makes your life easier: checking whether string x contains substring y by using the expression y in x. This statement returns True if there exists at least one occurrence of the string y in the string x. For example, the expression '42' in 'The answer is 42' evaluates to True, while the expression '21' in 'The answer is 42' evaluates to False.

Now let's look at our one-liner.

The Code

When given a list of strings, our next one-liner (Listing 2-4) creates a new list of tuples, each consisting of a Boolean value and the original string. The Boolean value indicates whether the string 'anonymous' appears in the original string! We call the resulting list mark because the Boolean values *mark* the string elements in the list that contain the string 'anonymous'.

```
## Data
txt = ['lambda functions are anonymous functions.',
       'anonymous functions dont have a name.',
       'functions are objects in Python.']


## One-Liner
mark = map(lambda s: (True, s) if 'anonymous' in s else (False, s), txt)


## Result
print(list(mark))
```

Listing 2-4: One-liner solution to mark strings that contain the string 'anonymous'

What's the output of this code?

How It Works

The map() function adds a Boolean value to each string element in the original txt list. This Boolean value is True if the string element contains the word *anonymous*. The first argument is the anonymous lambda function, and the second is a list of strings you want to check for the desired string.

You use the lambda return expression (True, s) if 'anonymous' in s else (False, s) to search for the 'anonymous' string. The value s is the input argument of the lambda function, which, in this example, is a string. If the string query 'anonymous' exists in the string, the expression returns the tuple (True, s). Otherwise, it returns the tuple (False, s).

The result of the one-liner is the following:

```
## Result
print(list(mark))
# [(True, 'lambda functions are anonymous functions.'),
# (True, 'anonymous functions dont have a name.'),
# (False, 'functions are objects in Python.')]
```

The Boolean values indicate that only the first two strings in the list contain the substring 'anonymous'.

You'll find lambdas incredibly useful in the upcoming one-liners. You're also making consistent progress toward your goal: understanding every single line of Python code you'll encounter in practice.

EXERCISE 2-1

Use list comprehension rather than the map() function to accomplish the same output. (You can find the solution at the end of this chapter.)

Using Slicing to Extract Matching Substring Environments

This section teaches you the important basic concept of *slicing*—the process of carving out a subsequence from an original full sequence—to process simple text queries. We'll search some text for a specific string, and then extract that string along with a handful of characters around it to give us context.

The Basics

Slicing is integral to a vast number of Python concepts and skills, both advanced and basic, such as when using any of Python's built-in data structures like lists, tuples, and strings. Slicing is also the basis of many advanced Python libraries such as NumPy, Pandas, TensorFlow, and scikit-learn. Studying slicing thoroughly will have a positive ripple effect throughout your career as a Python coder.

Slicing carves out subsequences of a sequence, such as a part of a string. The syntax is straightforward. Say you have a variable x that refers to a string, list, or tuple. You can carve out a subsequence by using the following notation:

```
x[start:stop:step].
```

The resulting subsequence starts at index start (included) and ends at index stop (excluded). You can include an optional third step argument that determines which elements are carved out, so you could choose to include just every step-th element. For example, the slicing operation x[1:4:1] used on variable x = 'hello world' results in the string 'ell'. Slicing operation x[1:4:2] on the same variable results in string 'el' because only every other element is taken into the resulting slice. Recall from Chapter 1 that the first element of any sequence type, such as strings and lists, has index 0 in Python.

If you don't include the step argument, Python assumes the default step size of one. For example, the slice call x[1:4] would result in the string 'ell'.

If you don't include the beginning or ending arguments, Python assumes you want to start at the start, or end at the end. For example, the slice call x[:4] would result in the string 'hell', and the slice call x[4:] would result in the string 'o world'.

Study the following examples to improve your intuitive understanding even further.

```
s = 'Eat more fruits!'

print(s[0:3])
# Eat
```
❶
```
print(s[3:0])
# (empty string '')

print(s[:5])
# Eat m

print(s[5:])
# ore fruits!
```
❷
```
print(s[:100])
# Eat more fruits!

print(s[4:8:2])
# mr
```
❸
```
print(s[::3])
# E rfi!
```
❹
```
print(s[::-1])
# !stiurf erom taE

print(s[6:1:-1])
# rom t
```

These variants of the basic [start:stop:step] pattern of Python slicing highlight the technique's many interesting properties:

- If start >= stop with a positive step size, the slice is empty ❶.
- If the stop argument is larger than the sequence length, Python will slice all the way to and including the rightmost element ❷.
- If the step size is positive, the default start is the leftmost element, and the default stop is the rightmost element (included) ❸.
- If the step size is negative (step < 0), the slice traverses the sequence in reverse order. With empty start and stop arguments, you slice from the rightmost element (included) to the leftmost element (included) ❹. Note that if the stop argument is given, the respective position is excluded from the slice.

Next, you'll use slicing along with the string.find(value) method to find the index of string argument value in a given string.

The Code

Our goal is to find a particular text query within a multiline string. You want to find the query in the text and return its immediate environment, up to 18 positions around the found query. Extracting the environment as well as the query is useful for seeing the textual context of the found string—just as Google presents text snippets around a searched keyword. In Listing 2-5, you're looking for the string 'SQL' in an Amazon letter to shareholders—with the immediate environment of up to 18 positions around the string 'SQL'.

```
## Data
letters_amazon = '''
We spent several years building our own database engine,
Amazon Aurora, a fully-managed MySQL and PostgreSQL-compatible
service with the same or better durability and availability as
the commercial engines, but at one-tenth of the cost. We were
not surprised when this worked.
'''

## One-Liner
find = lambda x, q: x[x.find(q)-18:x.find(q)+18] if q in x else -1

## Result
print(find(letters_amazon, 'SQL'))
```

Listing 2-5: One-liner solution to find strings in a text and their direct environment

Take a guess at the output of this code.

How It Works

You define a lambda function with two arguments: a string value x, and a query q to search for in the text. You assign the lambda function to the name find. The function find(x, q) finds the string query q in the string text x.

If the query q does not appear in the string x, you directly return the result -1. Otherwise, you use slicing on the text string to carve out the first occurrence of the query, plus 18 characters to the left of the query and 18 characters to the right, to capture the query's environment. You find that the index of the first occurrence of q in x is using the string function x.find(q). You call the function twice: to help determine the start index and the stop index of the slice, but both function calls return the same value because both the query q and the string x do not change. Although this code works perfectly fine, the redundant function call causes unnecessary computations—a disadvantage that could easily be fixed by adding a helper variable to temporarily store the result of the first function call. You could then reuse the result from the first function call by accessing the value in the helper variable.

This discussion highlights an important trade-off: by restricting yourself to one line of code, you cannot define and reuse a helper variable to store the index of the first occurrence of the query. Instead, you must execute the same function find to compute the start index (and decrement the result by 18 index positions) and to compute the end index (and increment the result by 18 index positions). In Chapter 5, you'll learn a more efficient way of searching patterns in strings (using regular expressions) that resolves this issue.

When searching for the query 'SQL' in Amazon's letter to shareholders, you find an occurrence of the query in the text:

```
## Result
print(find(letters_amazon, 'SQL'))
# a fully-managed MySQL and PostgreSQL
```

As a result, you get the string and a few words around it to provide context for the find. Slicing is a crucial element of your basic Python education. Let's deepen your understanding even more with another slicing one-liner.

Combining List Comprehension and Slicing

This section combines list comprehension and slicing to sample a two-dimensional data set. We aim to create a smaller but representative sample of data from a prohibitively large sample.

The Basics

Say you work as a financial analyst for a large bank and are training a new machine learning model for stock-price forecasting. You have a training data set of real-world stock prices. However, the data set is huge, and the model training seems to take forever on your computer. For example, it's common

in machine learning to test the prediction accuracy of your model for different sets of model parameters. In our application, say, you must wait for hours until the training program terminates (training highly complex models on large-scale data sets does in fact take hours). To speed things up, you reduce the data set by half by excluding every other stock-price data point. You don't expect this modification to decrease the model's accuracy significantly.

In this section, you'll use two Python features you learned about previously in this chapter: list comprehension and slicing. List comprehension allows you to iterate over each list element and modify it subsequently. Slicing allows you to select every other element from a given list quickly—and it lends itself naturally to simple filtering operations. Let's have a detailed look at how these two features can be used in combination.

The Code

Our goal is to create a new training data sample from our data—a list of lists, each consisting of six floats—by including only every other float value from the original data set. Take a look at Listing 2-6.

```
## Data (daily stock prices ($))
price = [[9.9, 9.8, 9.8, 9.4, 9.5, 9.7],
         [9.5, 9.4, 9.4, 9.3, 9.2, 9.1],
         [8.4, 7.9, 7.9, 8.1, 8.0, 8.0],
         [7.1, 5.9, 4.8, 4.8, 4.7, 3.9]]


## One-Liner
sample = [line[::2] for line in price]


## Result
print(sample)
```

Listing 2-6: One-liner solution to sample data

As usual, see if you can guess the output.

How It Works

Our solution is a two-step approach. First, you use list comprehension to iterate over all lines of the original list, price. Second, you create a new list of floats by slicing each line; you use line[start:stop:step] with default start and stop parameters and step size 2. The new list of floats consists of only three (instead of six) floats, resulting in the following array:

```
## Result
print(sample)
# [[9.9, 9.8, 9.5], [9.5, 9.4, 9.2], [8.4, 7.9, 8.0], [7.1, 4.8, 4.7]]
```

This one-liner using built-in Python functionality is not complicated. However, you'll learn about an even shorter version that uses the NumPy library for data science computations in Chapter 3.

Using Slice Assignment to Correct Corrupted Lists

This section shows you a powerful slicing feature in Python: slice assignments. *Slice assignments* use slicing notation *on the left-hand side* of an assignment operation to modify a subsequence of the original sequence.

The Basics

Imagine you work at a small internet startup that keeps track of its users' web browsers (Google Chrome, Firefox, Safari). You store the data in a database. To analyze the data, you load the gathered browser data into a large list of strings, but because of a bug in your tracking algorithm, every second string is corrupted and needs to be replaced by the correct string.

Assume that your web server always redirects the first web request of a user to another URL (this is a common practice in web development known under the HTML code 301: *moved permanently*). You conclude that the first browser value will be equal to the second one in most cases because the browser of a user stays the same while waiting for the redirection to occur. This means that you can easily reproduce the original data. Essentially, you want to duplicate every other string in the list: the list `['Firefox', 'corrupted', 'Chrome', 'corrupted']` becomes `['Firefox', 'Firefox', 'Chrome', 'Chrome']`.

How can you achieve this in a fast, readable, and efficient way (preferably in a single line of code)? Your first idea is to create a new list, iterate over the corrupted list, and add every noncorrupted browser twice to the new list. But you reject the idea because you'd then have to maintain two lists in your code—and each may have millions of entries. Also, this solution would require a few lines of code, which would hurt conciseness and readability of your source code.

Luckily, you've read about a beautiful Python feature: slice assignments. You'll use slice assignments to select and replace a *sequence of elements* between indices i and j by using the slicing notation `lst[i:j] = [0 0 ...0]`. Because you are using slicing `lst[i:j]` on the *left-hand side* of the assignment operation (rather than on the right-hand side as done previously), the feature is denoted as slice *assignments*.

The idea of slice assignments is simple: replace all selected elements in the original sequence on the left with the elements on the right.

The Code

Our goal is to replace every other string with the string immediately in front of it; see Listing 2-7.

```
## Data
visitors = ['Firefox', 'corrupted', 'Chrome', 'corrupted',
            'Safari', 'corrupted', 'Safari', 'corrupted',
            'Chrome', 'corrupted', 'Firefox', 'corrupted']


## One-Liner
visitors[1::2] = visitors[::2]


## Result
print(visitors)
```

Listing 2-7: One-liner solution to replace all corrupted strings

What's the fixed sequence of browsers in this code?

How It Works

The one-liner solution replaces the `'corrupted'` strings with the browser strings that precede them in the list. You use the slice assignment notation to access every corrupted element in the `visitors` list. I've highlighted the selected elements in the following code snippet:

```
visitors = ['Firefox', 'corrupted', 'Chrome', 'corrupted',
            'Safari', 'corrupted', 'Safari', 'corrupted',
            'Chrome', 'corrupted', 'Firefox', 'corrupted']
```

The code replaces these selected elements with the slice on the right of the assignment operation. These elements are highlighted in the following code snippet:

```
visitors = ['Firefox', 'corrupted', 'Chrome', 'corrupted',
            'Safari', 'corrupted', 'Safari', 'corrupted',
            'Chrome', 'corrupted', 'Firefox', 'corrupted']
```

The former elements are replaced by the latter. Therefore, the resulting `visitors` list is the following (highlighting the replaced elements):

```
## Result
print(visitors)
'''
['Firefox', 'Firefox', 'Chrome', 'Chrome',
'Safari', 'Safari', 'Safari', 'Safari',
'Chrome', 'Chrome', 'Firefox', 'Firefox']
'''
```

The result is the original list with each `'corrupted'` string replaced by its preceding browser string. This way, you clean the corrupted data set.

Using slice assignments for this problem is the quickest and most effective way of accomplishing your small task. Note that the cleaned data has nonbiased browser usage statistics: a browser with 70 percent market share in the corrupted data will maintain its 70 percent market share in the cleaned data. The cleaned data can then be used for further analysis—for example, to find out whether Safari users are better customers (after all, they tend to spend more money on hardware). You've learned a simple and concise way of modifying a list programmatically and in place.

Analyzing Cardiac Health Data with List Concatenation

In this section, you'll learn how to use list concatenation to repeatedly copy smaller lists and merge them into a larger list to generate cyclic data.

The Basics

This time, you're working on a small code project for a hospital. Your goal is to monitor and visualize the health statistics of patients by tracking their cardiac cycles. By plotting expected cardiac cycle data, you'll enable patients and doctors to monitor any deviation from that cycle. For example, given a series of measurements stored in the list [62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62] for a single cardiac cycle, you want to achieve the visualization in Figure 2-2.
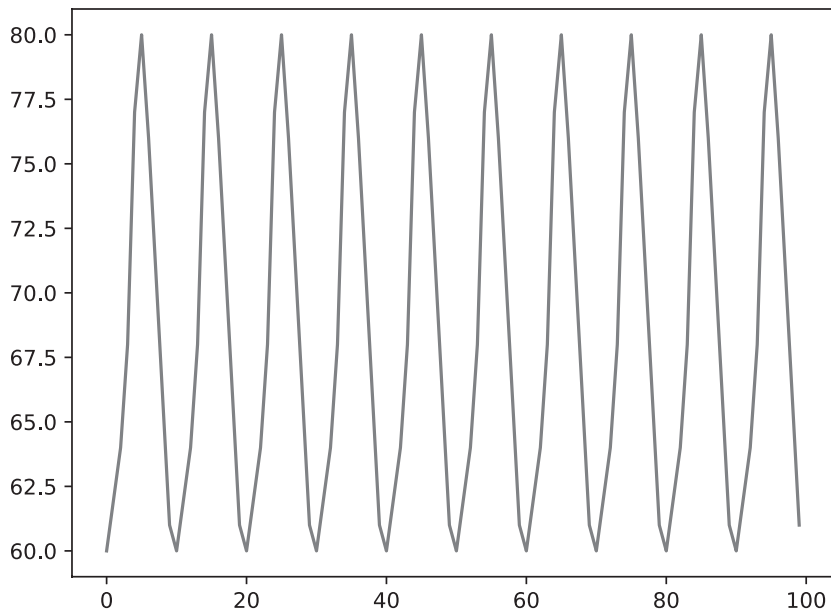
Figure 2-2: Visualizing expected cardiac cycles by copying selected values from the measured data

The problem is that the first and the last two data values in the list are redundant: [62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62]. This may have been useful when plotting only a single cardiac cycle to indicate that one full cycle has been visualized. However, we must get rid of this redundant data to ensure that our expected cardiac cycles do not look like the ones in Figure 2-3 when copying the same cardiac cycle.
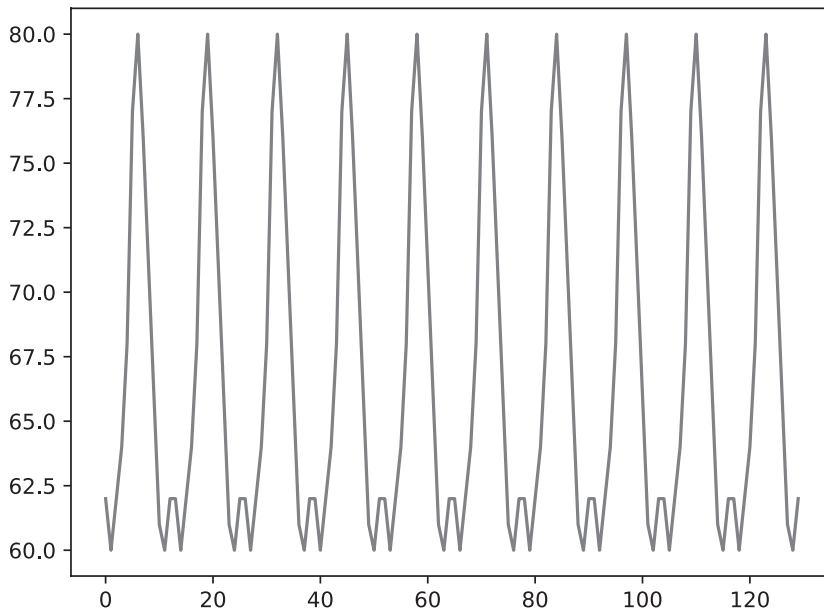

Figure 2-3: Visualizing expected cardiac cycles by copying all *values from the measured data (no filtering of redundant data)*

Clearly, you need to *clean* the original list by removing the redundant first and the last two data values: [62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62] becomes [60, 62, 64, 68, 77, 80, 76, 71, 66, 61].

You'll combine slicing with the new Python feature *list concatenation*, which creates a new list by *concatenating* (that is, *joining*) existing lists. For example, the operation [1, 2, 3] + [4, 5] generates the new list [1, 2, 3, 4, 5], but doesn't replace the original lists. You can use this with the * operator to concatenate the *same list* again and again to create large lists: for example, the operation [1, 2, 3] * 3 generates the new list [1, 2, 3, 1, 2, 3, 1, 2, 3].

In addition, you'll use the matplotlib.pyplot module to plot the cardiac data you generate. The matplotlib function plot(data) expects an iterable argument data—an *iterable* is simply an object over which you can iterate, such as a list—and uses it as y values for subsequent data points in a two-dimensional plot. Let's dive into the example.

The Code

Given a list of integers that reflect the measured cardiac cycle, you first want to clean the data by removing the first and last two values from the list. Second, you create a new list with expected future heart rates by copying the cardiac cycle to future time instances. Listing 2-8 shows the code.

```
## Dependencies
import matplotlib.pyplot as plt


## Data
cardiac_cycle = [62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62]


## One-Liner
expected_cycles = cardiac_cycle[1:-2] * 10


## Result
plt.plot(expected_cycles)
plt.show()
```

Listing 2-8: One-liner solution to predict heart rates at different times

Next, you'll learn about the result of this code snippet.

How It Works

This one-liner consists of two steps. First, you use slicing to clean the data by using the negative stop argument -2 to slice all the way to the right but skip the last two redundant values. Second, you concatenate the resulting data values 10 times by using the replication operator *. The result is a list of $10 \times 10 = 100$ integers made up of the concatenated cardiac cycle data. When you plot the result, you get the desired output shown previously in Figure 2-2.

Using Generator Expressions to Find Companies That Pay Below Minimum Wage

This section combines some of the Python basics you've already learned and introduces the useful function any().

The Basics

You work in law enforcement for the US Department of Labor, finding companies that pay below minimum wage so you can initiate further investigations. Like hungry dogs on the back of a meat truck, your Fair Labor Standards Act (FLSA) officers are already waiting for the list of companies that violated the minimum wage law. Can you give it to them?

Here's your weapon: Python's any() function, which takes an iterable, such as a list, and returns True if at least one element of the iterable evaluates to True. For example, the expression any([True, False, False, False]) evaluates to True, while the expression any([2<1, 3+2>5+5, 3-2<0, 0]) evaluates to False.

An interesting Python extension is a generalization of list comprehension: generator expressions. *Generator expressions* work exactly like list comprehensions—but without creating an actual list in memory. The numbers are created on the fly, without storing them explicitly in a list. For example, instead of using list comprehension to calculate the squares of the first 20 numbers, sum([x*x for x in range(20)]), you can use a generator expression: sum(x*x for x in range(20)).

The Code

Our data is a dictionary of dictionaries storing the hourly wages of company employees. You want to extract a list of the companies paying below your state's minimum wage (< \$9) for at least one employee; see Listing 2-9.

```
## Data
companies = {
    'CoolCompany' : {'Alice' : 33, 'Bob' : 28, 'Frank' : 29},
    'CheapCompany' : {'Ann' : 4, 'Lee' : 9, 'Chrisi' : 7},
    'SosoCompany' : {'Esther' : 38, 'Cole' : 8, 'Paris' : 18}}


## One-Liner
illegal = [x for x in companies if any(y<9 for y in companies[x].values())]


## Result
print(illegal)
```

Listing 2-9: One-liner solution to find companies that pay below minimum wage

Which companies must be further investigated?

How It Works

You use two generator expressions in this one-liner.

The first generator expression, y<9 for y in companies[x].values(), generates the input to the function any(). It checks each of the companies' employees to see whether they are being paid below minimum wage, y<9. The result is an iterable of Booleans. You use the dictionary function

values() to return the collection of values stored in the dictionary. For example, the expression companies['CoolCompany'].values() returns the collection of hourly wages dict_values([33, 28, 29]). If at least one of them is below minimum wage, the function any() would return True, and the company name x would be stored as a string in the resulting list illegal, as described next.

The second generator expression is the list comprehension [x for x in companies if any(...)] and it creates a list of company names for which the previous call of the function any() returns True. Those are the companies that pay below minimum wage. Note that the expression for x in companies visits all dictionary keys—the company names 'CoolCompany', 'CheapCompany', and 'SosoCompany'.

The result is therefore as follows:

```
## Result
print(illegal)
# ['CheapCompany', 'SosoCompany']
```

Two out of three companies must be investigated further because they pay too little money to at least one employee. Your officers can start to talk to Ann, Chrisi, and Cole!

Formatting Databases with the zip() Function

In this section, you'll learn how to apply database column names to a list of rows by using the zip() function.

The Basics

The zip() function takes iterables iter_1, iter_2, ..., iter_n and aggregates them into a single iterable by aligning the corresponding i-th values into a single tuple. The result is an *iterable* of tuples. For example, consider these two lists:

```
[1,2,3]
[4,5,6]
```

If you zip them together—after a simple data type conversion, as you'll see in a moment—you'll get a new list:

```
[(1,4), (2,5), (3,6)]
```

Unzipping them back into the original tuples requires two steps. First, you remove the outer square bracket of the result to get the following three tuples:

```
(1,4)
(2,5)
(3,6)
```

Then when you zip those together, you get the new list:

```
[(1,2,3), (4,5,6)]
```

So, you have your two original lists again! The following code snippet shows this process in full:

```
lst_1 = [1, 2, 3]
lst_2 = [4, 5, 6]

# Zip two lists together
zipped = list(zip(lst_1, lst_2))
print(zipped)
# [(1, 4), (2, 5), (3, 6)]

# Unzip to lists again
lst_1_new, lst_2_new = zip(❶*zipped)
print(list(lst_1_new))
print(list(lst_2_new))
```

You use the asterisk operator * to unpack ❶ all elements of the list. This operator removes the outer bracket of the list zipped so that the input to the zip() function consists of three iterables (the tuples (1, 4), (2, 5), (3, 6)). If you zip those iterables together, you package the first three tuple values 1, 2, and 3 into a new tuple, and the second three tuple values 4, 5, and 6 into another new tuple. Together, you get the resulting iterables (1, 2, 3) and (4, 5, 6), which is the original (unzipped) data.

Now, imagine you work in the IT branch of the controlling department of your company. You maintain the database of all employees with the column names: 'name', 'salary', and 'job'. However, your data is out of shape—it's a collection of rows in the form ('Bob', 99000, 'mid-level manager'). You want to associate your column names to each data entry to bring it into the readable form {'name': 'Bob', 'salary': 99000, 'job': 'mid-level manager'}. How can you achieve that?

The Code

Your data consists of the column names and the employee data organized as list of tuples (rows). Assign the column names to the rows and, thus, create a list of dictionaries. Each dictionary assigns the column names to the respective data values (Listing 2-10).

```
## Data
column_names = ['name', 'salary', 'job']
db_rows = [('Alice', 180000, 'data scientist'),
           ('Bob', 99000, 'mid-level manager'),
           ('Frank', 87000, 'CEO')]

## One-Liner
db = [dict(zip(column_names, row)) for row in db_rows]
```

```
## Result
print(db)
```

Listing 2-10: One-liner solution to apply a database format to a list of tuples

What's the printed format of the database db?

How It Works

You create the list by using list comprehension (see "Using List Comprehension to Find Top Earners" on page 18 for more on expression + context). The context consists of a tuple of every row in the variable db_rows. The expression zip(column_names, row) zips together the schema and each row. For example, the first element created by the list comprehension would be zip(['name', 'salary', 'job'], ('Alice', 180000, 'data scientist')), which results in a zip object that, after conversion to a list, is in the form [('name', 'Alice'), ('salary', 180000), ('job', 'data scientist')]. The elements are in (*key, value*) form so you can convert it into a dictionary by using the converter function dict() to arrive at the required database format.

NOTE *The zip() function doesn't care that one input is a list and the other is a tuple. The function requires only that the input is an iterable (and both lists and tuples are iterables).*

Here's the output of the one-liner code snippet:

```
## Result
print(db)
'''
[{'name': 'Alice', 'salary': 180000, 'job': 'data scientist'},
{'name': 'Bob', 'salary': 99000, 'job': 'mid-level manager'},
{'name': 'Frank', 'salary': 87000, 'job': 'CEO'}]
'''
```

Every data item is now associated with its name in a list of dictionaries. You've learned how to use the zip() function effectively.

Summary

In this chapter, you've mastered list comprehensions, file input, the functions lambda, map(), and zip(), the all() quantifier, slicing, and basic list arithmetic. You've also learned how to use and manipulate data structures to solve various day-to-day problems.

Converting data structures back and forth easily is a skill with a profound impact on your coding productivity. Rest assured that your programming productivity will soar as you increase your ability to quickly manipulate data. Small processing tasks like the ones you've seen in this chapter contribute significantly to the common "death by a thousand cuts": the overwhelming harm that performing many small tasks has on your overall productivity. By using the Python tricks, functions, and features

introduced in this chapter, you've obtained effective protection against those thousand cuts. Speaking metaphorically, the newly acquired tools help you recover from each cut much faster.

In the next chapter, you'll improve your data science skills even further by diving into a new set of tools provided by the NumPy library for numerical computations in Python.

SOLUTION TO EXERCISE 2-1

Here's how to use list comprehension instead of the map() function to achieve the same problem of filtering out all lines that contain the string 'anonymous'. In this case, I even recommend using the faster and cleaner list comprehension feature.

```
mark = [(True, s) if 'anonymous' in s else (False, s) for s in txt]
```