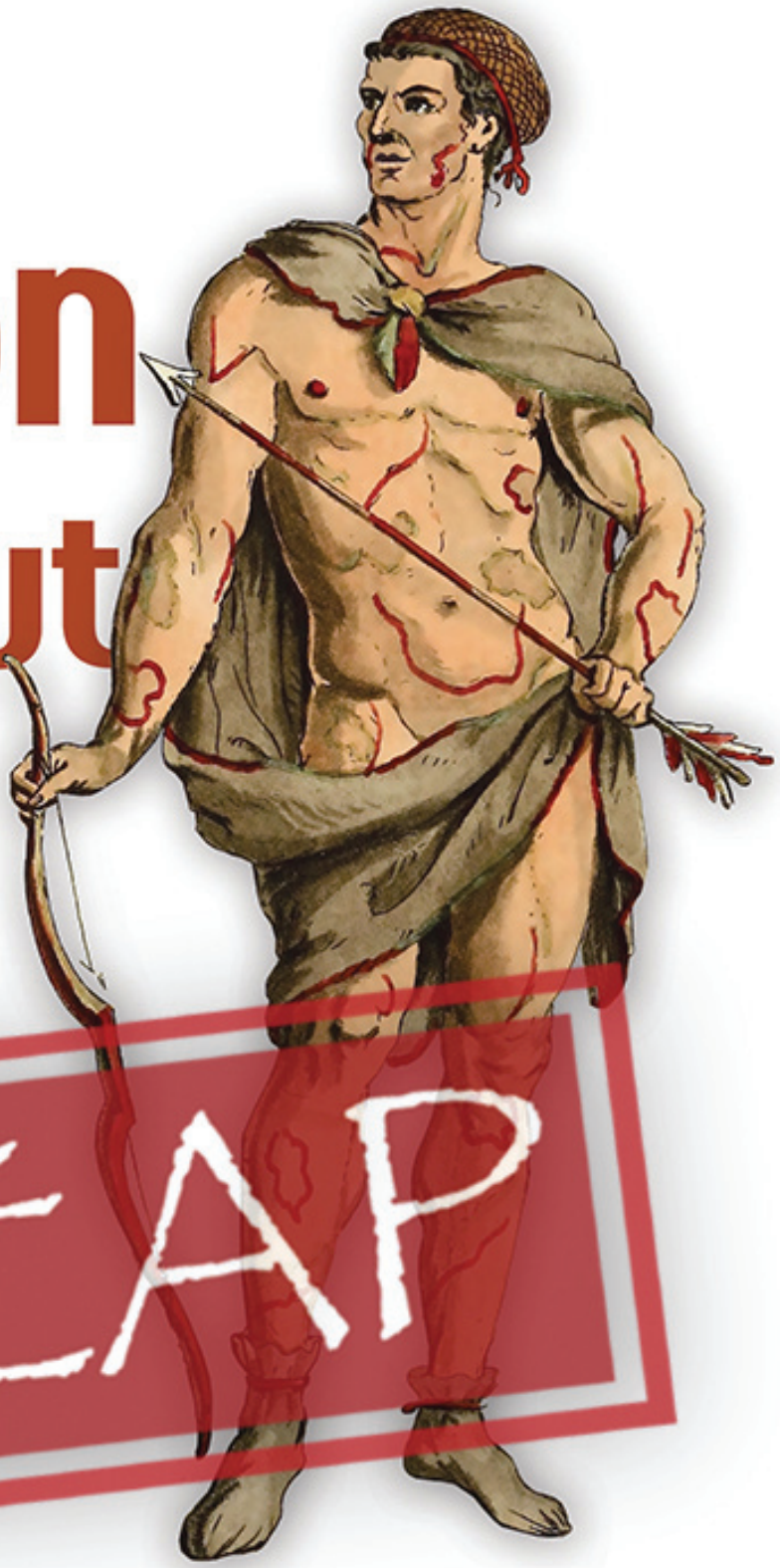
Solutions and discussions in text and video

Python Workout

50 Essential Exercises



Reuven M. Lerner

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Python Workout
50 Essential Exercises
Version 3

Copyright 2019 Manning Publications

For more information on this and other Manning titles go to
manning.com

©Manning Publications Co. To comment go to liveBook

Licensed to Shai Asher <shaia@outlook.com>

welcome

Hi, and thanks so much for purchasing the MEAP of the *Python Workout: 50 Essential Exercises*!

Over the last few years, Python has made inroads into many areas, including system administration, data science, devops, text processing, and Web development. As such, there are now many courses and books that aim to teach you Python.

Indeed, I spend just about every day teaching Python at companies around the world. It's fun and exciting to see so many people learning Python, getting more done in less time — and enjoying themselves along the way.

For many years, my students would consistently ask one question after taking my courses: Where can we go to practice the Python we've just learned?

It shouldn't surprise me that they would want such practice. Athletes don't just hop onto the field and start to play, even if they're good at what they do; they're constantly improving and sharpening their skills. Even if you understood everything that the teacher said in your high-school math class, you still needed to practice, in order to ensure that it sank in. And anyone who has learned a foreign language knows that there's a world of difference between speaking in class, and using it in a real-world setting.

And indeed, learning a programming language is much like learning a human language: You need to understand its underlying structure, its nouns and verbs, and the ways in which you can and cannot express yourself. Over time, these rules become natural, and you find yourself able to express more complex ideas — without having to refer to a translation guide, such as a phrasebook or Stack Overflow.

My goal in this book is to reinforce the ideas that you learned in your initial Python course, or when reading your first Python book. As you do each exercise, you'll probably struggle and feel some frustration. And when you see my solutions, you might well say, "Oh, of course! Why didn't I think of that?" These are normal parts of the learning process. Better to experience them in this book, as you gain fluency, than at work, on a problem that requires your attention.

The book is divided into 10 chapters, each covering a different area of Python. Overall, the problems get more complex as the book progresses — but that doesn't mean I'll wait to use functions until the "functions" chapter, or comprehensions until the "functional programming" chapter. You should use whatever tool you think is necessary.

Each chapter starts with a reference table, including links, pointing to resources that might help you to better understand topics that you've forgotten or never learned. I hope that these will help you to reinforce your Python understanding, beyond the book itself.

In addition to the exercises and solutions, there are also videos, in which I demonstrate solving each of the problems, and add to the discussion I present in the book. The solutions are the same as the ones you can read in the book, but for many people, the videos make the

©Manning Publications Co. To comment go to liveBook

Licensed to Shai Asher <shaia@outlook.com>

process come alive a bit more; you'll get more of the feel of being in one of my live training sessions. In addition, it's often helpful to see the coding and solving process happen over time, rather than seeing it all at once on the page. The best way to use the video segments is to have the book and the video (in Manning's liveVideo platform) open at the same time. For each exercise, start with the book, complete each exercise, and, if you like, read the solution and discussion. Then click on the corresponding video segment to see me demo the solution and offer further insights to the discussion.

If there's one request that I have from you, it's that you not look at the exercise, think about how to solve it, and look at the solution. The real and deep learning doesn't happen that way, much as we might like to think otherwise. Only through actually trying to answer the exercise will you truly learn and improve your fluency.

When you complete an exercise, don't just read the solution, either. Just as important as seeing the code is understanding why and how I arrived at that code. The process I use for finding a solution is a key factor in your learning. I've thus not only spelled out my process, but also provided (whenever possible) with a link to the code in the Python Tutor, a fantastic online resource for anyone teaching or learning Python. I encourage you to use this link to experiment with the code, and get a visual sense of the data structures involved.

The book reached its present form thanks to questions, suggestions, and feedback from the students I've taught around the world. If you find a bug, either in the code or in the explanations, then please don't hesitate to be in touch, using the liveBook Discussion Forum. I'll be delighted to hear from you, and to incorporate your suggestions into this book, as well as the exercises I continue to develop and use in my teaching.

—Reuven M. Lerner

©Manning Publications Co. To comment go to liveBook

Licensed to Shai Asher <shaia@outlook.com>

How to use the video supplements

Note: The current videos reference Python 2; they will be replaced with videos that use Python 3 and have improved audio and video quality.

Each project in this book comes with a companion video. These video modules provide a demonstration of the solution presented in the book, as well as additional explanation and insight. These videos are optional but extremely valuable discussions along with screencasts of the coding. They are the next best thing to being in the room at one of Reuven's live training sessions.

For now, the videos are available in Manning's liveBook platform. If you are reading this book in another format (pdf, ebook, etc), to access the videos, please log into your Manning account and navigate to the liveBook format.

To make the most of the projects, remember, the best learning comes from doing! So start with the book, reading the introduction to a chapter, to get a general understanding of the topic and the resources you might use to complete each project . Then read the problem, work through it, come up with a solution. Compare your solution to the answer, and read through the discussion to see how Reuven got to his answer.

Now, turn to the video for a demonstration of Reuven working through the problem and demonstrating how he arrived at the solution.

What's most important is that you work out your own solution before you check the answers in the book or watch the explanations in the videos. Again, the best learning comes from doing!

©Manning Publications Co. To comment go to liveBook

Licensed to Shai Asher <shaia@outlook.com>

brief contents

©Manning Publications Co. To comment go to liveBook

Licensed to Shai Asher <shaia@outlook.com>

1 Numeric types

In Python, we have three different numeric types: `int`, `float`, and `complex`. For most of us, it's enough to know about (and work with) `int` (for whole numbers) and `float` (for numbers with a fractional component).

It's hard to describe how often programs will use numbers. Whether you're calculating salaries, bank interest, or cellular frequencies, it's hard to imagine a program that doesn't use numbers in some way or another.

Numbers are not only fundamental to programming, but they also give us a good introduction to how a programming language operates. Understanding how variable assignment and function arguments work with integers and floats will help you to reason about more complex types, such as strings, tuples, and dicts.

This chapter contains exercises that work with numbers, as inputs and as outputs. While working with numbers is fairly basic and straightforward, the conversions between them, and the integration with other data types, can sometimes take some time to get used to.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

1.1 Useful references

Table 1.1 Reference table

Name	Description	Example	Link
random	module for generating random numbers and selecting random elements	number = random.randint(1, 100)	random.html
comparisons	operators for comparing values	x < y	comparisons.html
f-strings	strings into which expressions can be interpolated	f"It is currently {datetime.datetime.now()}"	python.org/dev/peps/pep-0498/ and #f-strings
for loops	iterate over the elements of an iterable	for i in range(10): print(i*i)	#for-statements
`input`	prompt the user to enter a string, and return a string	input("Enter your name: ")	#input
enumerate	helps us to number elements of iterables	for index, item in enumerate('abc'): print(f"{index}: {item}")	#enumerate
reversed	returns an iterator with the reversed elements of an iterable	r = reversed('abcd')	#reversed

1.2 Number guessing game

This first exercise is designed to get your feet (well, fingers) warmed up for the rest of the book. It also introduces a number of topics that will repeat themselves over your Python career: Loops, user input, converting types, and comparing values.

More specifically: Programs all have to get input in order to do something interesting, and that input often comes from the user. Knowing how to ask the user for input is thus not only useful, but allows us to think about the type of data we're getting, and what (and how) to convert it into a format we can use.

Python only provides two kinds of loops, `for` and `while`, and they appear in nearly every Python program. Knowing how to write and use loops will serve you well throughout your Python career. The fact that nearly every type of data knows how to work inside of a `for` loop means that they're useful far beyond the built-in iterable types. If you're working with database records, elements in an XML file, or the results from searching for text using regular expressions, you'll be using "for" loops quite a bit.

For this exercise:

- Write a program that chooses a random integer between 0 and 100 (inclusive).
- Then ask the user to guess what number has been chosen.
- Each time the user enters a guess, the program indicates one of:
  - Too high
  - Too low
  - Just right

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

- If the user guessed correctly, then the program exits. Otherwise, the user is asked to try again.
- The program only exits after the user guesses correctly.

We'll use the `randint` (docs.python.org/3/library/random.html#random.randint) function in the `random` module to generate a random number. Thus, you can say:

```
import random
number = random.randint(10, 30)
```

and `number` will contain an integer from 10 to (and including) 30.

We'll also be prompting the user to enter text with the `input` function. We'll actually be using `input` quite a bit in this book, to ask the user to tell us something. The function takes a single string as an argument, which is displayed to the user. The function then returns the string containing whatever the user entered. For example:

```
name = input("Enter your name: ")
print(f'Hello, {name}!')
```

NOTE If the user simply presses Enter when presented with the `input` prompt, the value returned by `input` is an empty string, not `None`. Indeed, the return value from `input` will always be a string, regardless of what the user entered.

NOTE In Python 2, you would ask the user for input using the `raw_input` function. Python 2's `input` function was considered dangerous, since it would ask the user for input and then evaluate the resulting string using the `eval` function. In Python 3, the dangerous function has gone away, and the safe one has been renamed `input`.

1.2.1 Solution

```
import random
answer = random.randint(0, 100)

while True:
    user_guess = int(input("What is your guess? "))  ❶

    if user_guess == answer:
        print(f"Right!  The answer is {user_guess}")
        break

    elif user_guess < answer:
        print(f"Your guess of {user_guess} is too low!")

    elif user_guess > answer:
        print(f"Your guess of {user_guess} is too high!")
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

❶ The user could enter non-digits, which would crash the program. We're going to assume, for the purposes of this exercise, that our user will only enter valid data, namely integers. Remember that the `int` function normally assumes that we're giving it a decimal number, which means that its argument may contain only digits. If you really want to be pedantic, you can use the `str.isdigit` method to check that a string contains only digits. Or you can trap the `ValueError` exception you'll get if you run `int` on something that cannot be turned into an integer.

1.2.2 Discussion

At its heart, this program is a simple application of the comparison operators (==, <, and >) to a number, such that a user can guess the random integer that the computer has chosen. However, there are several aspects of this program that merit discussion.

First and foremost, we use the `random` module to generate a random number. After importing `random`, we can then invoke `random.randint`, which takes two parameters, returning a random integer. In general, the `random` module is a useful tool whenever you need to choose a random value.

Note that the maximum number in `random.randint` is inclusive. This is unusual in Python; most of the time, such ranges in Python are exclusive, meaning that the higher number is not included.

> **TIP** The `random` module doesn't just generate random numbers. It also has functions to choose one or more elements from a Python sequence.

Now that the computer has chosen a number, it is the user's turn to guess what that number is. Here, we start an infinite loop in Python, which is most easily created with `while True`. Of course, it is important that there be a way to break out of the loop; in this case, it will be when the user correctly guesses the value of `answer`. When that happens, `break` command is used to exit from the innermost loop.

The `input` (docs.python.org/3/library/functions.html#input) function always returns a string. This means that if we want to guess a number, we must turn the user's input string into an integer. This is done in the same way as all conversions in Python, by using the target type as a function, passing the source value as a parameter. Thus `int('5')` will return the integer 5, whereas `str(5)` will return the string '5'. You can also create new instances of more complex types by invoking the class as a function, as in `list('abc')` or `dict([('a', 1), ('b', 2), ('c', 3)])`.

In Python 3, you cannot use < and > to compare different types. If you neglect to turn the user's input into an integer, the program will exit with an error, saying that it cannot compare a string

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

(i.e., the user's input) with an integer.

NOTE In Python 2, it wasn't an error to compare objects of different types. But the results you would get were a bit surprising, if you didn't know what to expect. That's because Python would first compare them by type, and then compare them within that type. In other words: All integers were smaller than all lists, and all lists were smaller than all strings.

Why would you ever want to use < and > on objects of different types? You probably wouldn't want to do so, and I found that this functionality confused people more than helped them. In Python 3, you cannot make such a comparison; trying to check with `1 < [10, 20, 30]` will result in a `TypeError` exception.

In this exercise, and the rest of this book, I use "f-strings" to insert values from variables into our strings. I am a big fan of "f-strings," and encourage you to try them, as well. (See the sidebar discussing f-strings, below.)

1.2.3 Beyond the exercise

You'll often be getting input from users, which comes as a string. This means that you'll often need to convert it into other types, such as (in this exercise) integers. Here are some additional ideas for ways to practice this idea:

1. Modify this program, such that it gives the user only 3 chances to guess the correct number. If they try three times without success, they're told that they didn't guess in time, and the program exits.
2. Not only should you choose a random number, but you should also choose a random number base, from 2 to 16, in which the user should submit their input. If the user inputs "10" as their guess, you'll need to interpret it in the correct number base; "10" might mean 10 (decimal) or 2 (binary) or 16 (hexadecimal).
3. Try the same thing, but have the program choose a random word from the dictionary, and then ask the user to guess the word. (You might want to limit yourself to words containing between 2-5 letters, to avoid making it too horribly diffciult.) Instead of telling the user that they should guess a smaller or larger number, have them choose an earlier or later word in the dictionary.

SIDEBAR **f-strings**

Many people, when doing the "number guessing game" exercise, try to print a combination of a string and a number, such as "You guessed 5". They quickly discover that Python doesn't allow you to add (using +) strings and integers. How, then, can you include both types in the same line of output?

This problem has long troubled newcomers to Python from other languages. The earliest method was to use the % operator on a string:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

```
"Hello, %s" % "world"
```

While C programmers rejoiced at having something that worked like `printf`, everyone else found this technique to be frustrating. Among other things, `%` wasn't super-intuitive for new developers, forced you to use parentheses when passing more than one argument, and didn't let you reference repeated values easily.

It was thus a vast improvement when the `str.format` method was introduced into Python, letting us say:

```
"Hello, {0}".format("world")
```

While I loved the use of `str.format`, many newcomers to Python found it a bit hard to use and very long. In particular, they didn't like the idea of referencing variables on the left and giving values on the right. And the syntax inside of the curly braces was unique to Python, which was frustrating for all.

Python 3.6 introduced "f-strings," which are similar to the sort of double-quoted strings programmers in Perl, PHP, Ruby, and Unix shells have enjoyed for decades. f-strings work basically the same way as `str.format`, but without having to pass parameters:

```
name = "world"
f"Hello, {name}"
```

It's actually even better than that: You can put whatever expression you want inside of the curly braces, and it'll be evaluated when the string is evaluated. For example:

```
name = "world"
x = 100
y = 'abcd'
f"x * 2 = {x*2), and y.capitalize() is {y.capitalize()}"
```

You can also affect the formatting of each data type by putting a code after a colon (`:`) inside of the curly braces. For example, you can force the string to be aligned left or right, on a field of 10 hash marks (`#`), with the following:

```
name = "world"
first = "Reuven"
last = "Lerner"
f"Hello, {first:#<10} {last:#>10}"  ❶
```

❶ The format code `#<10` means that the string should be placed, left aligned, in a field of 10 characters, with `#` placed wherever the word doesn't fill it. The format code `#>10` means the same thing, but right aligned.

I definitely encourage you to take a look at f-strings and to use them. They're one of my favorite changes to Python in the last few years.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

What if you're still using Python 2, and cannot use f-strings? Then you can and should still use `str.format`, a string method that works approximately the same way, but with less flexibility. Plus, you have to call the method, and reference the arguments by number of name.

1.3 Summing numbers

One of my favorite types of exercises involves re-implementing functionality that we've seen elsewhere, either inside of Python or in Unix. That's the background for this next exercise, in which you'll re-implement the `sum` (docs.python.org/3/library/functions.html#sum) function that comes with Python. That function takes a sequence of numbers, and returns the sum of those numbers. So if you were to invoke `sum([1,2,3])`, the result would be `6`.

The challenge here is to write a `mysum` function that does the same thing as the built-in `sum` function. However, instead of taking a single sequence as a parameter, it should take a variable number of arguments. Thus while we might invoke `sum([1,2,3])`, we would instead invoke `mysum(1,2,3)` or `mysum(10,20,30,40,50)`.

NOTE The built-in `sum` function takes an optional second argument, which we're ignoring here.

And no, you should not use the built-in `sum` function to accomplish this! (You would be amazed just how often someone asks me this question when teaching courses.)

This exercise is meant to not only help you think about numbers, but also the design of functions. And in particular, you should think about the types of parameters functions can take in Python. In many languages, functions can be defined multiple times, each with a different type signature (i.e., number of parameters, and parameter types). In Python, only one function definition (i.e., the last time that the function was defined) sticks. The flexibility comes from appropriate use of the different parameter types.

TIP If you're not familiar with it, you'll probably want to look into the "splat" operator, described here, in the Python tutorial: docs.python.org/3/tutorial/controlflow.html#arbitrary-argument-lists

1.3.1 Solution

```
def mysum(*numbers):
    output = 0
    for number in numbers:
        output += number
    return output
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

You can work through this code in the Python Tutor at https://goo.gl/mnpaFe .

1.3.2 Discussion

The above function is a simple example of how we can use Python's "splat" operator (aka `*`) to allow a function to receive any number of arguments. Because we have prefaced the name `numbers` with `\*`, we are telling Python that this parameter should receive all of the arguments, and that `numbers` will always be a tuple.

Even if no arguments are passed to our function, `numbers` will still be a tuple. It'll be an empty tuple, but a tuple nonetheless.

"splat" is especially useful when you want to receive an unknown number of arguments. Typically, you'll expect that all of the arguments will be of the same type, although Python does not enforce such a rule. In my experience, you will then take the tuple (`numbers`, in this case), and then iterate over each element with either a `for` loop or a list comprehension.

> **NOTE** If you are retrieving elements from `*args` with numeric indexes, then you're probably doing something wrong. Use individual, named parameters if you want to pick them off one at a time.

Because we expect all of the arguments to be numeric, we set our `output` local variable to 0 at the start of the function, and then add each of the individual numbers to it in a `for` loop.

Once we have this function, we can invoke it whenever we want, on any list, set, or tuple of numbers.

While you might not use `sum` (or re-implement it) very often, `*args` is an extremely common way for a function to accept an unknown number of arguments.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

TIP What if we have a list of numbers, such as `[1,2,3]`, and wish to use `mysum` with it? We cannot simply invoke `mysum([1,2,3])`; this will result in the `numbers` argument being a tuple whose first and only element is the list `[1,2,3]`, which looks like this: `([1,2,3],)`

Python will iterate over our one-element tuple, trying to add `0` to `[1,2,3]`. This will result in a `TypeError` exception, with Python complaining that it cannot add an integer to a list.

The solution in such a case is to preface the argument with `*`, when we invoke the function. If we call `mysum(*[1,2,3])`, our list becomes three separate arguments, which will then allow the function to be called in the usual way.

This is generally true when invoking functions: If you have an iterable object, and want to pass its elements to a function, just preface it with `\*` in the function call.

1.3.3 Beyond the exercise

It's extremely common to iterate over the elements of a list or tuple, performing an operation on each and then (for example) summing them. For example:

1. The builtin version of `sum` takes an optional second argument, which is used as the starting point for the summing. (That's why it takes a list of numbers as its first argument, unlike our `mysum` implementation.) So `sum([1,2,3], 4)` returns 10, because 1+2+3 is 6, and would be added to the starting value of 4. Re-implement your `mysum` function such that it works in this way. If a second argument is not provided, then it should default to 0. Note that while you can write a function in Python 3 that defines a parameter after `\*args`, I'd suggest avoiding it, just taking two arguments—a list and an optional starting point.
2. Write a function that takes a list of numbers. It should return the average (i.e., arithmetic mean) of those numbers.
3. Write a function that takes a list of words (strings). It should return a tuple containing three integers, representing the length of the shortest word, the length of the longest word, and the average word length.
4. Write a function that takes a list of Python objects. Sum the objects that either are integers or can be turned into integers, ignoring the others.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

1.4 Run timing

Python is often used by system administrators, to perform a variety of tasks, including producing reports from user inputs and files. It's not unusual to report how often a particular error message has occurred, or which IP addresses have accessed our server most recently, or which usernames are most likely to have incorrect passwords. Learning how to accumulate information over time and produce some basic reports (including average times) is thus useful and important. Moreover, knowing how to work with floating-point values, and the differences between them and integers, is important.

For this exercise, then, we'll assume that you run 10 km each day as part of your exercise regime. You want to know how long, on average, that run takes.

Write a program that asks how long it took to run 10 km today. The program continues to ask how long (in minutes) it took for additional runs, until the user enters `q`. At that point, the program exits---but only after calculating and displaying the average time that the 10 km run took.

For example, here is what the program would look like if the user enters three data points:

```
Enter 10 km run time: 15
Enter 10 km run time: 20
Enter 10 km run time: 10
Enter 10 km run time: q

Average of 15.0, over 3 runs
```

Note that the numeric inputs and outputs should all be floating-point values.

This exercise is meant to help you practice converting inputs into appropriate types, and also tracking information over time. You'll probably be tracking data that's more sophisticated than running times and distances, but the idea of accumulating data over time is common in programs, and it's important to see how to do this in Python.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

1.4.1 Solution

```
number_of_runs = 0
total_time = 0

while True:  ❶
    one_run = input("Enter 10 km run time, or 'q' to exit: ")

    if one_run == 'q':
        break

    else:
        number_of_runs += 1
        total_time += float(one_run)

average_time = total_time / number_of_runs

print(f"Average of {average_time}, over {number_of_runs} runs")
```

❶ Look, it's an infinite loop! It might seem weird to have `while True`, and it's a very bad idea to have such a loop without any `break` statement to exit when a condition is reached. But as a general way of getting an unknown number of inputs from the users, I think it's totally fine.

You can work through this code in the Python Tutor at https://goo.gl/PnQbgq .

1.4.2 Discussion

In the previous exercise, we saw that `input` is a function that returns a string, based on input from the user. In this case, however, there are two types of input that the user might provide: He or she might enter a number, but also might enter the empty string.

Because empty strings, as well as numeric 0, are considered to be "False" within an `if` statement, it's common for Python programs to use an expression as shown in the solution:

```
if not one_run:
    break
```

It's unusual, and would be a bit weird, to say:

```
if len(one_run) == 0:
    break
```

While the above works, it's not considered good Python style according to generally accepted conventions. These conventions can make your code more "Pythonic," and thus more readable by other developers.

Using `not` in front of a variable that might be empty, and thus provide us with a `False` value in this context, is much more common.

In a real-world Python application, if you're taking input from the user and calling `float` (docs.python.org/3/library/functions.html#float), you should probably wrap it within `try` (

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

docs.python.org/3/reference/compound_stmts.html#the-try-statement), in case the user gives us an illegal value:

```
try:
    n = float(input("Enter a number: "))
    print(f"n = {n}")
except ValueError as e:
    print("Hey!  That's not a valid number!")
```

Also remember that floats are not completely accurate. They are good enough for measuring the time it takes to run, but are a bad idea for any sensitive measurement, such as a scientific or financial calculation.

If you didn't know this already, then I suggest you go to your local interactive Python interpreter, and ask it for the value of `0.1 + 0.2`. You might be surprised by these results. (You can also go to http://0.30000000000000004.com/ and see how this works in other programming languages.)

One common solution for this problem is to use integers. For example, banks rarely use floating-point numbers; instead, they use integers representing (for example) hundredths of a cent, and then use integer math. If your application doesn't require extreme accuracy, then you probably don't need to worry about this too much, and can just use floats.

TIP

If you want to print a floating-point number in Python, then you might well want to use an f-string. Why? Because in this way, you can specify the number of digits that will be printed out. For example:

```
>>> s = 0.1 + 0.7
>>> print(s)
0.7999999999999999
```

That's probably not what you want. However, by putting `s` inside of an f-string, you can limit the output:

```
>>> s = 0.1 + 0.7
>>> print(f'{s:.2f}')
0.80
```

Here, I've told the f-string that I want to take the value of `s` and then display it as a floating-point number (`f`) with a maximum of two digits after the decimal point. See the reference table at the start of this chapter for the full documentation on f-strings and the formatting codes that can be used for different data types.

1.4.3 Beyond the exercise

Floats are both necessary and dangerous in the programming world; necessary because many things can only be represented with fractional numbers, but dangerous because floats aren't exact. You should thus think about when and where you use them:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

1. Write a function that takes a float and two integers (`before` and `after`). The function should return a float consisting of `before` digits before the decimal point and `after` digits after. Thus if we call the function with `1234.5678`, `2` and `3`, the return value should be `34.567`.
2. Explore the `Decimal` class (docs.python.org/3.7/library/decimal.html), which has an alternative floating-point representation that's as accurate as any decimal number can be. Write a function that takes two strings from the user, turns them into `Decimal` instances, and then prints the floating-point answer. In other words, make it possible for the user to enter `0.1` and `0.2`, and for us to get `0.3` back.

1.5 Hexadecimal output

Loops are everywhere in Python, and the fact that the built-in data structures are all iterable makes it easy to work through them, one element at a time. However, we typically iterate over an object forward, from its first element to the last one. Moreover, Python doesn't automatically provide us with the indexes of the elements.

In this exercise, we'll see how a bit of creativity, along with the builtin `reversed` and `enumerate` functions, can help us to get around these issues.

Hexadecimal numbers are fairly common in the world of computers. Actually, that's not entirely true: Some programmers use them all of the time. Other programmers, typically using high-level languages and doing things such as Web development, barely ever remember how to use them.

Now, the fact is that I barely use hexadecimal numbers in my day-to-day work. And even if I were to need them, I could use Python's built-in `hex` function and `0x` prefix. The former takes an integer and returns a hex string; the latter allows me to enter a number using hexadecimal notation, which can be more convenient. Thus, `0x50` is 80, and `hex(80)` will return the string `0x50`.

For this exercise, you need to write a program that takes a hex number and returns the decimal equivalent. That is, if the user enters `50`, then we will assume that it is a hex number (equal to `0x50`), and will print the value 80 on the screen. And no, you shouldn't convert the number all at once using the `int` function, although it is permissible to use `int` one digit at a time.

This exercise isn't meant to test your math skills; not only can you get the hex equivalent of integers with the `hex` function, but most people don't even need that in their day-to-day lives. However, this does touch upon the conversion (in various ways) across types that we can do in Python, thanks to the fact that sequences (e.g., strings) are iterable. Consider also the builtin functions that we can use to solve this problem even more easily than if we had to write things from scratch.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

TIP Python's exponentiation operator is `*\*`. So the result of `2**3` is the integer `8`.

1.5.1 Solution

```
d = 0
h = input("Enter a hex number to convert to decimal: ")
for power, digit in enumerate(reversed(h)):  ❶
    d += int(digit, 16) * (16 ** power)  ❷
print(d)
```

❶ `reversed` returns a new iterable, which returns another iterable's elements in reverse order. By invoking `enumerate` on the output from `reversed`, we get each element of `h`, one at a time, along with its index, starting with 0.

❷ Python's `**` operator is used for exponentiation.

You can work through this code in the Python Tutor at https://goo.gl/MJth5n .

1.5.2 Discussion

A key aspect of Python strings is that they are sequences of characters, over which we can iterate in a `for` (docs.python.org/3/reference/compound_stmts.html#the-for-statement) loop.

However, `for` loops in Python, unlike their C counterparts, don't give us (or even use) the characters' indexes. Rather, they iterate over the characters themselves.

If we want the numeric index of each character, we can use the builtin `enumerate` (docs.python.org/3/library/functions.html#enumerate) function. This function returns a two-element tuple with each iteration; using Python's multiple-assignment ("unpacking") syntax, we can capture each of these values, and stick them into our `power` and `digit` variables.

Here's an example of how we can use `enumerate` to print the first four letters of the alphabet, along with the letters' indexes in the string:

```
for index, one_letter in enumerate('abcd'):
    print(f"{index}: {one_letter}")
```

NOTE Why does Python have `enumerate` at all? Because in many other languages, such as C, `for` loops iterate over sequences of numbers, which are used to retrieve elements from a sequence. But in Python, our `for` loops retrieve the items directly, without needing any explicit index variable. `enumerate` thus produces the indexes based on the elements—precisely the opposite of how things work in other languges.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

You also see the use of `reversed` (docs.python.org/3/library/functions.html#reversed) here, such that we start with the final digit and work our way up to the first digit. `reversed` is a built-in function that returns a new string whose value is the the reverse of the old one. We could get the same result using slice syntax, `h[::-1]`, but I find that many people are confused by this syntax. Also, the slice returns a new string, whereas `reversed` returns an iterator, which consumes less memory.

We need to convert each digit of our decimal number, which was entered as a string, into an integer. We do that with the built-in `int` (docs.python.org/3/library/functions.html#int) function, which we can think of as creating a new instance of the `int` class or type. We also see that `int` takes two arguments. The first is mandatory, and is the string we want to turn into an integer. The second is optional, and contains the number base. Since we're converting from hexadecimal (i.e., base 16), we pass `16` as the second argument.

1.5.3 Beyond the exercise

Every Python developer should have a good understanding of the iteration protocol, which is used by `for` loops and in many functions. Combining `for` loops with other objects, such as `enumerate` and slices, can help to make your code shorter and more maintainable.

1. Re-implement the above program such that it doesn't use the `int` function at all, but rather uses the builtin `ord` and `chr` functions to identify the character. This implementation should be more robust, ignoring characters that aren't legal for the entered number base.
2. Write a program that asks the user for their name, and then produces a "name triangle": The first letter of their name, then the first two letters, then the first three, and so forth, until the entire name is written on the final line.

1.6 Summary

It's hard to imagine a Python program that doesn't use numbers. Whether as numeric indexes (into a string, list, or tuple), counting the number of times an IP address appears in a logfile, or calculating interest rates on bank loans, you'll be using numbers all of the time.

Remember that Python is strongly typed, meaning that integers and strings (for example) are different types. You can turn strings into integers with `int`, and integers into strings with `str`. And you can turn either of these types into a float with `float`.

In this chapter, we saw a few ways in which we can work with numbers of different types. You're unlikely to write programs that only use numbers in this way, but feeling confident about how they work and fit into the larger Python ecosystem is important.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

2 Strings

Strings in Python are the way in which we work with text. Words, sentences, paragraphs, and even entire files are read into and manipulated via strings. Because so much of our work revolves around text, it's no surprise that strings are one of the most common data types.

There are two important things to remember about Python strings: (1) They are immutable, and (2) in Python 3, they contains Unicode characters, encoded in UTF-8. (See the sidebars on each of these subjects, below.)

There's no such thing as a "character" type in Python; we can talk about a "one-character string," but that just means a string whose length is 1.

Python's strings are interesting and useful not only because they allow us to work with text, but also because they're a Python sequence. This means that we can iterate over them (character by character), retrieve their elements via numeric indexes, and search in them with the `in` operator.

This chapter contains exercises designed to help you work with strings in a variety of ways. The more familiar you are with Python's string-manipulation techniques, the easier it will be to work with text.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

2.1 Useful references

Table 2.1 Reference table

Name	Description	Example	Link
in	operator for searching in a sequence	'a' in 'abcd'	\#typesseq
slice	retrieve a subset of elements from a sequence	'abcdefg'[1:7:2] # returns 'bdf'	#slice
str.split	break strings apart, returning a list	'abc def ghi'.split() # returns ['abc', 'def', 'ghi']	#str.split
str.join	combine strings to create a new one	'*'.join(['abc', 'def', 'ghi']) # returns 'abc*def*ghi'	#str.join
list.append	add an element to a list	mylist.append('hello')	#typesseq-mutable
sorted	return a sorted list, based on an input sequence	sorted([10, 30, 20]) # returns [10, 20, 30]	#sorted
iterating over files	open a file, and iterate over its lines, one at a time	for one_line in open(filename):	#open

2.2 Pig Latin

Pig Latin (wikihow.com/Speak-Pig-Latin) is a common children's "secret" language in English-speaking countries. It's normally secret among children who forget that their parents were once children themselves.) The rules for translating words from English into Pig Latin are quite simple:

- If the word begins with a vowel (a, e, i, o, or u), then add `way` to the end of the word. So `air` becomes `airway` and `eat` becomes `eatway`.
- If the word begins with any other letter, then we take the first letter, put it on the end of the word, and then add `ay`. Thus, `python` becomes `ythonpay` and `computer` becomes `omputercay`.

(And yes, I recognize that the rules can be made more sophisticated. Let's keep it simple for the purposes of this exercise.)

For this exercise, write a Python program that asks the user to enter to enter an English word. Your program should then print the word, translated into Pig Latin. You may assume the the word contains no capital letters or punctuation.

This exercise isn't meant to help you translate documents into Pig Latin for your job. (If it is, then I really have to question your career choices.) However, it demonstrates some of the powerful techniques that you should know when working with sequences, including searches, iteration, and slices. It's hard to imagine a Python program that doesn't include any of these techniques.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

2.2.1 Solution

```
word = input("Enter a word: ")

if word[0] in 'aeiou':
    print(f'{word}way')
else:
    print(f'{word[1:]}{word[0]}ay')
```

You can work through this code in the Python Tutor at https://goo.gl/nh9wcT .

2.2.2 Discussion

This has long been one of my favorite exercises to give students in my introductory programming classes. It was inspired by Brian Harvey, whose excellent series Computer Science, Logo Style (http://www.cs.berkeley.edu/~bh/v1-toc2.html), has long been one of my favorites for beginning programmers.

The first thing to consider for this solution is how we'll check to make sure that `word[0]`, the first letter in `word`, is a vowel. I've often seen people start to use a loop, as in:

```
starts_with_vowel = False
for vowel in 'aeiou':
    if word[0] == vowel:
        starts_with_vowel = True
        break
```

Even if the above code will work, it's already starting to look a bit clumsy and convoluted.

TIP

`for` and `while` loops in Python support an `else` clause, just as `if` statements do. The `else` attached to a loop means: Only execute this code if we *didn't* encounter a `break`. We could thus rewrite the above code as follows:

```
for vowel in 'aeiou':
    if word[0] == vowel:
        starts_with_vowel = True
        break
else:
    starts_with_vowel = False
```

Note that in this code, the `else` is attached to the `for` loop, and **not** to the `if` statement inside of the `for` loop.

I'm often torn about using `else` clauses. On the one hand, I find them to be an elegant way to say, "I want to do something if we didn't accomplish something in the loop." On the other hand, I know that for many beginning Python programmers, `else` blocks seem quite weird in loops. Many people are convinced that such a block is misplaced, and was really meant to be aligned with the `for`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

Another solution that I commonly see is this:

```
if (word[0] ==  'a' or word[0] == 'e' or
       word[0] == 'i' or word[0] == 'o' or word[0] == 'u'):
    break
```

As I like to say to my students, "unfortunately, this code works." Why do I dislike this code so much? Not only is it longer than necessary, but it's highly repetitive. The "don't repeat yourself" rule of programming, often referred to as DRY, should always be at the back of your mind when writing code.

Moreover, Python programs tend to be short: If you find yourself repeating yourself and writing an unusually long expression or condition, you have likely missed a more "Pythonic" way of doing things.

We can take advantage of the fact that Python sees a string as a sequence, and use the built-in `in` operator to search for `word[0]` in a string containing the vowels:

```
if word[0] in 'aeiou':
```

The above has the combined advantage of being readable, short, accurate, and fairly efficient. True, the time needed to search through a string---or any other Python sequence---rises along with the length of the sequence. But such "linear time," sometimes expressed as `O(n)`, is often good enough, especially when the strings through which we'll be searching are fairly short.

TIP The `in` operator works on all sequences (strings, lists, and tuples) and many other Python collections. It effectively runs a `for` loop on the elements, which means that using `in` on a dictionary will work, but will search through the keys, ignoring the values.

Once we have determined whether the word begins with a vowel, we can apply the appropriate Pig Latin rule.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

SIDEBAR **Slices**

All of Python's sequences---strings, lists, and tuples---support "slicing." The idea is that if I say

```
s = 'abcdefgh'
print(s[2:6])   ❶
```

❶ Returns `cdef`

I'll get all of the characters from `s`, starting at index 2 and until (but not including) index 6, meaning the string `cdef`. A slice can also indicate the step size:

```
s = 'abcdefgh'
print(s[2:6:2])   ❶
```

❶ Returns `ce`

The above code will print the string `ce`, since we start at index 2 (`c`), then move forward two indexes to `e`, and then we reach the end.

Slices are Python's way of retrieving a subset of elements from a sequence. You can even omit the starting and/or ending indexes, to indicate that you want to start from the sequence's first element or end at its last element. For example, we can get every other character from our string with:

```
s = 'abcdefgh'
print(s[::2])   ❶
```

❶ Returns `aceg`

2.2.3 Beyond the exercise

It's hard to exaggerate just how often you'll need to work with strings in Python. Moreover, Python is often used in text analysis and manipulation. Here are some ways that you can extend the exercise to push yourself further:

1. Handle capitalized words: If a word is capitalized (i.e., the first letter is capitalized, but the rest of the word isn't), then the Pig Latin translation should be similarly capitalized.
2. Handle punctuation: If a word ends with punctuation, then that should be shifted to the end of the translated word.
3. Consider an alternative version of Pig Latin, in which we don't check to see if the first letter is a vowel, but rather we check to see if the word contains two different vowels. Thus, "wine" would have "way" added to the end, but "wind" would be translated into

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

"indway". How would you check for two different vowels in the word? (Hint: Sets can come in handy here.)

SIDEBAR **Immutable?**

One of the most important concepts in Python is the distinction between mutable and immutable data structures.

The basic idea is simple: If a data structure is immutable, then it cannot be changed. Ever.

For example, if I define a string, and then try to change it:

```
s = `abcd`
s[0] = '!'    ❶
```

❶ You'll get an exception when running this code

The above code won't work; we'll get an exception, with Python telling us that we're not allowed to modify a string.

Many data structures in Python are immutable, including such basics as integers and boolean values. But strings are where people get tripped up most often, partly because we use strings so often, and partly because many other languages have mutable strings.

Why would Python do such a thing? There are a number of reasons, chief among which is that it makes the implementation more efficient. But it also has to do with the fact that strings are the most common type used as dictionary keys. If strings were mutable, then they wouldn't be allowed as dict keys---or we would have to allow for mutable keys in dicts, which would create a whole host of other issues.

Because immutable data cannot be changed, we can make a number of assumptions about it: If we pass an immutable type to a function, then it won't be modified by the function. If we share immutable data across threads, then we don't have to worry about locking it, because it cannot be changed. And if we invoke a method on an immutable type, then we get a new object back---because we cannot modify immutable data.

Learning to work with immutable strings takes some time, but the trade-offs are generally worthwhile. If you find yourself needing a mutable string type, then you might want to look at StringIO (docs.python.org/3/library/io.html#io.StringIO), which provides file-like access to a mutable, in-memory type.

Many newcomers to Python often think that "immutable" is just another word for "constant," but it isn't. Constants, which many programming languages offer, permanently connect a name with a value. In Python, there is no such thing as a constant; you can always reassign a name to point to a new value. But you cannot modify a string or a tuple, no matter how hard you try.

For example:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

```
s = 'abcd'
s[0] = '!'   ❶
t = s   ❷
s = '!bcd'   ❸
```

❶ Not allowed, since strings are immutable

❷ The variables `s` and `t` now point to the same string.

❸ `s` now points to the new string, but `t` continues to point to the old string, unchanged.

2.3 Pig Latin sentence

Now that you have successfully written a translator for a single English word, let's make things more difficult: Translate a series of English words into Pig Latin. (To make things easier, we won't actually ask for a real sentence. More specifically, there will be no capital letters or punctuation.) So, if someone were to enter

```
this is a test translation
```

the output would be

```
histay isway away estay ranslationtay
```

The user's input can contain any number of words. We want the output to be print on a single line, rather than with each word on a separate line.

This exercise might seem, at least superficially, like the previous one. But here, the emphasis is not on the Pig Latin translation. Rather, it's on the ways in which we typically use loops in Python, and how loops go together with breaking strings apart and putting them back together again. It's also common to want to take a sequence of strings and print them out on a single line. There are a few ways to do this, and I want you to consider the advantages and disadvantages of each.

2.3.1 Solution

```
sentence = input("Enter a sentence: ")

output = [ ]
for word in sentence.split():
    if word[0] in 'aeiou':
        output.append(f"{word}way")
    else:
        output.append(f"{word[1:]}{word[0]}ay")

print(' '.join(output))
```

You can work through this code in the Python Tutor at https://goo.gl/Rzhc1J .

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

2.3.2 Discussion

The core of the above program is nearly identical to the one in the previous section, in which we translated a single word into Pig Latin. Once again, we're getting a text string as input from the user. The difference is that in this case, rather than treating the string as a single word, we're treating it as a sentence---meaning that we need to separate it into individual words.

We can do that with `str.split` (docs.python.org/3/library/stdtypes.html#str.split). `str.split` can take an argument, which determines which string should be used as the separator between fields.

It's often the case that you want to use any and all whitespace characters, regardless of how many there are, to split the fields. In such a case, don't pass an argument at all; Python will then treat any number of spaces, tabs, and newlines as a single separation character. The difference can be significant:

```
s = 'abc  def  ghi'    ❶
s.split(' ')           ❷
s.split()              ❸
```

❶ 2 spaces separating

❷ returns: ['abc', '', 'def', '', 'ghi']

❸ returns: ['abc', 'def', 'ghi']

NOTE If you don't pass any arguments to `str.split`, it's effectively the same as passing `None`.

You can pass any string to `str.split`, not just a single-character string. Meaning that if you want to split on `::`, you can do that.

However, you cannot split on more than one thing, saying that both `,` and `::` are field separators. In order to do that, you'll need to use regular expressions, and the `re.split` function in the Python standard library, described here: docs.python.org/3/library/re.html#re.split

Thus, we can take the user's input and break it into words---again, assuming that there are no punctuation characters---and then translate each individual word into Pig Latin.

While the one-word version of our program could simply print its output right away, this one needs to store the accumulated output and then print it all at once.

It's certainly possible to use a string for that, and to invoke `+=` on the string with each iteration. But as a general rule, it's not a good idea to build strings in that way. Rather, you should add elements to a list using `list.append` (docs.python.org/3/library/stdtypes.html#str.split), and then invoke `str.join` to turn the list's elements into a long string.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

That's because strings are immutable, and `+=` on a string forces Python to create a new string. If we're adding to a string many times, then each time will trigger the creation of a new object whose contents are larger than the previous iteration. By contrast, lists are mutable, and adding to them with `list.append` is relatively inexpensive, in both memory and computation.

Another way to ensure that the results are all printed on one line is to take advantage of the `print` function's `end` parameter. By default, `end` is set to `\n`, the newline character, meaning that after printing its value to the screen, `print` will then insert a newline. You can change that to any string by passing a value in your call to `print`, as in:

```
sentence = input("Enter a sentence: ")

for word in sentence.split():
    if word[0] in 'aeiou':
        print(f"{word}way", end=' ')   ❶
    else:
        print(f"{word[1:]}{word[0]}ay", end=' ')
```

❶ After printing, add a space character, rather than the default `\n`.

If you expect to have a great deal of elements in the `output` list, then this might be a more efficient solution than the one I presented above.

2.3.3 Beyond the exercise

Splitting, joining, and manipulating strings are common actions in Python. Here are some additional activities you can try to push yourself even further:

1. Take a text file, creating (and printing) a nonsensical sentence from the nth word on each of the first 10 lines, where n is the line number.
2. Write a function that transposes a list of strings, in which each string contains multiple words separated by whitespace. So if you were to pass the list `['abc def ghi', 'jkl mno pqr', 'stu vwx yz']` to the function, it would return `['abc jkl stu', 'def mno vwx', 'ghi pqr yz']`.
3. Read through an Apache logfile. If there is a 404 error---you can just search for `' 404 '` , if you want---then display the IP address, which should be the first element.

2.4 Ubbi Dubbi

When they hear that Python's strings are immutable, many people wonder how the language can be used for text processing. After all, if you cannot modify strings, then how can you do any serious work with them?

Moreover, there are times when a simple `for` loop, as we did with the Pig Latin examples, won't work. If we're modifying each word only a single time, then that's fine — but if we're potentially modifying it several times, then we have to make sure that each modification doesn't

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

affect future modifications.

This exercise is meant to help you practice thinking in this way. Here, you'll implement a translator from English into another secret children's language, Ubbi Dubbi (https://en.wikipedia.org/wiki/Ubbi_dubbi). (This was popularized on the wonderful American children's program "Zoom," which was on television when I was growing up.) The rules of Ubbi Dubbi are even simpler than those of Pig Latin, although programming a translator is more complex and requires a bit more thinking.

In Ubbi Dubbi, every vowel (a, e, i, o, or u) is prefaced with `ub`. Thus `milk` becomes `mubilk` (`m-ub-ilk`) and `program` becomes `prubogrubam` (`prub-ogrub-am`). In theory, you only put an `ub` before every vowel **sound**, rather than before each vowel. Given that this is a book about Python and not linguistics, I hope that you'll forgive this slight difference in definition.

Ubbi Dubbi is enormously fun to speak, and it's somewhat magical if and when you can begin to understand someone else speaking it. Even if you don't understand it, Ubbi Dubbi sounds extremely funny. See some YouTube videos (https://www.youtube.com/results?search_query=ubbi+dubbi) on the subject, if you need convincing.

For this exercise, you'll translate a word from English into Ubbi Dubbi. So if the user enters `octopus`, you'll output `uboctubopubus`. And if the user enters `elephant`, you'll output `ubelubephubant`.

As with the original Pig Latin translator, you can ignore capital letters, punctuation, and corner cases, such as multiple vowels combining to create a new sound. When you do have two vowels next to one another, preface each of them with `ub`. Thus, `soap` will become `suboubap`, despite the fact that `oa` combines to a single vowel sound.

Much like the "Pig Latin sentence" exercise, this brings to the forefront the various ways in which we often need to scan through strings for particular patterns, translate from one Python data structure or pattern to another, and how iterations can play a central role in doing so.

2.4.1 Solution

```
word = input("Enter a word: ")

output = []
for letter in word:
    if letter in 'aeiou':
        output.append(f'ub{letter}')   ❶
    else:
        output.append(letter)

print(''.join(output))
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

❶ Why append to a list, and not to a string? To avoid allocating too much memory. For short strings, it's not a big deal. But for long loops and large strings, it's a bad idea.

You can work through this code in the Python Tutor at https://goo.gl/v4YE3d .

2.4.2 Discussion

The task here is to ask the user for a word, and then to translate that word into Ubbi Dubbi. This is a slightly different task than we had with Pig Latin, because we need to operate on a letter-by-letter basis. We cannot simply analyze the word and produce output based on the entire word. Moreover, we have to avoid getting ourselves into an infinite loop, in which we try to add `ub` before the `u` in `ub`.

The solution is to iterate over each character in `word`, adding it to a list, `output`. If the current character is a vowel, then we add `ub` before the letter. Otherwise, we just add the letter. At the end of the program, we join and then print the letters together. This time, we don't join the letters together with a space character (`' '`), but rather with an empty string (`''`). This means that the resulting string will consist of the letters joined together with nothing between them---or, as we often call such collections, a "word."

2.4.3 Beyond the exercise

It's not uncommon to want to replace one value with another in strings. Python has a few different ways to do this: You can use `str.replace`, or `str.translate`, two string methods that translate strings and sets of characters, respectively. But sometimes, there's no choice but to iterate over a string, look for the pattern we want, and then append the modified version to a list that we grow over time.

1. Handle capitalized words: If a word is capitalized (i.e., the first letter is capitalized, but the rest of the word isn't), then the Ubbi Dubbi translation should be similarly capitalized.
2. In academia, it's common to remove the authors' names from a paper submitted for peer review. Consider how you would, given a text file stored in a Python string and a list of authors' names (as strings), replace their names with _ characters.
3. In URLs, we often replace special and non-printable characters with a `%` followed by the hexadecimal value of the character's ASCII value. For example, if a URL is to include a space character (ASCII 32, aka 0x20), we replace it with `%20`. Given a string, URL-encode any character that isn't a letter or number. For the purposes of this exercise, we'll assume that all characters are indeed in ASCII (i.e., one byte long), and not multibyte UTF-8 characters.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

2.5 Sorting a string

If strings are immutable, then does this mean we're stuck with them, forever, precisely as they are?

Sort of — the strings themselves cannot be changed, but we can create new strings based on them, using a combination of builtin functions and string methods. Knowing how to work around strings' immutability, and piece together functionality that effectively changes strings, even though they're immutable, is a useful skill to have.

In this exercise, you'll explore this idea by writing a function, `strsort`, that takes a single string as its input, and returns a string. The returned string should contain the same characters as the input, except that its characters should be sorted in order, from smallest Unicode value to highest Unicode value.

For example, the result of invoking `strsort('cba')` will be the string `abc`.

2.5.1 Solution

```
def strsort(s):
    return ''.join(sorted(s))
```

You can work through this code in the Python Tutor at https://goo.gl/eCeXx5 .

2.5.2 Discussion

The above implementation of `strsort` takes advantage of the fact that Python strings are sequences. Normally, we think of this as relevant in a `for` loop, in that we can iterate over the characters in a string. However, we don't need to restrict ourselves to such situations.

For example, we can use the built-in `sorted` (docs.python.org/3/library/functions.html#sorted) function, which takes an iterable---which means not only a sequence, but anything over which we can iterate, such as a set of files---and returns its elements in sorted order. Invoking `sorted` in our string will thus do the job, in that it will sort the characters in Unicode order. However, it returns a list, rather than a string.

In order to turn our list into a string, we use the `str.join` method. We use an empty string (`''`) as the glue we will use to join the elements, thus returning a new string whose characters are the same as the input string, but in sorted order.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

SIDEBAR **Unicode**

What is Unicode? The idea is a simple one, but the implementation can be extremely difficult, and is confusing to many developers.

The idea behind Unicode is that we should be able to use computers to represent any character used in any language from any time. This is a very important goal, in that it means we won't have problems creating documents in which we want to show Russian, Chinese, and English on the same page. Previous to Unicode, mixing character sets from a number of languages was difficult or impossible.

Unicode assigns each character a unique number. But those numbers can (as you imagine) get very big. Thus, we have to take the Unicode character number (known as as "code point") and translate it into a format that can be stored and transmitted as bytes. Python and many other languages use what's known as UTF-8, which is a "variable-length encoding." Meaning that different characters might require different numbers of bytes. Characters that exist in ASCII are encoded into UTF-8 with the same number as from ASCII, in one byte. French, Spanish, Hebrew, Arabic, Greek, and Russian all use two bytes for their non-ASCII characters. And Chinese, as well as your childrens' emojis, are three bytes or more.

How much does this affect us? Both a lot and a little. On the one hand, it's convenient to be able to work with different languages so easily. On the other hand, it's easy to forget that there's a difference between bytes and characters, and that you sometimes (e.g., when working with files on disk) need to translate from bytes to characters, or vice versa.

2.5.3 Beyond the exercise

This exercise is designed to give you additional reminders that strings are sequences, and can thus be put wherever other sequences (lists and tuples) can be used. We don't often think in terms of "sorting a string," but there's no difference between running `sorted` on a string, a list, or a tuple. The elements (in the case of a string, the characters) are returned in sorted order.

However, `sorted` returns a list, and we wanted to get a string. We thus needed to turn the resulting list back into a string---something that `str.join` is designed to do. `str.split` and `str.join` are two methods with which you should become intimately familiar, because they're so useful and help in so many cases.

Consider a few other variations of, and extensions to, this exercise, which also use `str.split` and `str.join`, as well as `sorted`:

1. Given the string "Tom Dick Harry", break it into individual words, and then sort those words alphabetically. Then print them with commas (`,`) between the names.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

2. Which is the last word, alphabetically, in a text file?
3. Which is the longest word in a text file?

2.6 Summary

Python programmers are constantly dealing with text. Whether it's because we're reading from files, displaying things on the screen, or just using dictionaries, strings are a data type with which we're likely familiar from other languages.

At the same time, strings in Python are unusual, in that they're also sequences---and thus, thinking in Python requires that you consider their sequence-like qualities. This means searching (using `in`), sorting (using `sorted`), and using slices. It also means thinking about how you can turn strings into lists (using `str.split`) and turning sequences back into strings (using `str.join`). While these might seem like simple tasks, they crop up on a regular basis in production Python code. The fact that these data structures and methods are written in C, and have been around for many years, means that they are also highly efficient---and not worth re-inventing.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

3 Lists and tuples

Consider a program that has to work with documents. Or keep track of users. Or log the IP addresses that have accessed our server. Or store the names and birthdates of children in a school. In all of these cases, we're storing many pieces of information. We'll want to display, search through, extend, and modify this information.

These are such common tasks that every programming language supports a number of "collections," data structures that designed for handling such cases. Lists and tuples are Python's built-in collections. Technically, they differ in that lists are mutable, whereas tuples are immutable. But in practice, lists are meant to be used for sequences of the same type, whereas tuples are meant for sequences of different types.

For example, a list of documents, users, or IP addresses would be best stored in lists — because we have many objects of the same type. A record containing one person's name and birthdate would be best stored in a tuple, because the name and birthdate are of different types. A bunch of such name-birthday tuples, however, could be stored in a list, because it contains a sequence of tuples — and the tuples are all of the same type.

Because they're mutable, lists support many more methods and operators. After all, there's not much you can do with a tuple other than pass it, retrieve its elements, and make some queries about its contents. Lists, by contrast, can be extended, contracted, and modified, as well as searched, sorted, and replaced. So you can't add a person's shoe size to the name-birthday tuple you've created for them. But you can add a bunch of additional name-birthday tuples to the list you've created, as well as remove elements from that list if they're no longer a close friend.

Learning to distinguish between when you would use lists vs. when you would use tuples can take some time. If the distinction isn't totally clear to you just yet, it's not your fault!

Lists and tuples are both Python "sequences," which means that we can run `for` loops on them, search using the `in` operator, and retrieve from them using both individual indexes and with

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

slices. The third sequence type in Python is the string, which we looked at in the previous chapter. I find it useful to think of the sequences in this way:

Table 3.1 Sequence comparison

Type	Mutable?	Contains	Syntax	Retrieval
str	no	one-element strings	s = 'abc'	s[0] # returns 'a'
list	yes	any Python type	mylist = [10, 20, 30, 40, 50]	mylist[2] # returns 30
tuple	no	any Python type	t = (100, 200, 300, 400, 500)	t[3] # returns 400

In this chapter, we'll practice working with lists and tuples. We'll see how to create them, modify them (in the case of lists), and use them to keep track of our data. We'll also use "list comprehensions," a syntax that is confusing to many, but which allows us to take one Python iterable and create a new list based on it. We'll talk about comprehensions quite a bit in this chapter and the following ones; if you're not familiar or comfortable with them, look at the references provided in the reference table, below.

Table 3.2 Reference table

Name	Description	Example	Link
lists	ordered, mutable sequence	[10, 20, 30]	#list
tuples	ordered, immutable sequence	(3, 'clubs')	#tuple
list comprehensions	return a list based on an iterable	[str(x) for x in [10, 20, 30]] # returns ['10', '20', '30]	think-like-an-accountant/
range	return an iterable sequence of integers	numbers = range(10, 50, 3) # every 3rd integer, from 10 until (and not including) 50	#range
operator.itemgetter	returns a function that operates like square brackets	final = operator.itemgetter(-1) # final('abcd') == 'd'	#operator.itemgetter
collections.Counter	subclass of dictionary useful for counting items in an iterable	c = Counter('abcdab') # roughly the same as {'a':2, 'b':2, 'c':1, 'd':1}	#collections.Counter
max	builtin function, returning the largest element of an iterable	max([10, 20, 30]) # returns 30	#max
str.format	string method, returning a new string based on a template (similar to f-strings)	'x = {0}, y = {1}'.format(100, [10, 20, 30]) # returns 'x = 100, y = [10, 20, 30]'	#str.format

3.1 First-last

For many programmers coming from a background in Java or C#, the dynamic nature of Python is quite strange. How can a programming language fail to police which type can be assigned to which variable? Fans of dynamic languages, such as Python, respond that this allows us to write generic functions that handle many different types.

Indeed, we need to do so: In many languages, you can define a function multiple times, so long as each definition has different parameters. In Python, you can only define a function a single time---or more precisely, defining a function a second time will overwrite the first definition---and thus, we need to use other techniques to work with different types of inputs.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

Such functions demonstrate the elegance and power of dynamic typing. You can write a single function that works with many types, rather than many nearly identical functions, each for a specific type.

The fact that sequences — strings, lists, and tuples — all implement many of the same APIs is not an accident. Python encourages us to write such generic functions. For example, all three sequence types can be searched with `in`, can return individual elements with an index, and can return multiple elements with a slice.

We'll practice these ideas with this exercise: Write a function, `firstlast`, that takes a sequence (string, list, or tuple), and returns the first and last elements of that sequence, in a two-element sequence of the same type. So `firstlast('abc')` will return the string `ac`, while `firstlast([1,2,3,4])` will return the list `[1,4]`.

3.1.1 Solution

```
def firstlast(sequence):
    return sequence[:1] + sequence[-1:]   ❶
```

❶ In both cases, we're using slices, not indexes

You can work through this code in the Python Tutor at https://goo.gl/1H78ys .

3.1.2 Discussion

This exercise is as tricky as it is short. However, I believe that it helps to demonstrate the difference between retrieving an individual element from a sequence, and a slice from that sequence. It also shows the power of a dynamic language; we don't need to define several different versions of `firstlast`, each handling a different type. Rather, we can define a single function that handles not only the built-in sequences, but also any new types we might define which can handle indexes and slices.

One of the first thing that Python programmers learn is that they can retrieve an element from a sequence---a string, list, or tuple---using square brackets and a numeric index. So you can retrieve the first element of `s` with `s[0]`, and the final element of `s` with `s[-1]`.

But that's not all: You can also retrieve a "slice," or a subset of the elements of the sequence, by using a colon inside of the square brackets. The easiest and most obvious way to do this is something like `s[2:5]`, which means that you want a string whose content is from `s`, starting at index 2, up to and not including index 5. (Remember that in a slice, the final number is always "up to and not including.")

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

NOTE In this book, I use many diagrams from the Python Tutor (http://PythonTutor.com/), an amazing online resource for teaching and learning Python. (I often use it in my in-person classes.) You can enter nearly any Python code into the site, and then walk through its execution, piece by piece. Each solution also has a link pointing to the code in the Python Tutor, so that you can run it without typing it into the site.

In the Python tutor, global variables (including functions and classes) are shown in the "global frame." Remember that if you define a variable outside of a function, you have created a global variable. Any variables you create inside of a function are local variables—and are shown, in the Python tutor, inside of their own shaded boxes. Simple data structures, such as integers and strings, are shown alongside the variables pointing to them, while lists, tuples, and dictionaries are shown in graphical format.

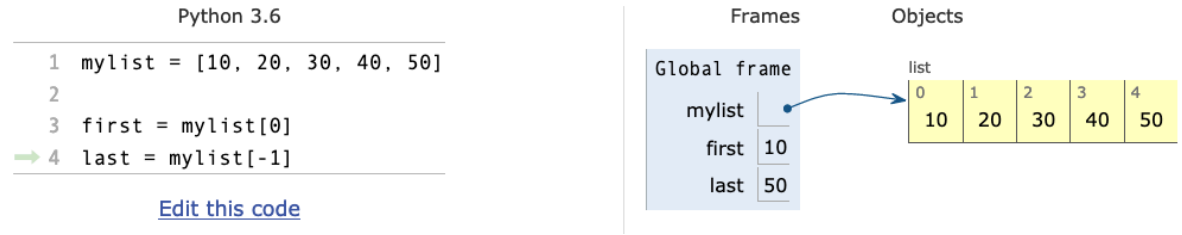


Figure 3.1 Individual elements (from the Python Tutor)

When you retrieve a single element from a sequence, you can get any type at all. String indexes return one-character strings, but lists and tuples can contain anything. By contrast, when you use a slice, you're guaranteed to get the same type back---so the slice of a tuple is a tuple, regardless of the size of the slice or the elements it contains. And the slice of a list will return a list. In these diagrams from the Python Tutor, notice that the data structures are different, and thus the results of retrieving from each will be different:

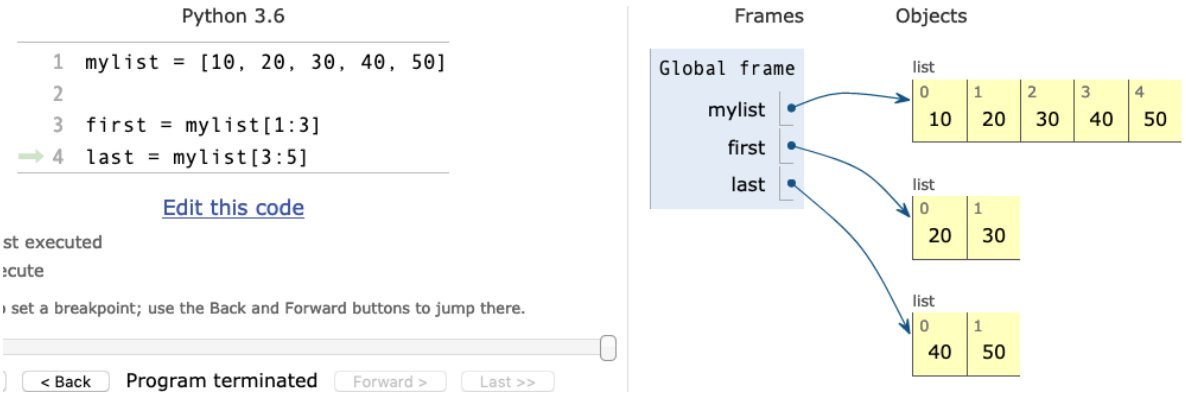


Figure 3.2 Retrieving slices from a list (from the Python Tutor)

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

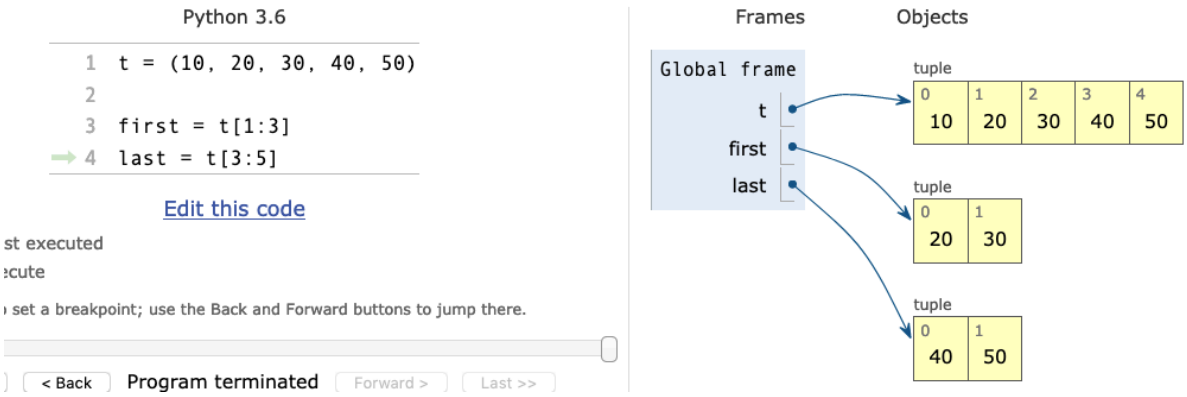


Figure 3.3 Retrieving slices from a tuple (from the Python Tutor)

NOTE

When retrieving a single index, you cannot go beyond the bounds:

```
s = 'abcd'
s[5]   # raises an IndexError exception
```

However, when retrieving with a slice, Python is more forgiving, ignoring any index beyond the data structure's boundaries:

```
s = 'abcd'
s[3:100]   # returns 'd'
```

In the above diagrams, there is no index 5. And yet, Python forgives us, showing the data all the way to the end. We just as easily could have omitted the final number.

Given that we're trying to retrieve the first and last elements of `sequence`, and then join them together, it might seem reasonable to grab them both (via indexes) and then add them together:

```
# not a real solution!
def firstlast(sequence):
    return sequence[0] + sequence[-1]
```

But this is what really happens:

```
def firstlast(sequence):   ❶
    return sequence[0] + sequence[-1]

t1 = ('a', 'b', 'c')
output1 = firstlast(t1)
print(output1)   ❷

t2 = (1,2,3,4)
output2 = firstlast(t2)
print(output2)   ❸
```

❶ Not a real solution!

❷ prints the string 'ac', not ('a', 'c')

❸ prints the integer 5, not (1, 4)

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

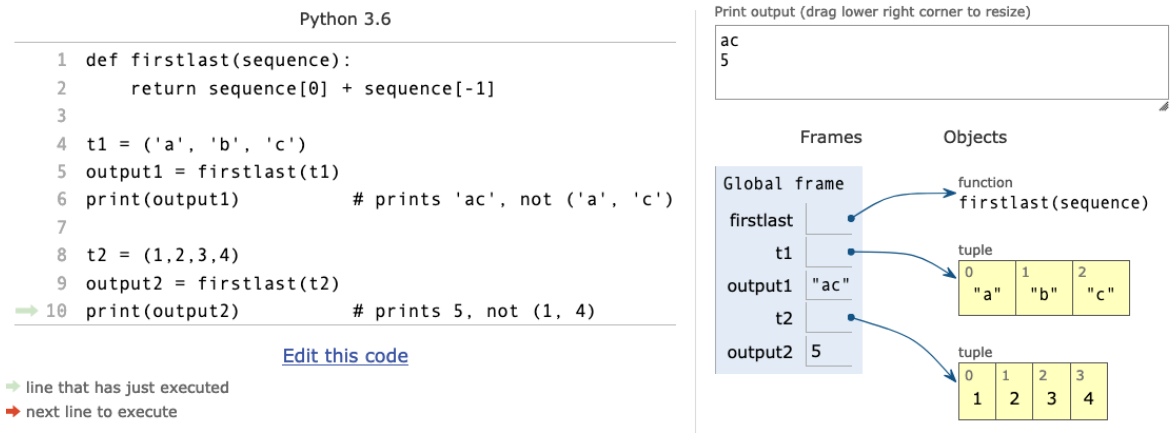


Figure 3.4 Naive, incorrect adding of slices (from the Python Tutor)

We can't simply use + on the individual elements of our tuples. As we see in the above diagram, if the elements are strings or integers, then using + on those two elements will give us the wrong answer. We want to be adding tuples — or whatever type `sequence` is.

The easiest way to do that is to use a slice, using `s[:1]` to get the first element, and we use `s[-1:]` to get the final element. Notice that we have to say `s[-1:]`, so that the sequence will start with the element at -1, and end at the end of the sequence itself.

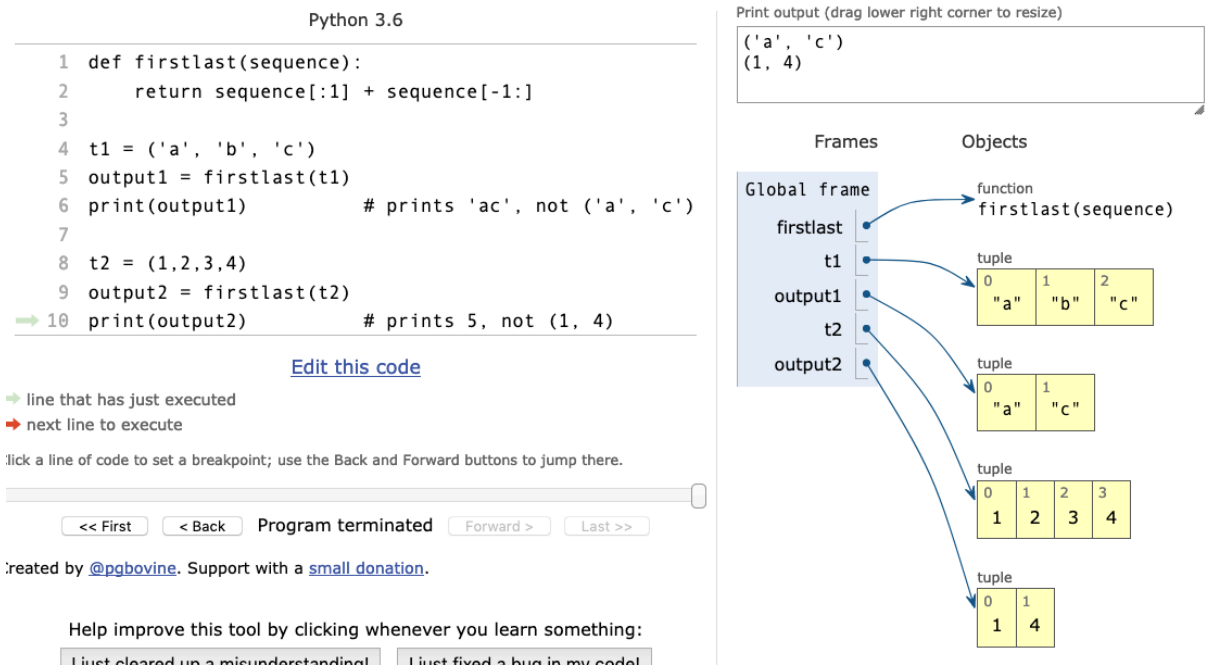


Figure 3.5 Working solution (from the Python Tutor)

The bottom line is that when you retrieve a slice from an object x, you get back a new object of the same type as x. But if you retrieve an individual element from x, you'll get whatever was stored in x — which might be the same type as x, but you cannot be sure.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

3.1.3 Beyond the exercise

One of these techniques involves taking advantage of Python's dynamic typing. That is: While data is strongly typed, variables don't have any types. This means that we can write a function that expects to take any indexable type (i.e., one that can get either a single index or a slice as an argument), and then return something appropriate. This is a common technique in Python, one with which you should become familiar and comfortable.

For example:

1. Don't write one function that squares integers, and another that squares floats. Write one function that handles all numbers.
2. Don't write one function that finds that largest element of a string, another for a list, and another for a tuple. Write just one function that works on all of them.
3. Don't write one function that works on files, and another that works on the `io.StringIO` file simulator using in testing. Write one function that works on both, finding the largest word in the file.

Slices are a great way to get at just part of a piece of data. Whether it's a substring or part of a list, slices allow you to grab just part of any sequence. I'm often asked by students in my courses how they can iterate over just the final n elements of a list; when I remind them that they can do this with the slice `mylist[-3:]` and a `for` loop, they're somewhat surprised and embarrassed that they didn't think of this first; they were sure that it must be more difficult than that.

Here are some ideas for other tasks you can try, using indexes and slices:

1. Write a function that takes a list or tuple of numbers. Return a two-element list, containing (respectively) the sum of the even-indexed numbers and the sum of the odd-indexed numbers. So calling the function as `even_odd_sums([10, 20, 30, 40, 50, 60])`, you'll get back `[90, 120]`.
2. Write a function that takes a list or tuple of numbers. Return the result of alternately adding and subtracting numbers from each other. So calling the function as `plus_minus([10, 20, 30, 40, 50, 60])`, you'll get back the result of `10+20-30+40-50+60`, or `50`.
3. Write a function that emulates the builtin `zip` function, taking any number of iterables and returning a list of tuples. Each tuple will contain one element from each of the iterables passed to the function. Thus, if I call `myzip([10, 20,30], 'abc')`, the result will be `[(10, 'a'), (20, 'b'), (30, 'c')]`. You can return a list (not an iterator), and can assume that all of the iterables are of the same length.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

3.2 All A's

Lists are mutable, which means that we can modify them. We normally imagine that this means adding to them (using the `list.append`, `list.extend`, and `list.insert` methods) or removing from them (using the `list.remove` and `list.pop` methods). But you can also replace elements of a list based on their positions — using either indexes or slices. This can happen when you're working with (for example) a list of numbers or files, and need to swap some of the values.

In this exercise, we'll take this idea to an extreme: Start with a list, `mylist`, which contains any number of elements, of any type. You want to modify this list, such that it contains six elements, each of which is the letter `a`. For example:

```
mylist = list(range(10))
alias = mylist

# DO SOMETHING HERE

print(mylist)  # prints ['a','a','a','a','a','a']
print(alias)   # prints ['a','a','a','a','a','a']
```

3.2.1 Solution

```
mylist = list(range(10))   ❶
mylist[:] = ['a'] * 6
print(mylist)
```

❶ In Python 3, "range" returns an object. So if you want a list, you must invoke `list` on it.

You can work through this code in the Python Tutor at https://goo.gl/UXapVG .

3.2.2 Discussion

This specific, change-everything operation is not something you're likely going to want to do. However, once you understand how lists change, and how this affects other variables that might point to the same list, you'll understand many other aspects — or dangers — of mutable data.

This exercise combines several of my favorite aspects of Python sequences:

1. The slicing syntax, which allows us to retrieve part of a sequence,
2. The fact that if a slice is on the left side of an assignment, you can replace its items with those of another sequence, regardless of the number of elements in either sequence, and
3. The fact that you can multiply a list to get a larger list.

Let's take each piece in turn, and then see how they come together:

First, we create the 10-element list with `list(range(10))`. We don't really care about the contents of the list, since we will be replacing it anyway.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

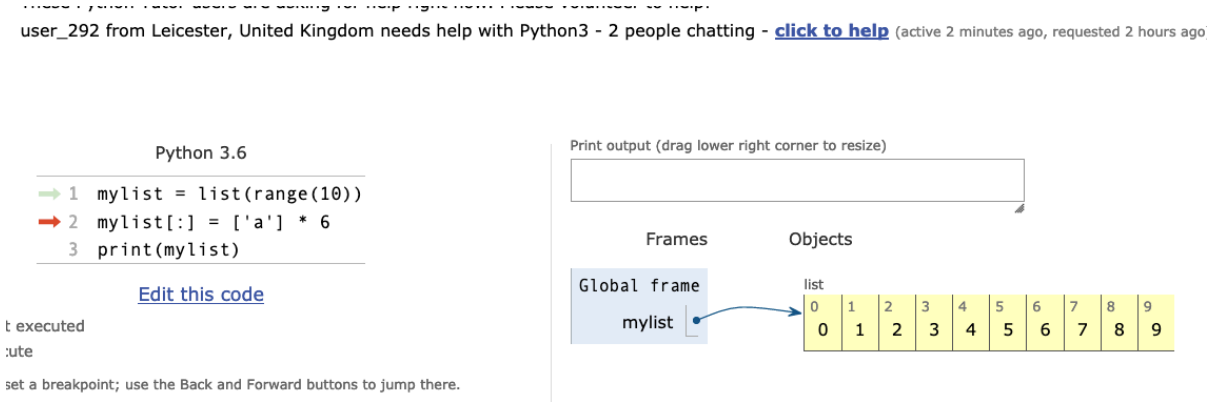


Figure 3.6 Creating the list (from the Python Tutor)

We then create a six-element list of `['a']` by simply multiplying `['a'] * 6`. Python's `*` operator is syntactic sugar for the *`mul`* method, which means that so long as the data type on the left has implemented *`mul`*, we can multiply it. In the case of lists, this method simply returns the multiplied list a set number of times. So `[1] * 3` results in `[1, 1, 1]`, while `a * 3` results in `aaa`.

NOTE

All Python operators are actually methods. When you say:

```
x = 2 + 3
```

you might think that you're simply adding integers. But in fact, Python sees the + and translates it into the *add* method.

But wait, methods are connected to objects. On which object are we running *add*? On the left-hand operand. We can thus rewrite the above code as:

```
x = type(2).__add__(2, 3)
```

The fact that Python operators are actually methods in disguise might seem unimportant right now. But it makes it possible for us to reason about objects and types we haven't yet used. It also shows us how to make our own objects that fit into these Python conventions, working with the builtin operators.

Finally, we assign to a slice of `mylist`---but not just any slice. We assign to the slice of all elements of the list, replacing the original list's contents with the list of six a's we created.

There is a world of difference between the following two lines:

```
mylist[:] = ['a'] * 6
mylist = ['a'] * 6
```

In the first case, `mylist` continues to point to the same list. Any other variable that is pointing to

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

`mylist` will reflect this assignment, as well. This is because we are changing the object itself.

When you assign to a slice, the sequence on the right side of the assignment does not need to be the same length as the left side. The fact that we have a 10-element list, and that we're assigning a smaller list's elements to it, is totally fine. For example, here we'll replace elements of a list with a string:

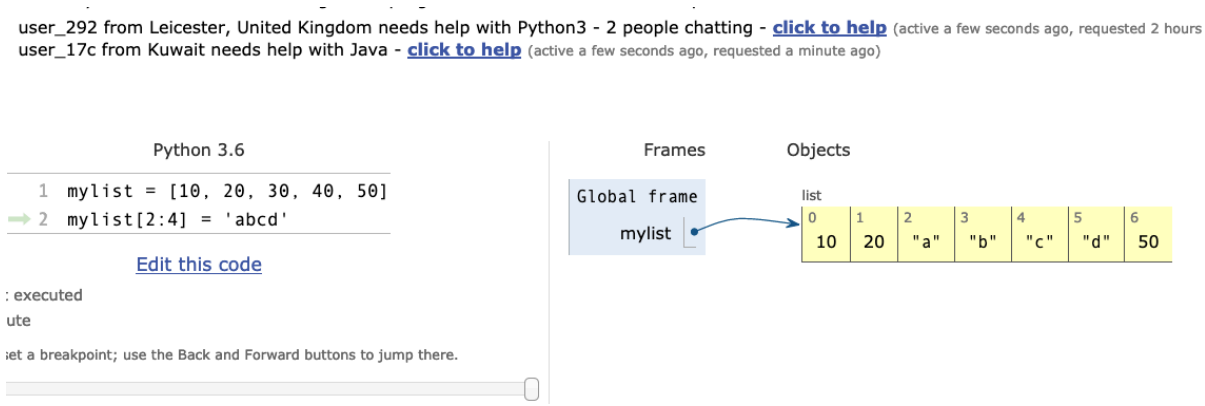


Figure 3.7 Replacing elements in a list with an iterable (from the Python Tutor)

Which elements of `mylist` do we want to replace? All of them. We thus put, on the left side of the assignment, `mylist[:]`. What do we want in its place, the six-element list we created with `['a'] * 6`. Here's how that looks:

Figure 3.8 Replacing elements in a list with an iterable (from the Python Tutor)

By contrast, when we assign to `mylist` (rather than `mylist[:]`), we are not changing the object. The original version of the list continues, for at least a short time, to exist within Python's memory. At some point, it will be garbage collected, assuming that `mylist` was the only reference to that data structure. But if some other variable was pointing to the same value as `mylist`, the = sign used in assignment broke that connection between them.

This is the double-edged sword of mutable data structures: They are quite convenient to work with, and make it possible to update the data based on current needs. However, we have to remember when we are modifying the data structure, and when we are changing its contents.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

3.2.3 Beyond the exercise

Modifying lists is something that we do all of the time — and can even happen while you're executing a `for` loop on one, such as when you're executing a depth-first search through a directory's tree structure. Here are some additional things you can try that go beyond the exercise, which will increase your fluency with modifying lists:

1. Swap the first element of a list with the last, the second with the second-to-last, and so forth—so that the list `[10, 20, 30, 40, 50]` becomes `[50, 40, 30, 20, 10]`.
2. Given a list of integers `mylist`, produce a new list, in which even numbers in `mylist` are multiplied by 5 and odd numbers in `mylist` are multiplied by 3.
3. You can modify a list while iterating over it—something that is often a bad idea, but which can help in certain situations. Start with a three-element list of numbers, [10, 20, 30]. Print each number, and then append 5x that number to the end of the list. So when you get to 10, add 50 to the end of the list. And when you get to 20, add 100 to the end of the list. Stop printing (and exit the loop) when 5x the current number would be greater than 1,000.

SIDEBAR **Are lists arrays?**

Newcomers to Python often look for the "array" type. While Python does have two different arrays (one in the `array` module, in the standard library, and another one in the popular `numpy` package on PyPI), lists are typically the go-to datatype for anyone needing an array or array-like structure.

And you know what? For the most part, we don't really need or use arrays in Python. They don't align with the language's dynamic nature. Instead, we normally use lists and tuples.

Now, lists aren't arrays: Arrays have a fixed length, as well as a type. And while you could potentially argue that Python's lists handle only one type, namely anything that inherits from the built-in `object` class, it's definitely not true that lists have a fixed length. The above exercise demonstrates that pretty clearly, but doesn't use the `list.append` or `list.remove` methods.

Behind the scenes, Python lists are implemented as arrays of pointers to Python objects. But if arrays are of fixed size, how can Python use them to implement lists? The answer is that Python allocates some extra space in its list array, such that we can add a few items to it. But at a certain point, if you add enough items to our list, these "spare" locations will be used up, thus forcing Python to allocate a new array, and move all of the pointers to that location. This is done for us automatically and behind the scenes, but it shows that adding items to a list isn't completely free of computational overhead. You can see this in action using the `sys.getsizeof` method, which shows the number of bytes needed to store the list (or any other data structure):

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

```
>>> import sys
>>> for i in range(25):
...     print(f"len(mylist) = {len(mylist)}, getsizeof(mylist) =
          {sys.getsizeof(mylist)}")
...     mylist.append(i)
```

Running the above code gives us the following output:

```
len(mylist) = 0, getsizeof(mylist) = 64
len(mylist) = 1, getsizeof(mylist) = 96
len(mylist) = 2, getsizeof(mylist) = 96
len(mylist) = 3, getsizeof(mylist) = 96
len(mylist) = 4, getsizeof(mylist) = 96
len(mylist) = 5, getsizeof(mylist) = 128
len(mylist) = 6, getsizeof(mylist) = 128
len(mylist) = 7, getsizeof(mylist) = 128
len(mylist) = 8, getsizeof(mylist) = 128
len(mylist) = 9, getsizeof(mylist) = 192
len(mylist) = 10, getsizeof(mylist) = 192
len(mylist) = 11, getsizeof(mylist) = 192
len(mylist) = 12, getsizeof(mylist) = 192
len(mylist) = 13, getsizeof(mylist) = 192
len(mylist) = 14, getsizeof(mylist) = 192
len(mylist) = 15, getsizeof(mylist) = 192
len(mylist) = 16, getsizeof(mylist) = 192
len(mylist) = 17, getsizeof(mylist) = 264
len(mylist) = 18, getsizeof(mylist) = 264
len(mylist) = 19, getsizeof(mylist) = 264
len(mylist) = 20, getsizeof(mylist) = 264
len(mylist) = 21, getsizeof(mylist) = 264
len(mylist) = 22, getsizeof(mylist) = 264
len(mylist) = 23, getsizeof(mylist) = 264
len(mylist) = 24, getsizeof(mylist) = 264
```

As you can see, then, the list grows as necessary, but always has some spare room, allowing it to avoid growing if you're just adding a handful of elements.

How much do you need to care about this in your day-to-day Python development? As with all matters of memory allocation and Python language implementation, I think of this as useful background and knowledge, either for when you're in a real bind when optimizing, or just for a better sense of and appreciation for how Python does things.

But if you are worried about the size of your data structures, or the way in which Python is allocating memory behind the scenes, on a very regular basis, then I'd argue that you're probably worrying about the wrong things---or you're using the wrong language for the job at hand. Python as a fantastic language for many things, but super efficient usage of memory isn't one of its claims to fame.

3.3 Summing anything

We've seen how we can write a function that takes a number of different types. We've also seen how we can write a function that returns different types, using the argument that the function received.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

In this exercise, we'll see how we can have even more flexibility experimenting with types. What happens if you're running methods not on the argument itself, but on elements within the argument? For example, what if we want to sum the elements of a list — regardless of whether those elements are integers, floats, strings, or even lists?

This challenge asks you to redefine `mysum`, such that it can take any number of arguments. The arguments must all be of the same type and know how to respond to the `+` operator. (Thus, the function should work with numbers, strings, lists, and tuples, but not with sets and dictionaries.)

The result should be a new, longer sequence of the type provided by the parameters. Thus, the result of `mysum('abc', 'def')` will be the string `abcdef`, and the result of `mysum([1,2,3], [4,5,6])` will be the six-element list `[1,2,3,4,5,6]`. Of course, it should also still return the integer `6` if we invoke `mysum(1,2,3)`.

Working through this exercise will give you a chance to think about sequences, types, and how we can most easily create return values of different types, from the same function.

3.3.1 Solution

```
def mysum(*items):
    if not items:   ❶
        return items
    output = items[0]
    for item in items[1:]:
        output += item    ❷
    return output
```

❶ In Python, everything is considered `True` in an `if` except for `None`, `False`, 0, and empty collections. So if the tuple `items` is empty, we'll just return an empty tuple.

❷ We're assuming that the elements of `items` can be added together.

You can work through this code in the Python Tutor at https://goo.gl/mm1Tgz .

3.3.2 Discussion

The above version of `mysum` is more complex than the one we saw previously. It still accepts any number of arguments, which are put into the `items` tuple thanks to the "splat" (`*`) operator.

> **TIP** While we traditionally call the "takes any number of arguments" parameter `*args`, in fact you can use any name you want. The important part is the `*`, not the name of the parameter; it still works the same way, and is always a tuple.

The first thing that we do is check to see if we received any arguments. If not, then we return `items`, an empty tuple. This is necessary, because the rest of the function requires that we know the type of the passed arguments, and that we have an element at index 0. Without any

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

arguments, neither will work.

Notice that we don't check for an empty tuple by comparing it with `()`, or checking that its length is 0. Rather, we can say `if not items`, which asks for the boolean value of our tuple. Because an empty Python sequence is `False` in a boolean context, we get `False` if `args` is empty, and `True` otherwise.

In the next line, I grab the first element of `items`, and assign it to `output`. If it's a number, then `output` will be a number. But if it's a string, then `output` will be a string, and so on. This gives us the base value to which we'll add (using +) each of the subsequent values in `items`.

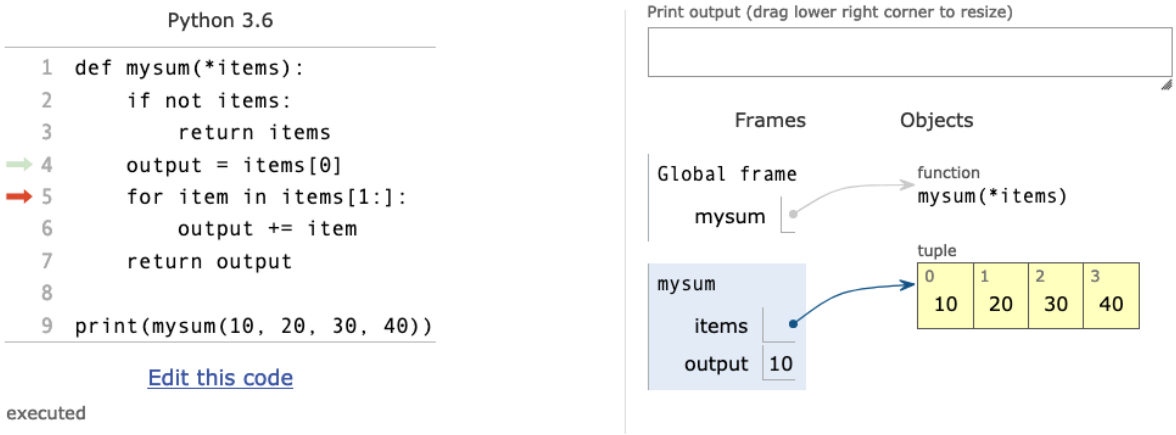


Figure 3.9 After assigning the first element to output (from the Python Tutor)

Once that is in place, we do what the original version of `mysum` did---but instead of iterating over all of `items`, we can now iterate over `items[1:]`, meaning all of the elements except for the first one. Here, we again see the value of Python's slices, and how we can use them to solve problems.

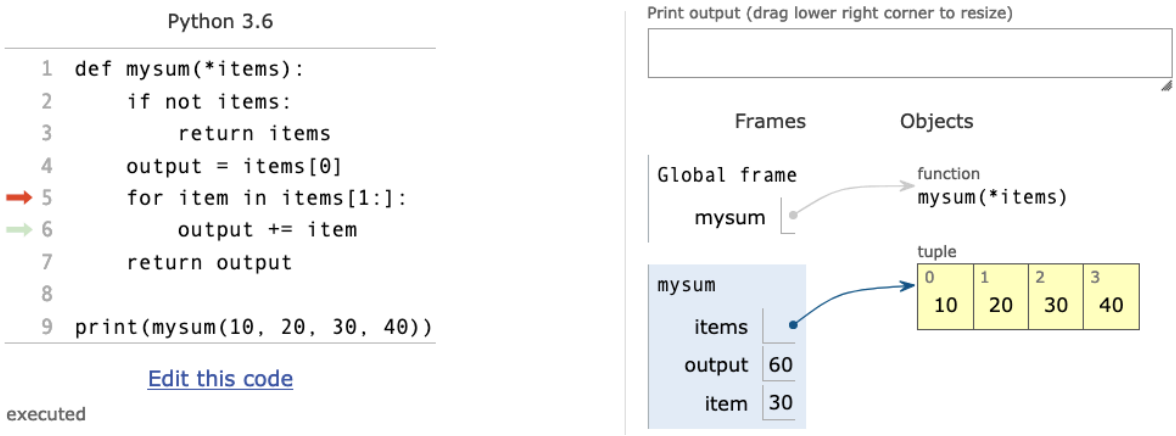


Figure 3.10 After adding additional elements to output (from the Python Tutor)

You can think of this implementation of `mysum` as the same as our original version: Except that

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

instead of adding each element to 0, we're adding each one to `items[0]`.

But wait, what if the person passed us only a single argument, and thus `args` doesn't contain anything at index 1? Fortunately, slices are forgiving, and allow us to specify indexes beyond the sequence's boundaries. In such a case, we'll just get an empty sequence, over which the `for` loop will run zero times. Meaning, we'll just get the value of `items[0]` returned to us as `output`.

3.3.3 Beyond the exercise

This exercise demonstrates some of the ways in which we can take advantage of Python's dynamic typing to create a function that works with many different types of inputs, and even produces different types of outputs. Here are a few other problems you can try to solve, which have similar goals:

1. Write a function, `mysum_bigger_than`, which works the same as `mysum`, except that it takes a first argument, preceding `\*args`. That argument indicates the threshold for including an argument in the sum. Thus, calling `mysum_bigger_than(10, 5, 20, 30, 6)` would return `50`—because `5` and `6` aren't greater than `10`. This function should similarly work with any types, and assumes that all of the arguments are of the same type. Note that `>` and `<` work on many different types in Python, not just on numbers; with strings, lists, and tuples, it refers to their sort order.
2. Write a function, `sum_numeric`, which takes any number of arguments. If the argument is or can be turned into an integer, then it should be added to the total. Those arguments which cannot be handled as integers should be ignored. The result is the sum of the numbers. Thus, `sum_numeric(10, 20, 'a', '30', 'bcd')` would return `60`. Notice that even if the string `30` is an element in the list, it's converted into an integer and added to the total.
3. Write a function that takes a list of dictionaries, and returns a single dictionary that combines all of the keys and values. If a key appears in more than one argument, then the value should be a list containing all of the values from the arguments.

3.4 Alphabetizing names

Let's assume that you have phone book data in a list of dictionaries, as follows:

```
people = [{'first':'Reuven', 'last':'Lerner', 'email':'reuven@lerner.co.il'},
 {'first':'Donald', 'last':'Trump', 'email':'president@whitehouse.gov'},
 {'first':'Vladimir', 'last':'Putin', 'email':'president@kremvax.ru'}
 ]
```

First of all, if these are the only people in your phone book, then you should rethink whether Python programming is truly the best use of your time and connections. Regardless, let's assume that you want to print information about all of these people, but in phone-book order---that is, sorted by last name and then by first name. Each line of the output should just look like this:

```
LastName, FirstName: email@example.com
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

3.4.1 Solution

```
import operator
for person in sorted(people, key=operator.itemgetter('last', 'first')):   ❶
    print(f"{person['last']}, {person['first']}: {person['email']}")
```

❶ The "key" parameter to "sorted" gets a function, whose result indicates how we'll sort.

You can work through this code in the Python Tutor at https://goo.gl/eK6EqN .

3.4.2 Discussion

While Python's data structures are useful by themselves, they become even more powerful and useful when combined. Lists of lists, lists of tuples, lists of dictionaries, and dictionaries of dictionaries are all quite common. Learning to work with these is an important part of being a fluent Python programmer. This exercise shows how you can not only store data in such structures, but also manipulate, sort, and retrieve from them.

There are two parts to the solution I propose. In the first part, we sort our data according to the criteria I proposed, namely last name and then first name. The second part of the solution addresses how we'll print output to the end user.

Let's take the second problem first: We have a list of dictionaries. This means that when we iterate over our list, `person` is assigned a dictionary in each iteration. The dictionary has three keys: `first`, `last`, and `email`. We will want to use each of these keys to display each phone-book entry.

We could thus say:

```
for person in people:
    print(f"{person['last']}, {person['first']}: {person['email']}")
```

So far, so good. But we still haven't covered the first problem, namely sorting the list of dictionaries by last name and then first name. Basically, we want to tell Python's sort facility that before it compares two dictionaries from our `people` list, it should turn the dictionary into a list, consisting of the person's last and first names. In other words, we want:

```
{'first':'Vladimir', 'last':'Putin', 'email':'president@kremvax.ru'}
```

to become

```
['Putin', 'Vladimir']
```

If we want to apply a function to each list element before the sorting comparison takes place, pass a function to the `key` parameter. Thus, we can sort elements of a list by saying:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

```
mylist = ['abcd', 'efg', 'hi', 'j']
mylist.sort(key=len)
```

After executing the above, `mylist` will now be sorted in increasing order of length, because the built-in `len` function (docs.python.org/3/library/functions.html#len) will be applied to each element before it is compared with others. In the case of our alphabetizing exercise, we could write a function that takes a dict and returns the sort of list that's necessary:

```
def person_dict_to_list(d):
    return [d['last'], d['first']]
```

We could then apply this function when sorting our list:

```
people.sort(key=person_dict_to_list)
```

Following that, we could then iterate over the now-sorted list, and display our people.

However, it feels wrong to me to sort `people` permanently, if it's just for the purposes of displaying its elements. Furthermore, I don't see the point in writing a special-purpose named function if I'm only going to use it once.

We can thus use two pieces of Python which come from the functional programming world---the built-in `sorted` function, which returns a new, sorted list based on its inputs and `lambda` (docs.python.org/3/reference/compound_stmts.html#lambda), which returns a new, anonymous function, allowing us to avoid the definition of a new function. Combining these, we get to this solution:

```
for person in sorted(people, key=lambda person: [person['last'], person['first']]):
    print("{person['last']}, {person['first']}: {person['email']}")
```

Many of the Python developers I meet are less than thrilled to use `lambda`. It works, but makes the code less readable and more confusing to many. (See the sidebar for more thoughts on `lambda`.) Fortunately, the `operator` module has the `itemgetter` function, which returns a function that does precisely what we want to do here, namely return a list of `person['last']` and `person['first']`. Using `itemgetter`, we can just say:

```
for person in sorted(people, key=operator.itemgetter('last', 'first')):
    print(f"{person['last']}, {person['first']}: {person['email']}")
```

3.4.3 Beyond the exercise

Learning to sort Python data structures, and particularly combinations of Python's built-in data structures, is an important part of working with Python. It's not enough to use the built-in `sorted` function, although that's a good part of it; understanding how sorting works, and how you can use the `key` parameter, is also essential. This exercise might have introduced this idea, but consider a few more sorting opportunities:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

1. Given a sequence of positive and negative numbers, sort them by absolute value.
2. Given a list of strings, sort them according to how many vowels they contain.
3. Given a list of lists, with each list containing zero or more numbers, sort by the sum of each inner list's numbers.

SIDEBAR **What is lambda?**

Many Python developers ask me just what `lambda` is, what it does, and where they might want to use it.

The answer is that `lambda` returns a function object, allowing us to create an anonymous function. And we can use it wherever we might use a regular function, without having to "waste" a variable name.

Consider the following code:

```
glue = '*'
s = 'abc'
print(glue.join(s))
```

The above code returns `a*b*c`, the string returned by calling `glue.join` on `s`. But why do I need to define either `glue` or `s`? Can't I just use strings without any variables? Of course you can, as we see here:

```
print('*'.join('abc'))
```

The above code produces the same result as we had before. The difference is that instead of using variables, we're using literal strings. These strings are created when we need them here, and go away after our code is run. You could say that they are "anonymous strings." Anonymous strings, also known as string literals, are perfectly normal and natural, and we use them all of the time.

Now consider that when we define a function using `def`, we're actually doing two things: We're both creating a function object and assigning that function object to a variable. We call that variable "a function," but it's no more a function than `x` is an integer after we say that `x=5`. Assignment in Python always means that a name is pointing to an object, and functions are objects just like anything else in Python.

For example, consider the following code:

```
mylist = [10, 20, 30]

def hello(name):
    return f"Hello, {name}"
```

If we execute the above code in the Python tutor, we can see that we have defined two variables: One (`mylist`) points to an object of type list. The second (`hello`) points to a function object:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

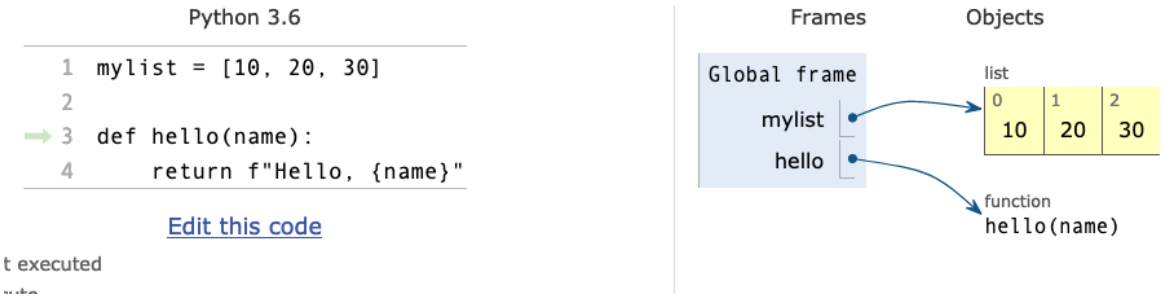


Figure 3.11 Both mylist and hello point to objects (from the Python Tutor)

SIDEBAR

Because functions are objects, they can be passed as arguments to other functions. This seems weird at first, but you quickly get used to the idea of passing around all objects, including functions.

For example, I'm going to define a function (`run_func_with_world`) that takes a function as an argument. It then invokes that function, passing it the string `world` as an argument:

```
def hello(name):
    return f"Hello, {name}"

def run_func_with_world(func):
    return func('world')

print(run_func_with_world(hello))
```

Notice that we're now passing `hello` as an argument to the function `run_func_with_world`. As far as Python is concerned, this is totally reasonable and normal.

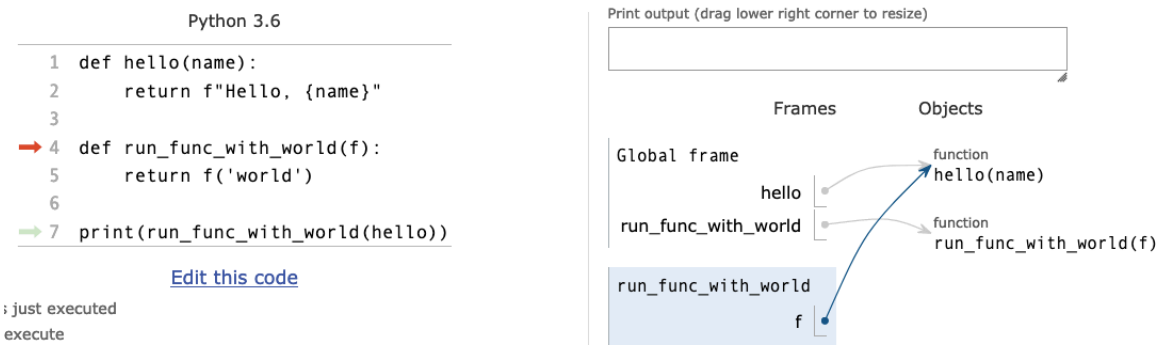


Figure 3.12 Calling hello from another function (from the Python Tutor)

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

SIDEBAR

There are many instances in which we'll want to write a function that takes another function as an argument. One such example is `sorted`.

What does this have to do with `lambda`? Well, we can always create a function using `def`---but then we find ourselves creating a new variable. And for what? So that we can use it a single time? Ignoring environmental concerns, you probably don't want to buy metal forks, knives, and spoons for a casual picnic; rather, you can just buy plasticware. In the same way, if I only need a function a single time, then why would I define it formally, and give it a name?

This is where `lambda` enters the picture: It lets us create an anonymous function, perfect for passing to other functions. It goes away, removed from memory as soon as it is no longer needed.

Meaning: If we think of `def` as both creating a function object and then defining a variable that points to that object, then we can think of `lambda` as doing just the first of these tasks. It creates a function object. The code that I wrote above, in which I called `run_func_with_world` and passed it `hello` as an argument, could be rewritten using `lambda` as follows:

```
def run_func_with_world(f):
    return f('world')

print(run_func_with_world(lambda name: f"Hello, {name}"))
```

Here, I've removed the definition of `hello`, but I've created an anonymous function that does the same thing, using `lambda`.

Figure 3.13 Calling an anonymous function from a function (from the Python Tutor)

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

SIDEBAR To create an anonymous function with `lambda`, use the reserved world `lambda` and then list any parameters before a colon. Then write the one-line expression that the `lambda` returns. And indeed, in a Python `lambda` you are restricted to a single expression—no assignment is allowed.

Nowadays, many Python developers prefer not to use `lambda`, partly because of its restricted syntax, and partly because more readable options, such as `itemgetter`, are available and do the same thing. I'm still a softie when it comes to `lambda`, and like to use it when I can---but I also realize that for many developers, it makes the code harder to read and maintain. You'll have to decide just how much `lambda` you want to have in your code.

3.5 Word with most repeated letters

Write a function, `most_repeating_word`, that takes a sequence of strings as input. The function should return the string that contains the greatest number of repeated letters. That is, if `words` is set to

```
words = ['this', 'is', 'a', 'test', 'program']
```

then your function could return either `program` or `test`, since both contain a letter twice---`r` in the case of `program`, and `t` in the case of `test`. It doesn't really matter which of these words is considered the "winner."

You will probably want to use `Counter`, from the `collections` module, which is perfect for counting the number of items in a sequence.

3.5.1 Solution

```
from collections import Counter
import operator

words = ['this', 'is', 'a', 'test', 'program']

def most_repeating_letter_count(word):  ❶
    return Counter(word).most_common(1)[0][1]  ❷

def most_repeating_word(words):
    word_counts = {word : most_repeating_letter_count(word)
                   for word in words}

    return max(word_counts.items(), key=operator.itemgetter(1))[0]  ❸

most_repeating_word(words)
```

❶ What letter appears the most times, and how many times does it appear?

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

❷ `Counter.most_common` returns a list of two-element tuples (value and count), in descending order.

❸ Just as you can pass `key` to `sorted`, you can also pass it to `max`, and use a different sort method.

You can work through this code in the Python Tutor at https://goo.gl/8a8UFg .

3.5.2 Discussion

This solution combines a few of my favorite Python techniques into a short piece of code:

1. `Counter`, a subclass of `dict` defined in the `collections` module, which makes it easy to count things
2. a dict comprehension, creating a dict based on a sequence
3. passing a function to the `key` parameter in `max`

In order for our solution to work, we'll need to find a way to determine how many times each letter appears in a word. The easiest way to do that is `Counter`. It's true that `Counter` inherits from `dict`, and thus can do anything that a `dict` can do. But we normally build an instance of `Counter` by initializing it on a sequence. For example:

```
>>> Counter('abcabcabbbc')
Counter({'a': 3, 'b': 5, 'c': 3})
```

We can thus feed `Counter` a word, and it'll tell us how many times each letter appears in that word. We could, of course, iterate over the resulting `Counter` object, and grab the letter that appears the most times. But why work so hard, when we can invoke `Counter.most_common`?

```
>>> Counter('abcabcabbbc').most_common()   ❶
[('b', 5), ('a', 3), ('c', 3)]
```

❶ Show how often each item appears in the string, from most common to least common, in a list of tuples

The result of invoking `Counter.most_common` is a list of tuples, with the names and values of the counter's values in descending order. So in the above example, we see that `b` appears 5 times in the input, `a` appears 3 times and `c` also appears 3 times. If we were to invoke `most_common` with an integer argument `n`, we would only see the `n` most common items:

```
>>> Counter('abcabcabbbc').most_common(1)   ❶
[('b', 5)]
```

❶ Only show the most common item, and its count

This is perfect for our purposes. Indeed, I think it would be useful to wrap this up into a function that'll return the number of times the most frequently appearing letter is in the word:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

```
def most_repeating_letter_count(word):
    return Counter(word).most_common(1)[0][1]   ❶
```

❶ The `(1)[0][1]` at the end looks a bit confusing. It means: We only want the most commonly appearing letter, returned in a one-element list of tuples. We then want the first element from that list, a tuple. We then want the count for that most common element, at index 1 in the tuple.

Remember that we don't care which letter is repeated. We just care how often the most frequently repeated letter is indeed repeated. And yes, I also dislike the multiple indexes at the end of this function call, which is part of the reason I want to wrap this up into a function, so that I don't have to see it as often. But we can call `most_common` with an argument of `1` to say that we're only interested in the highest-scoring letter, then that we're interested in the first (and only) element of that list, and then that we want the second element (i.e., the count) from the tuple.

Now that we've managed to score a word, we will need to do the same thing for all of the words in a sentence. This means iterating over the sentence, broken into individual words, and applying our function to it. This sounds like a perfect opportunity to use a list comprehension, which translates one sequence into another, producing a list.

However, if we do that, then we'll get the scores---not the words themselves. We will thus create a `dict` comprehension, creating a dictionary whose keys are our words and whose values are the results from invoking `most_repeating_letter_count` on each word.

```
def most_repeating_word(words):
    word_counts = {word : most_repeating_letter_count(word)
                   for word in words}
```

But even that's not quite enough; this produces a dictionary, but it doesn't return the word whose letters repeat the greatest number of times. In order to do that, we could sort the dictionary by value (rather than by key), using the built-in `sorted` function.

That would work, but we can achieve the same results by invoking the builtin `max` function, which returns the final (i.e., largest) element from `sorted`. In other words, we can write a bit less code by using `max`, passing it `words.items()`---meaning, the key-value pairs from our `word_counts` dict. By passing `max` the `key` parameter, giving it `operator.itemgetter(1)`, we can sort the key-value pairs by value. `max` will thus return a two-element tuple, with the winning word and its highest repeated-letter count. The final trick is to retrieve the item at index 0 from that tuple, the word itself.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

3.5.3 Beyond the exercise

Sorting, manipulating complex data structures, and passing functions to other functions are all rich topics deserving of your attention and practice. Here are a few things you can do to go beyond this exercise, and explore these ideas some more:

1. Instead of finding the word with the greatest number of repeated letters, find the word with the greatest number of repeated vowels.
2. Write a program to read `/etc/passwd` on a Unix computer. The first field contains the username, and the final field contains the user's "shell," the command interpreter. Display the shells in decreasing order of popularity, such that the most-popular shell is shown first, the second-most popular shell second, and so forth.
3. For an added challenge, after displaying each shell, also show the usernames (sorted alphabetically) who use each of those shells.

3.6 Printing tuple records

A common use for tuples is as records, similar to a "struct" in some other languages. And of course, displaying those records in a table is a standard thing for programs to do. In this exercise, we'll do a bit of both — reading from a list of tuples, and turning them into formatted output for the user.

For example, assume that we are in charge of an international summit in London. We know how many hours it will take each of several world leaders to arrive:

```
people = [('Donald', 'Trump', 7.85),
          ('Vladimir', 'Putin', 3.626),
          ('Jinping', 'Xi', 10.603)]
```

The planner for this summit needs to have a list of the world leaders who are coming, along with the time it will take for them to arrive. However, this travel planner doesn't need the degree of precision that the computer has provided; it's enough for us to have two digits after the decimal point.

For this exercise, write a Python program that takes the above `people` list, and produces a table that looks like the following:

```
Trump      Donald      7.85
Putin      Vladimir    3.63
Xi         Jinping    10.60
```

Notice that the last name is printed before the first name (taking into account that Chinese names are generally shown that way), followed by a decimal-aligned indication of how long it will take for each leader to arrive in London. Each name should be printed in a 10-character field, and the time should be printed in a 5-character field, with one space character of padding between each of the columns. Travel time should display only two digits after the decimal point, which means

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

that even though the input for Xi Jinping's flight is 10.603 hours, the value displayed should be `10.75`.

3.6.1 Solution

```
import operator
for person in sorted(people, key=operator.itemgetter(1, 0)):  ❶
        print("{1:10} {0:10} {2:5.2f}".format(*person))
```

❶ `operator.itemgetter` can be used with any data structure that takes square brackets. We can also pass it more than one argument, as seen here.

3.6.2 Discussion

Tuples are often used in the context of structured data and database records. In particular, you can expect to receive a tuple when you retrieve one or more records from a relational database. You will then need to retrieve the individual fields using numeric indexes.

This exercise had several parts: First of all, we needed to sort the people in alphabetical order according to last name and first name. I used the built-in `sorted` function to sort the tuples, using a similar algorithm to what we used with the list of dictionaries in an earlier exercise. The `for` loop thus iterated over each element of our sorted list, getting a tuple (which it called `person`) in each iteration.

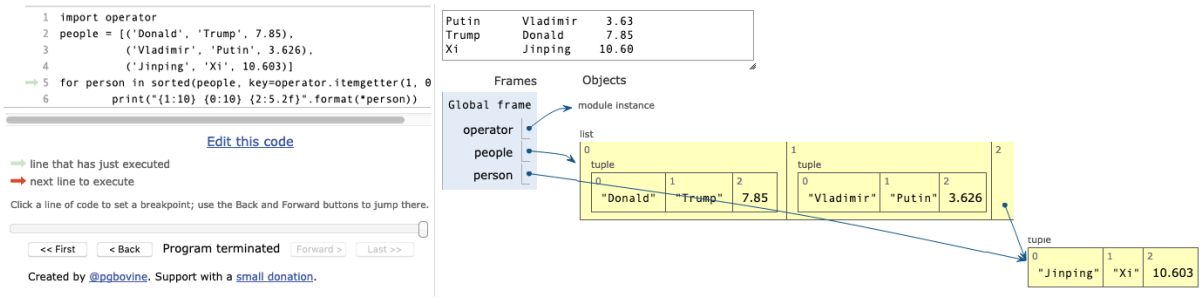


Figure 3.14 Iterating over our list of tuples (from the Python Tutor)

The contents of the tuple then needed to be printed in a strict format. While it's often nice to use Python 3's f-strings, `str.format` (docs.python.org/3/library/stdtypes.html#str.format) can still be useful in some circumstances. Here, I take advantage of the fact that `person` is a tuple, and that `*person`, when passed to a function, becomes not a tuple, but the elements of that tuple. This means that we're passing three separate arguments to `str.format`, which we can access via `{0}`, `{1}`, and `{2}`.

In the case of the last name and first name, we wanted to use a 10-character field, padding with space characters. We can do that in `str.format` by adding a colon (`:`) character after the index

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

we wish to display. Thus, `{1:10}` tells Python to display the item with index 1, inserting spaces if the data contains fewer than 10 characters. Strings are left aligned by default, such that the names will be displayed flush left within their columns.

The third column is a bit trickier, in that we wanted to display only two digits after the decimal point, a maximum of 5 characters, to have the travel-time decimal aligned, and (as if that weren't enough) to pad the column with space characters.

In `str.format` (and in f-strings), each type is treated differently. So if we simply give `{2:10}` as the formatting option for our floating-point numbers (i.e., `person[2]`), the number will be right-aligned. We can force it to be displayed as a floating-point number if we put an `f` at the end, as in `{2:10f}`, but that will just fill with zeroes after the decimal point. And indeed, we only want two digits after the decimal point, with a maximum of five digits total. The specifier for that would be `{5.2f}`---which produces the output we wanted.

3.6.3 Beyond the exercise

Here are some ideas you can use to extend this exercise, and learn more about similar data structures:

1. If you find tuples annoying because they use numeric indexes, you're not alone! Re-implement this exercise using `namedtuple` objects, defined in the `collections` module. Many people like to use named tuples because they give the right balance between readability and efficiency.
2. Define a list of tuples, in which each tuple contains the name, length (in minutes), and director of the movies nominated for "best picture" Oscar awards last year. Ask the user whether they want to sort the list by title, length, or director's name, and then present the list sorted by the user's choice of axis.
3. Extend the above exercise by allowing the user to sort by two or three of these fields, not just one of them. The user can specify the fields by entering them separated by commas; you can use `str.split` to turn them into a list.

3.7 Summary

In this chapter, we explored a number of ways in which we can use lists and tuples, and manipulate them within our Python programs. It's hard to exaggerate just how common lists and tuples are, and how familiar you should be with them. To summarize some of the most important points to remember about them:

1. Lists are mutable and tuples are immutable, but the real difference between them is how they are used: Lists are for sequences of the same type, and tuples are for records that contain different types.
2. You can use the built-in `sorted` function to sort either lists or tuples. You'll get a list back from your call to `sorted`.
3. You can modify the sort order by passing a function to the `key` parameter. This function

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

will be invoked once for each element in the sequence, and the output from the function will be used in ordering the elements.

4. If you want to count the number of items contained in a sequence, try using the `Counter` class from the `collections` module. It not only lets us count things quickly and easily, and provides us with a `most_common` method, but also inherits from `dict`, giving us all of the dictionary functionality we know and love.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

4 Dictionaries and sets

Dictionaries (https://docs.python.org/3/library/stdtypes.html#mapping-types-dict) are one of Python's most powerful and important data structures. You may recognize them from other programming languages, in which they can be known as "hashes," "associative arrays," "hash maps," or "hash tables."

In a dictionary, we don't enter individual elements, as in a list or tuple. Rather, we enter pairs of data, with the first item known as the "key," and the second item known as the "value."

This seemingly small difference, that we can use arbitrary keys to locate our values, rather than using integer indexes starting at 0, is actually crucial. Many programming tasks involve name-value pairs — such as usernames/user IDs, IP addresses/hostnames, e-mail addresses/encrypted passwords. Moreover, much of the Python language itself is implemented using dicts. So knowing how dicts work, and how to use them better yourself, will give you insights into the actual implementation of Python.

There are three main ways that I use dictionaries:

1. As small databases, or records. It's often convenient to use dicts for storing name-value pairs. We can load a configuration file into Python as a dict, retrieving the values associated with configuration options. We can store information about a file, or a user's preference, or a variety of other things with standard names and unknown values. When used this way, a dictionary is defined once, often at the top of a program, and doesn't change.
2. For storing closely related names and values. Rather than create a number of separate variables, you can create a dictionary with several key-value pairs. I do this when I want to store (for example) several pieces of information about a Web site, such as its URL, my username, and the last date I visited. Sure, you could use several variables to keep track of this information, but a dict lets you manage it more easily—as well as pass it to a function or method all at once, via a single variable.
3. For accumulating information over time. If you're keeping track of which errors have occurred in your program, and how many times each has happened, a dictionary can be a great way to do this. You can also use one of the classes that inherit from `dict`, such as

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

`Counter` or `defaultdict`, both defined in the `collections` module (https://docs.python.org/3/library/collections.html). When used this way, a dictionary grows over time, adding new key-value pairs and updating the values as the program executes.

You will undoubtedly find additional ways to use dictionaries in your programs, but these are the three that occur most often in my work.

4.1 Hashing and dictionaries

From what I've written so far, it might sound like any Python object can be used as the key or value in a dictionary. But that's not true: While absolutely anything can be stored in a Python value, only "hashable" types, meaning those on which we can run the `hash` function, can be used as keys. This same `hash` function ensures that a dictionary's keys are unique, and that searching for a key can be quite fast.

What's a hash function? Why does Python use one? And how does it affect what we do?

The basic idea is as follows: Let's assume that you have an office building with 26 offices. If a visitor comes to the building looking to meet with a Ms. Smith, how can he know where to find her? Typically, there will be a receptionist or office directory. Without such help, the visitor will need to go through the offices, one by one, looking for Ms. Smith's office.

This is the way that we search through a string, list, or tuple in Python. The time it takes to find a value in such a sequence is described in computer science literature as `O(n)`. This means that as the sequence gets longer, finding what you're looking for takes proportionally more time.

Now let's re-imagine our office environment. There's still no directory or receptionist. But there is a sign saying that if you're looking for an employee, then just go to the office whose number matches the first letter of their last name — using the scheme a=1, b=2, c=3, and so forth.

Since we want to find Ms. Smith, we calculate that S is the 19th letter in the English alphabet, go to room 19, and are delighted to find that she's there.

If we're looking for Mr. Jones, of course, we would instead go to room 10, since J is the 10th letter of the alphabet.

This sort of search, as you can see, doesn't require much time at all. Indeed, it doesn't matter whether our company has 2 employees or 25 employees — as the company grows, visitors can still find our employees' offices in the same amount of time. This is known in the programming world as `O(1)`, or "constant time," and it's pretty hard to beat.

Of course, there is a catch: What if there are two people whose last names begin with "S"? There are a few different ways that we can solve this problem, but we'll assume that it's solvable. For example, use the first two letters of the last name. Or we can have all of the people whose names

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

begin with "S" share an office. Then we have to search through all of the people in a given office, which is typically not going to be too terrible.

The description I gave you here is a simplified version of a hash function, which are used in a variety of places in the programming world. Hash functions are especially popular for cryptography and security, because while their mapping of inputs to outputs is deterministic, it's virtually impossible to calculate without using the hash function itself. However, they're also central to how Python's dictionaries work.

A dictionary entry consists of a key-value pair. The key is passed to Python's hash function, which returns the location at which the key-value pair should be stored. So if you say `d['a'] = 1`, then Python will executed `hash('a')`, and use the result to store the key-value pair. And when you ask for the value of `d['a']`, Python can invoke `hash('a')` and immediately check in the indicated memory slot whether the key-value pair is there.

Dictionaries are called "mappings" in the Python world, because the hash function "maps" our key to an integer, which we can then use to store our key-value pairs.

I'm leaving out a number of details here, including the significant behind-the-scenes changes that occurred in Python 3.6. These changes guaranteed that key-value pairs will be stored (and retrieved) in chronological order, and reduced memory usage by about one third. But this mental model should help to explain how dicts accomplish search times of `O(1)` (i.e., constant time, regardless of how many key-value pairs are added), and why they're used not only by Python developers, but by the language itself. You can learn more about this new implementation in a great talk by Raymond Hettinger at https://www.youtube.com/watch?v=p33CVV29OG8 .

Keys are guaranteed to be unique, thanks to the hash function. Moreover, searching for a key is very fast, often described as `O(1)`, or "constant time." This means that no matter how large your dictionary gets, looking for keys will still take a short amount of time. (This is also explained in greater depth in the "hashing" sidebar.) By contrast, as a list grows, searching for an item takes proportionally longer, known in the computer-science world as `O(n)` — because finding an element requires potentially iterating over the entire list.

4.2 Sets

Closely related to dictionaries are sets (https://docs.python.org/3/library/stdtypes.html#set), which you can think of as dictionaries without values. (I often joke that this means sets are actually immoral dictionaries.) Sets are extremely useful when I need to look something up in a large collection, such as filenames, e-mail addresses, or postal codes, because searching is `O(1)`, just as in a dict. I have also increasingly found myself using sets to remove duplicate values from an input list — such as IP addresses in a logfile, or license plate numbers that have passed through a parking garage entrance in a given day.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

In this chapter, you'll use dictionaries and sets in a variety of ways to solve problems. It's safe to say that nearly every Python program uses dictionaries, or perhaps an alternative dictionary such as `defaultdict` from the `collections` module.

Table 4.1 Reference table

Name	Description	Example	Link
`input`	prompt the user to enter a string, and return a string	`input("Enter your name: ")`	input
dicts	Python dictionaries	`d = {'a':1, 'b':2}`	mapping-types-dict
`d[k]`	Retrieves the value associated with key `k` in dictionary `d`	`x = d.get(k, 10)`	mapping-types-dict
`dict.get`	Just like `d[k]`, except that it returns `None` (or the second, optional argument) if `k` isn't in `d`	`x = d.get(k, 10)`	#dict.get
`dict.items`	Returns an iterator that returns a key-value pair (as a tuple) with each iteration	`for key, value in d.items():`	#dict.items
Dictionary comprehensions	Create a dictionary, based on an existing iterable	`{one_word : len(one_word) for one_word in sentence.split()}`	pep-0274
sets	Python sets	`s = {1,2,3} # creates a 3-element set`	set.add
`set.add`	Adds one item to a set	`s.add(10)`	frozenset.add
`set.update`	Adds the elements of one or more iterables to a set	`s.update([10, 20, 30, 40, 50])`	hfrozenset.update
`str.isdigit`	Returns `True` if all of the characters in a string are digits 0-9.	`'12345'.isdigit() # returns True`	str.isdigit

4.3 How many different numbers?

In my consulting work, I'm sometimes interested in finding all of the different error messages, IP addresses, or usernames in a logfile. But if a message, address, or username appears twice, then there's no added benefit. I'd thus like to ensure that I'm looking at each value once and only once, without the possibility of repeats.

In this exercise, you can assume that your Python program contains a list of integers. We want to print the number of different integers contained within that list. Thus, consider the following:

```
numbers = [10, 20, 30, 40, 30, 40, 50]
```

With the above definition, running `len(numbers)` will return 7, because the list contains 7 elements. How can we get a result of 5, reflecting the fact that the list contains 5 different values?

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

4.3.1 Solution

```
numbers = [1, 2, 3, 1, 2, 3, 4, 1]
unique_numbers = set(numbers)    ❶
print(len(unique_numbers))
```

❶ You can create a set either with the `set` class, or with curly braces. When converting from lists to sets, you must use the word `set`, since curly braces around a list will result in an error.

You can work through this code in the Python Tutor at https://tinyurl.com/y2ehe2ww .

4.3.2 Discussion

A set, by definition, contains unique elements — just as a dictionary's keys are guaranteed to be unique. Thus, if you ever have a list of values from which you want to remove all of the duplicates, you can just create a set. You can create the set as in the above, example:

```
unique_numbers = set(numbers)
```

or you can do so by creating an empty set, and then adding new elements to it:

```
numbers = [1, 2, 3, 1, 2, 3, 4, 1]
unique_numbers = set()
for number in numbers:
    unique_numbers.add(number)
```

The above example uses `set.add`, which adds one new element to a set. You can add items en masse with `set.update`, which takes an iterable as an argument:

```
numbers = [1, 2, 3, 1, 2, 3, 4, 1]
unique_numbers = set()
unique_numbers.update(numbers)    ❶
```

❶ You can only use `set.update` with an iterable. Think of it as shorthand for running a `for` loop on each of the elements of `numbers`, and running `set.add` on each one.

4.3.3 Beyond the exercise

Whenever I hear the word "unique" or "different" in a project's specification, I think of sets, because they automatically enforce uniqueness and work with a sequence of values. (Dicts, by contrast, work with pairs of values.) So if you have a sequence of usernames, dates, IP addresses, e-mail addresses, or products, and want to reduce that to a sequence containing the same data, but with each item appearing only once, then sets can be extremely useful.

Here are some things you can try to work with sets even more:

1. Read through a server (e.g., Apache or nginx) logfile. What were the different IP

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

addresses that tried to access your server?

2. Reading from that same server log, what were the different response codes that were returned to users? 200 represents "OK," but there are also 403, 404, and 500 errors. (Regular expressions aren't required here, but will probably help.)
3. Use `os.listdir` to get the names of files in the current directory. What are the different file extensions (i.e., suffixes following the final `.` character) that appear in that directory? It'll probably be helpful to use `os.path.splitext` (https://docs.python.org/2/library/os.path.html#os.path.splitext) here.

4.4 Flip a dictionary

In many ways, dictionaries are the most important data structure in Python. Knowing how to create a dictionary based on other data structures, and how to take an existing dictionary and turn it into other data structures, is important. You'll also likely need to take an existing dictionary and modify it, removing or modify certain elements. For example, you might want to remove all users whose ID number is less than 500. Or you might want to find the user IDs of all users whose names begin with the letter "a".

It's also not uncommon to want to flip a dictionary — that is, to exchange its keys and values. Imagine a dict in which the keys are usernames and the values are user ID numbers; it might be useful to flip that, so that you can search by ID number.

To do this exercise, first create a dictionary of any size, in which the keys are unique and the values are also unique. (A key may appear as a value, or vice versa.) For example:

```
d = {'a':1, 'b':2, 'c':3}
```

You are to turn the dictionary inside out, such that the keys and the values are reversed.

4.4.1 Solution

```
d = {'a':1, 'b':2, 'c':3}
flipped_d = { value : key
              for key, value in d.items() }   ❶
print(flipped_d)
```

❶ All iterables are acceptable in a comprehension. Even those that return 2-element tuples, such as `dict.items`.

You can work through this code in the Python Tutor at https://tinyurl.com/y5ftjhcg .

4.4.2 Discussion

This program assumes that you know how to create or use a dictionary comprehension; if not, see the reference table at the top of the chapter for a link to the original Python document that introduces them. As the name implies, a dict comprehension returns a dictionary. The key is named before the `:` character, and the value is named after the `:`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

In this particular case, we're looping over the elements of a dictionary, named `d`. We use the `dict.items` method to do so, which returns two values---the key and value---with each iteration. These two values are passed by parallel assignment to the variables `key` and `value`.

Another way of solving this exercise is to iterate over `d`, rather than over the output of `d.items()`. That would provide us with the keys, requiring that we retrieve each value:

```
{ d[key]:key for key in d }
```

If you're not familiar with dictionary comprehensions, and want to solve this in a more traditional (if longer) way, you can say:

```
output = { }

for key in d:
    output[d[key]] = key
```

However, I'd argue that learning to use dictionary comprehensions effectively is an important part of Python development. Moreover, traditional `for` loops might be easier to understand and use if you're new to Python, but they serve a different purpose than comprehensions. In a comprehension, I'm trying to create a new object based on an old one. It's all about the values that are returned by the expression at the start of the comprehension. By contrast, `for` loops are about commands, and executing those commands.

Consider what your goal is, and whether you're better served with a comprehension or a `for` loop.

Some examples:

1. Given a string, you want a list of the `ord` values for each character? This should be a list comprehension, because you're creating a list based on a string, which is iterable.
2. You have a list of dicts, in which each dict contains your friends' first and last names, and want to insert this data into a database. In this case, you'll use a regular `for` loop, because you're interested in the side effects, not the return value.

.

.

4.4.3 Beyond the exercise

Dict comprehensions provide us with an extremely useful way to create new dictionaries. They're typically used when you want to create a dict based on an iterable, such as a list or file. I'm especially partial to using them when I want to read from a file, and turn the file's contents into a dict. Here are some additional ideas for ways to practice the use of dict comprehensions:

1. Given a string containing several (space-separated) words, create a dict in which the keys

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

are the words, and the values are the number of vowels in each word. If the string is 'this is an easy test', then the resulting dict would be `{'this':1, 'is':1, 'an':1, 'easy':2, 'test':1}`.

2. Create a dictionary whose keys are filenames and whose values are the lengths of the files. The input can be a list of files from `os.listdir` or `glob.glob`.
3. Find a configuration file in which the lines look like "name=value". Use a dict comprehension to read from the file, turning each line into a key-value pair. (If you wish, you can do this for Unix's `/etc/passwd` file, creating a dict from the usernames (index 0) and user ID number (index 2) on each line containing a user record.

4.5 Rainfall

Dictionaries have a variety of users, including keep track of accumulated data over the life of a program. In this exercise, you'll use a dict for just that.

You want to track rainfall in a number of cities. Users of your program will enter the name of a city; if the city name is blank, then the program exits and prints a report (described below). If the city name isn't blank, then the user should also be asked how much rain has fallen in that city (typically measured in millimeters). After entering the quantity of rain, the user is again asked for a city name, rainfall amount, and so on---until the user presses "Enter" instead of typing the name of a city.

When the user enters a blank city name, the program exits---but first, it reports how much total rainfall there was in each city. Thus, if I enter:

```
Boston
5
New York
7
Boston
5
[Enter; blank line]
```

The program should output:

```
Boston: 10
New York: 7
```

The order in which the cities appear is not important, and the cities aren't known to the program in advance.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

4.5.1 Solution

```
rainfall = { }                                                          ❶

while True:
    city_name = input("Enter city name: ")
    if not city_name:
        break

    mm_rain = input("Enter mm rain: ")                                  ❷
    rainfall[city_name] = rainfall.get(city_name, 0) + int(mm_rain)     ❸

for city,rain in rainfall.items():
    print(f"{city}: {rain}")
```

❶ We don't know what cities the user will enter. So we create an empty dictionary, ready to be filled.

❷ If you're from the US, then you might be surprised to hear that other countries measure rainfall in millimeters.

❸ The first time we encounter a city, we'll add 0 to its current rainfall. Any subsequent time, we'll add the current rainfall to the previously stored rainfall. `dict.get` makes this possible.

You can work through this code in the Python Tutor at https://tinyurl.com/y5aet22j .

4.5.2 Discussion

This program uses dictionaries in a classic way, as a tiny database of names and values that grows over the course of the program. In the case of this program, we use the `rainfall` dictionary to keep track of the cities and the amount of rain that has fallen there to date.

We use an infinite loop, which is most easily accomplished in Python with `while True`. Only when the program encounters `break` will it exit from the loop.

At the top of each loop, we get the name of the city for which the user is reporting rainfall. As we have already seen, Python programmers typically don't check to see if a string is empty by checking its length. Rather, they check to see if the string contains a true or false value in a boolean context. If a string is empty, then it will be false in the `if` statement. Our statement `if not city_name` means, "If the `city_name` variable contains a false value," or in simpler terms, "if `city_name` is empty."

Let's walk through the execution of this program with the examples from the beginning of the chapter, and see how the program works:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

Figure 4.1 Asking the user for the first input

When the user is asked for input the first time, the user is presented with a prompt. The `rainfall` dictionary has already been defined, and we're looking to populate it with a key-value pair.

Figure 4.2 After adding the key-value pair to the dict

After entering a city name (`Boston`), we enter the amount of rain that fell (`5`).

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

Because this is the first time that `Boston` has been listed as a city, we add a new key-value pair to `rainfall`. We do this by assigning the key `Boston` and the value `5` to our dict. Notice that this code uses `dict.get` with a default, to either get the current value associated with `Boston` (if there is one), or `0` if there isn't. The first time we ask about a city, there is no key named `Boston`, and certainly no previous rainfall.

Figure 4.3 Adding to an existing name-value pair

There are two parts to this exercise that often surprise or frustrate new Python programmers:

The first is that `input` (https://docs.python.org/3/library/functions.html?#input) returns a string. This is fine when the user enters a city, but not as good when the user enters the amount of rain that fell. Storing the rainfall as a string works relatively well when a city is entered only once. However, if a city is entered more than once, the program will find itself having to add (with the `+` operator) two strings together. Python will happily do this, but the result will be a newly concatenated string, rather than the value of the added integers.

For this reason, we invoke `int` on `mm_rain`, such that we get an integer. If you want, you could replace `int` with `float`, and thus get a floating-point value back. Regardless, it's important that if you use `input` to get input from the user, and if you want to use a numeric value rather than a string, you must convert it.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

NOTE My solution deliberately doesn't check to see if the user's input can be turned into an integer. Which means that if the user enters a string containing something other than the digits 0-9, the call to `int` will return an error. I didn't want to complicate the solution code too much.

If you do want to trap such errors, then you have two basic options: One is to wrap the call to `int` inside of a `try` block. If the call to `int` fails, we can catch the exception. For example:

```
try:
    mm_rain = int(input("Enter mm rain: "))
except ValueError:
    print("You didn't enter a valid integer; try again.")
    continue

rainfall[city_name] = rainfall.get(city_name, 0) + mm_rain
```

In the above code, we let the user enter whatever they want. If we encounter an error (exception) when converting, we send the user back to the start of our `while` loop, when we ask for the city name. A slightly more complex implementation would have the user simply re-enter the value of `mm_rain`.

A second solution is to use the `str.isdigit` method, which returns `True` if a string contains only the digits 0-9, and `False` otherwise. For example:

```
mm_rain = input("Enter mm rain: ").strip()
if mm_rain.isdigit():
    mm_rain = int(mm_rain)
else:
    print("You didn't enter a valid number; try again.")
    continue
```

Once again, this would send the user back to the start of the `while` loop, asking them to enter the city name once again. It also assumes that we're only interested in getting integer values, because `str.isdigit` returns `False` if you give it a floating point number.

The second tricky part of this exercise is that you must handle the first time a city is named (i.e., before the city's name is a key in `rainfall`), as well as subsequent times.

The first time that someone enters `Boston` as a city name, we will need to add the key-value pair for that city and its rainfall into our dict. The second time that someone enters `Boston` as a city name, we need to add the new value to the existing one.

One simple solution to this problem is to use the `dict.get` method with two arguments. With one argument, `dict.get` either returns the value associated with the named key or `None`. But with two arguments, `dict.get` returns either the value associated with the key or the second

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

argument.

Thus, when we call `rainfall.get(city_name, 0)`, Python checks to see if `city_name` already exists as a key in `rainfall`. If so, then the call to `rainfall.get` will return the value associated with that key. If `city_name` is not in `rainfall`, we get `0` back.

An alternative solution would use the `defaultdict` (https://docs.python.org/3/library/collections.html#collections.defaultdict), a class defined in the `collections` (https://docs.python.org/3/library/collections.html) module that allows you to define a dictionary that works just like a regular one---until you ask it for a key that doesn't exist. In such cases, `defaultdict` invokes the function with which it was defined. For example:

```
from collections import defaultdict
rainfall = defaultdict(int)  ❶
rainfall['Boston'] += 30
rainfall                 # defaultdict(<type 'int'>, {'Boston': 30})

rainfall['Boston'] += 30

rainfall                 # defaultdict(<type 'int'>, {'Boston': 60})
```

❶ `defaultdict(int)` means that if we say `rainfall[k]` and `k` isn't in `rainfall`, the `int` function will execute without any arguments, giving us the int 0 back.

4.5.3 Beyond the exercise

It's pretty standard to use dictionaries to keep track of accumulated values (such as the number of times something has happened, or amounts of money) associated with arbitrary values. The keys can represent what you're tracking, and the values can track data having to do with the key. Here are some additional things you can do:

1. Instead of printing the total rainfall for each city, print the total, as well as the average rainfall for reported days. Thus, if you were to enter 30, 20, and 40 for Boston, you would see that the total was 90, and that the average was 30.
2. Open a logfile from a Unix/Linux system—for example, one from the Apache server. For each response code (i.e., three-digit code indicating the HTTP request's success or failure), store a list of IP addresses that generated that code.
3. Read through a text file on disk. Use a dict to track how many words of each length are in the file—that is, how many 3-letter words, 4-letter words, 5-letter words, and so forth. Display your results.

4.6 Dictdiff

Dictionaries are central to Python development; knowing how to work with them is crucial to your Python career. Moroever, once your learn how to use `dict.get` effectively, you'll find that your code is shorter, more elegant, and more maintainable.

Write a function, `dictdiff`, that takes two dictionaries as arguments. The function returns a new

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

dictionary that expresses the difference between the two dictionaries.

If there are no differences between the dictionaries, then `dictdiff` returns an empty dictionary.

For each key-value pair that differs, the return value of `dictdiff` will have a key-value pair in which the value is a list containing the values from the two different dictionaries. If one of the dictionaries doesn't contain that key, it should contain None.

Thus:

```
d1 = {'a':1, 'b':2, 'c':3}
d2 = {'a':1, 'b':2, 'c':4}
print(dictdiff(d1, d1))   ❶
print(dictdiff(d1, d2))   ❷

d3 = {'a':1, 'b':2, 'd':3}
d4 = {'a':1, 'b':2, 'c':4}
print(dictdiff(d3, d4))   ❸

d5 = {'a':1, 'b':2, 'd':4}
print(dictdiff(d1, d5))   ❹
```

❶ prints `{}`, because we're comparing `d1` with itself

❷ prints `{'c': [3, 4]}`, because `d1` contains `c:3` and `d2` contains `c:4`

❸ prints `{'c': [None, 4], 'd': [3, None]}`

❹ prints `{'c': [3, None], 'd': [None, 4]}`, because `d1` has `c:3` and `d5` has `d:4`

4.6.1 Solution

```
def dictdiff(first, second):
    output = {}
    all_keys = set(first.keys()) | set(second.keys())          ❶

    for key in all_keys:
        if first.get(key) != second.get(key):
            output[key] = [first.get(key), second.get(key)]    ❷
    return output
```

❶ Get all keys from both `first` and `second`, without repeats

❷ We take advantage of the fact that `dict.get` returns `None` when a key doesn't exist

You can work through this code in the Python Tutor at https://tinyurl.com/yxn7qsfe .

4.6.2 Discussion

Let's start by thinking about the overall design of this program:

1. We create an empty `output` dictionary.
2. We go through each of the keys in `first` and `second`.
3. For each key, we check if the key also exists in the other dictionary.
4. If the key exists in both, then we check if the values are the same.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

5. If the values are the same, then we do nothing to `output`.
6. If the values are different, then we add a key-value pair to `output`, with the currently examined key and a list of the values from `first` and `second`.
7. If the key doesn't exist in one dict, then we use `None` as the value.

This all sounds good, but there's a problem with this approach: It means that we're going through each of the keys in `first` and then each of the keys in `second`. Given that at least some keys will hopefully overlap, this sounds like an inefficient approach.

It would be better and smarter for us to collect all of the keys from `first` and `second`, to put them into a set (thus ensuring that each appears only once), and then to iterate over them.

The thing is, we can't easily add the dictionaries' keys together. The next best thing is to run the `dict.keys` method on each of the dicts, and feeding this method's output into a set. We can then use the `|` operator, aka the `union` operation on our sets, getting a new set back. And of course, sets are iterable, allowing us to put the output from this line into a `for` loop. All that is happening in this line:

```
all_keys = set(first.keys()) | set(second.keys())
```

NOTE In Python 2, `dict.keys` and many similar methods returned lists, which do support the `+` operator. In Python 3, almost all such methods were modified to return iterators. When the returned result is small, there's almost no difference between the implementations. But when the returned result is large, then there is a big difference, and most everyone prefers to use an iterator. Thus, the behavior in Python 3 is preferable, even if it's a bit surprising for people moving from Python 2.

Because a set is effectively a dictionary without values, we know for sure that by putting these lists into our `all_keys` set, we'll only pass through each key once.

Rather than checking whether a key exists in each dictionary, and then retrieving its value, and then checking whether the values are the same, I used the `dict.get` (https://docs.python.org/3/library/stdtypes.html#dict.get) method. This saves us from getting a `KeyError` exception. Moreover, if one of the dictionaries lacks the key in question, we get `None` back. We can use that not only to check whether the dicts are the same, but also to retrieve the values.

Now let's walk through each of the examples I gave as part of the problem description, and see what happens:

```
d1 = {'a':1, 'b':2, 'c':3}
print(dictdiff(d1, d1))
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

We see this here, in the following screenshot:

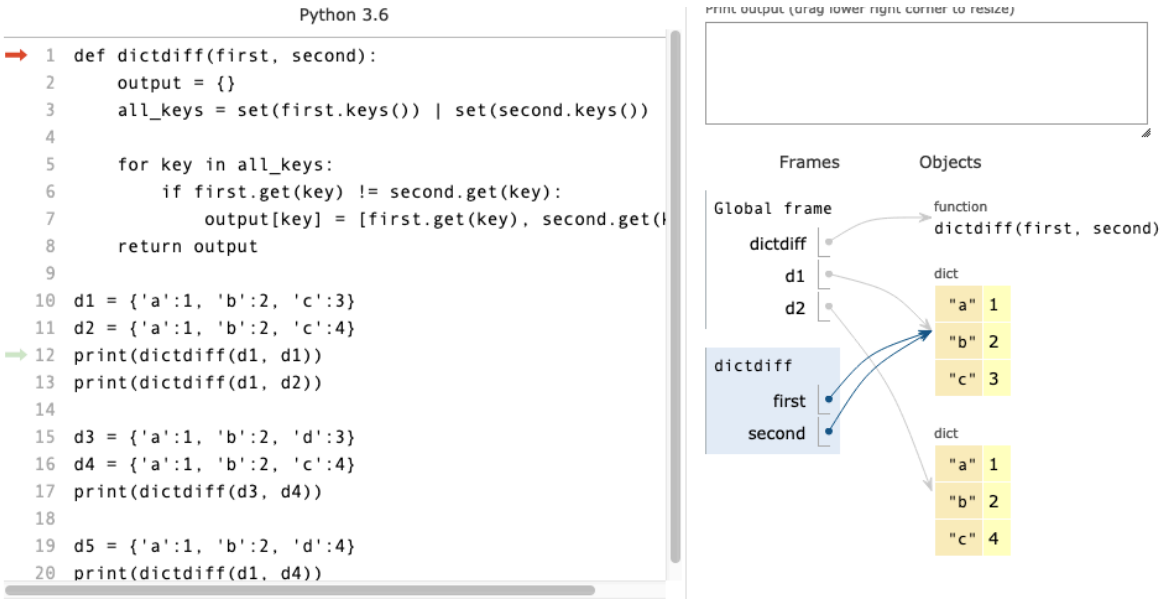


Figure 4.4 Taking the diff of d1 and itself

In this image, we see that the local variables `first` and `second` both point to the same dictionary, `d1`. When we iterate over the combined set of keys, we're actually iterating over the keys of `d1`:

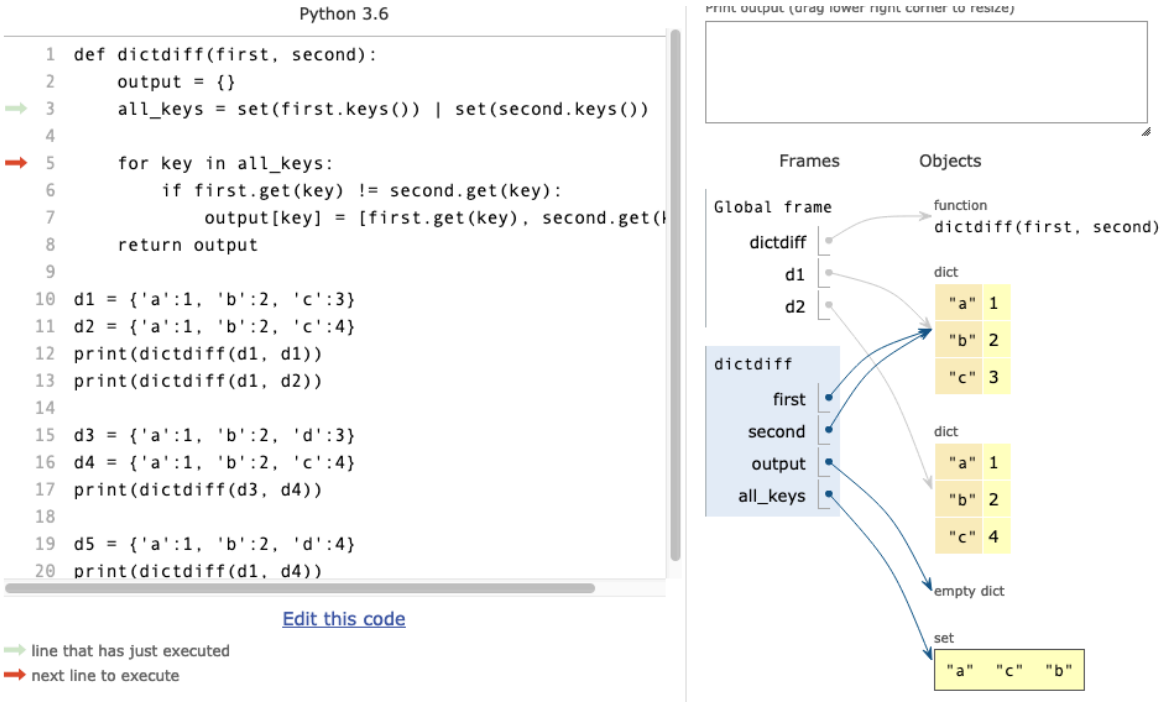


Figure 4.5 Iterating over the keys of d1

Because we never find any differences, the return value (`output`) is `{}`, the empty dict.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

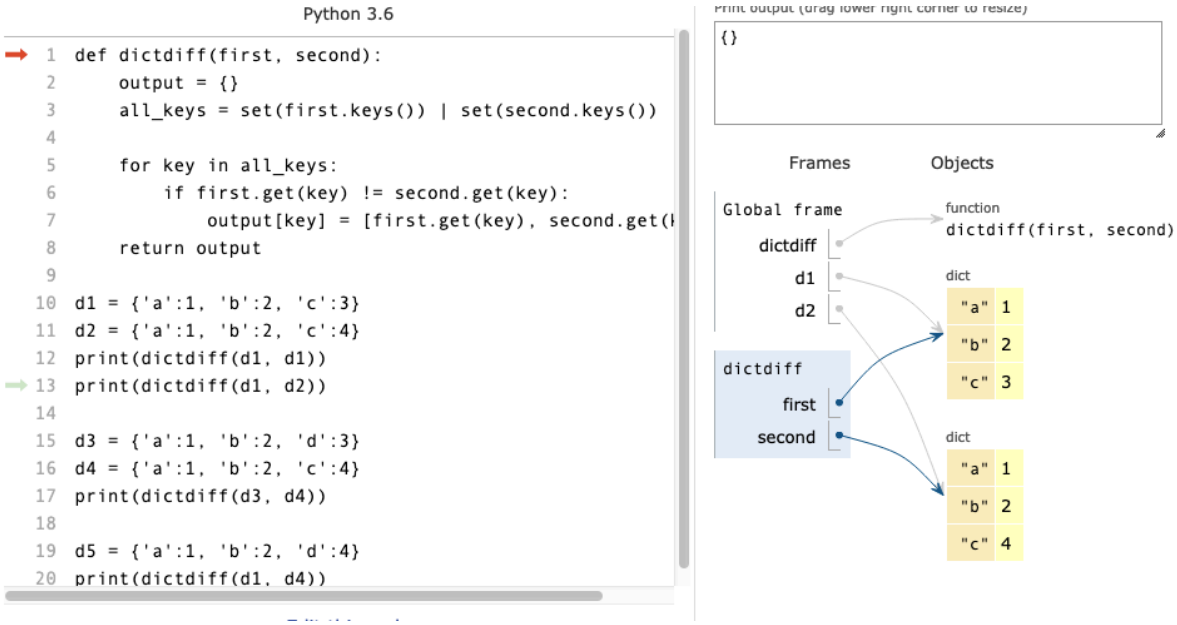


Figure 4.6 Comparing d1 and d2

When we compare `d1` and `d2`, we see that `first` and `second` point to two different dictionaries. They also have the same keys, but different values for the `c` key. We can see in the next image how our `output` dictionary gets a new key-value pair, representing the `c` key's different values:

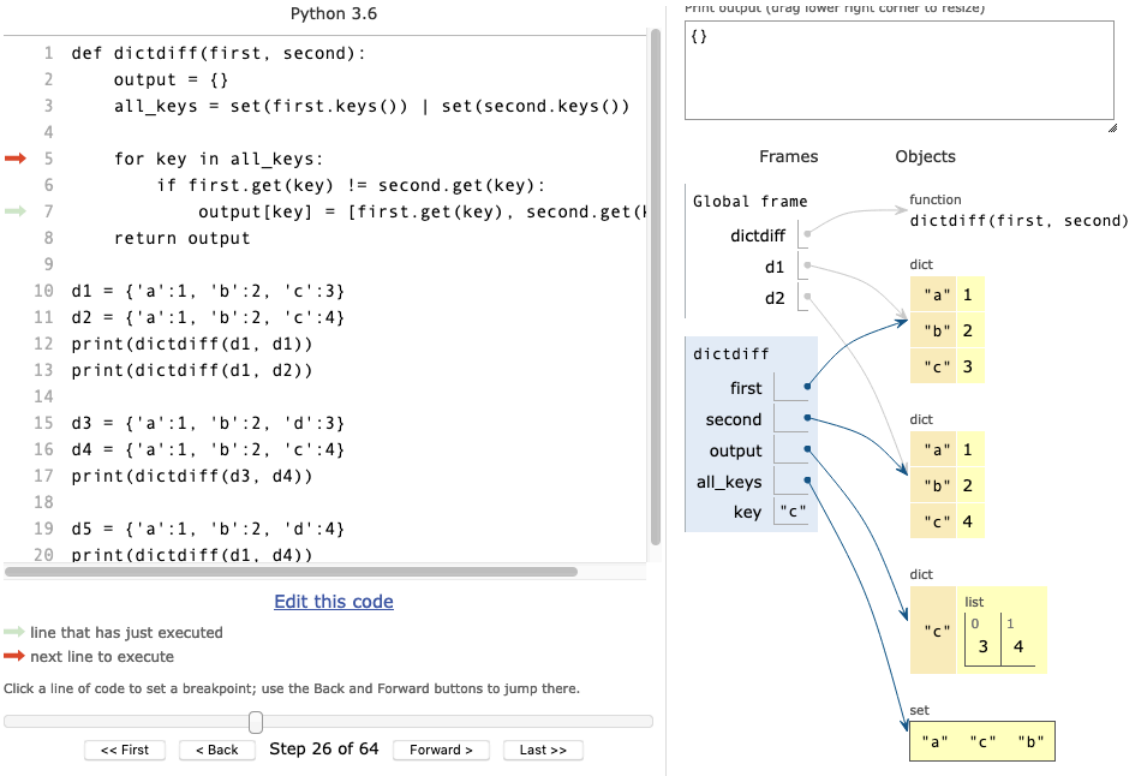


Figure 4.7 Adding a value to output

When we compare `d3` and `d4`, we can see how things get more complex: Our `output` dictionary will now have two key-value pairs, and each value will be (as specified) a list:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

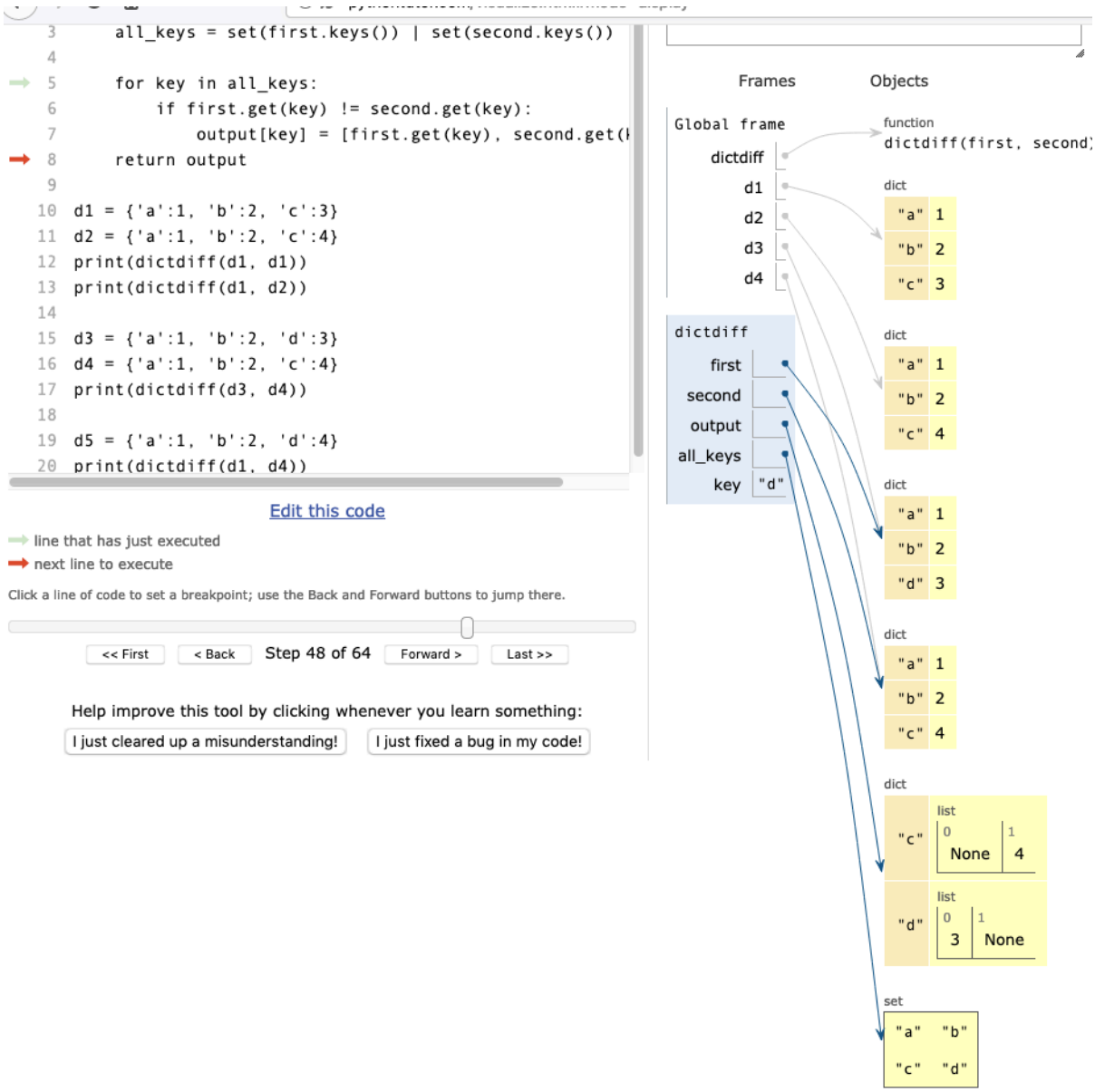


Figure 4.8 Comparing dicts with different keys and values

In this way, you can see how we build our dictionary from nothing to become a report, describing the differences between the two arguments.

4.6.3 Beyond the exercise

Python functions can return any object they like, and that includes dicts. It's often useful to write a function that creates a dict; the function can combine or summarize other dictionaries (as in this exercise), or it can turn other objects into dictionaries. Here are some ideas that you can pursue:

1. The `dict.update` method merges two dicts. Write a function that takes any number of dicts, and returns a dictionary that reflects the combination of all of them. If the same key appears in more than one dictionary, then the most recently merged dict's value should appear in the output.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

2. Write a function that takes any even number of arguments and returns a dict based on them. The even-indexed arguments become the dict keys, while the odd-numbered arguments become the dict values. Thus, calling the function with the arguments ('a', 1, 'b', 2) will result in the dict {'a':1, 'b':2} being returned.
3. Write a function, `dict_partition`, that takes one dictionary (`d`) and a function (`f`) as arguments. `dict_partition` will return two dictionaries, each containing key-value pairs from `d`. The decision regarding where to put each of the key-value pairs will be made according to the output from `f`, which will be run on each key-value pair in `d`. If `f` returns `True`, then the key-value pair will be put in the first output dict. If `f` returns `False`, then the key-value pair will be put in the second output dict.

4.7 Summary

Dictionaries are, without a doubt, that most versatile and important data structure in the Python world. Learning to use them effectively and efficiently is a crucial part of becoming a fluent developer. In this chapter, we practiced several ways in which to use them — including tracking counts of elements, and storing data we got from the user. We also saw that you can use `dict.get` to retrieve from a dict without having to fear that the key doesn't exist.

When working with dicts, remember:

1. The keys must be hashable, such as a number or string
2. The values can be anything at all, including another dict
3. The keys are unique
4. You can iterate over the keys in a `for` loop or comprehension
5. Dict comprehensions make it easy to create new dicts

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

5 Files

Files are an indispensable part of the world of computers, and thus of programming. We read data from files, and write to files. Even when something isn't really a file — such as a network connection — we try to use a similar interface to files, because they're so familiar.

To normal, everyday users, there are different types of files — Word, Excel, PowerPoint, and PDF, among others. To programmers, things are both simpler and more complicated. They're simpler, in that we see files as data structures to which we can write strings, and from which we can read strings. But files are also more complicated, in that when we read the string into memory, we might need to parse it into a data structure.

Working with files is one of the easiest and most straightforward things you can do in Python. It's also one of the most common things that we need to do, since programs that don't interact with the filesystem are rather boring.

In this chapter, we'll practice working with files — reading from them, writing to them, and manipulating the data that they contain. Along the way, you'll get used to some of the paradigms that are commonly used when working with Python files, such as iterating over a file's contents and writing to files in a `with` block.

In some cases, we'll work with data in CSV (comma-separated values) or JSON (JavaScript object notation), two common formats handled by modules in Python's standard library. If you've forgotten the basics of CSV and/or JSON, then I have some short reminders in this chapter.

After this chapter, you'll not only be more comfortable working with files. You'll also better understand how you can translate from in-memory data structures (e.g., lists and dictionaries) to on-disk data formats (e.g., CSV and JSON), and back. In this way, files make it possible for us to keep data structures intact, even when the program isn't running, or when the computer is shut down — or even to transfer such data structures to other computers.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

Table 5.1 Reference table

Name	Description	Example	Link
files	Overview of working with files in Python	f = open('/etc/passwd')	#reading-and-writing-files
`with`	Put an object in a "context manager." In the case of a file, ensures it's flushed and closed by the end of the block.	`with open('/etc/passwd') as f`	#with
Context managers	Make your own objects work in `with` statements	`with MyObject() as m:`	python-with-context-managers
`set.update`	Adds elements to a set	`s.update([10, 20, 30])`	#frozenset.update
`os.stat`	Retrieve information (size, permissions, type) about a file	`s = os.stat('/etc/passwd')`	#os.stat
`os.listdir`	Return a list of files in a directory	`filenames = os.listdir('/etc/')`	#os.listdir
`glob.glob`	Return a list of files matching a pattern	`conf_files = glob.glob('/etc/*.conf')`	#glob.glob
dictionary comprehensions	Create a dictionary based on an iterator	`{one_word : len(word) for one_word in 'this is a test'.split()}`	python.org/dev/peps/pep-0274/
`csv`	Module for working with CSV files	`x = csv.reader(f)`	docs.python.org/3.7/library/csv.html
`json`	Module for working with JSON	`x = json.loads(json_string)`	docs.python.org/3.7/library/json.html

5.1 Last line

It's very common for new Python programmers to learn how they can iterate over the lines of a file, printing one line at a time. But what if I'm not interested in each line, or even in most of the lines? What if I'm only interested in a single line — the final line of the file?

Now, retrieving the final line of a file might not seem like a super-useful action. But consider the Unix "head" and "tail" utilities, which show the first and last lines of a file, respectively — and which I use all the time to examine files, particularly logfiles and configuration files. Moreover, knowing how to read specific parts of a file, as opposed to the entire thing, is a useful and practical skill to have.

In this exercise, you should ask the user for the name of a text file. The program then prints the file's final line on the screen.

5.1.1 Solution

```
filename = input("Enter a filename: ")
final_line = ''
for current_line in open(filename):    ❶
    final_line = current_line
print(final_line, end='')    ❷
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

❶ Iterate over each line of the file. You don't need to declare a variable; just iterate directly over the result of `open`

❷ The `end` parameter to the `print` function lets you specify what should be printed after the call to `print`. By default, it's a newline character. Because we're already getting a newline from the file, I use an empty string here.

You can work through a version of this code in the Python Tutor at http://tinyurl.com/y2qrnymc .

NOTE The Python Tutor site, which I use for diagrams and also to allow you to experiment with my solutions, doesn't support files. This is understandable; a free server system that lets people run arbitrary code is hard enough to create and support. Allowing people to work with arbitrary files would add plenty of logistical and security problems.

However, there is a solution: `StringIO`. `StringIO` objects are what Python calls "file-like objects." They implement the same API as `file` objects, allowing us to read from them and write to them just like files. Unlike files, though, `StringIO` objects never actually touch the filesystem.

`StringIO` wasn't designed for use with the Python tutor, although it's a great workaround for the limitations there. More typically, I see (and use) `StringIO` in automated tests. After all, you don't really want to have a test touch the filesystem; that would make things run much more slowly. Instead, you can use a `StringIO` to simulate a file.

If you're doing any software testing, then you should take a serious look at `StringIO`, part of the Python standard library. You can load it with:

```
from io import StringIO
```

The versions of code available online are thus slightly different than the ones in the book itself. However, they should work the same way, allowing you to explore the code visually. Unfortunately, exercises that involve directory listings cannot be papered over as easily, and thus lack any Python Tutor link.

5.1.2 Discussion

The above code uses a number of common Python idioms that I'll explain here — and along the way, we'll see how using these idioms leads not just to more readable code, but also to more efficient execution.

Calling `open` returns a file object. In many other languages, this object is known as a "file handle," a sort of agent or mediator between our program and the outside world. Using such an

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

object allows us to paper over the many different types of filesystems out there, and just think in terms of "a file." Such an object also allows us to take advantage of whatever optimizations, such as buffering, might be in use by the operating system.

Here's how `open` is usually invoked:

```
f = open(filename)
```

In this case, `filename` is a string representing a valid file name. When we invoke `open` with just one argument, it should be a filename. The second, optional, argument is a string indicating whether we want to read from, write to, or append to the file (using `r`, `w`, or `a`), and whether the file should be read by character (the default) or by bytes (the `b` option). (See the section below about the `b` option, and reading the file in byte, or binary, mode.) I could thus more fully write the above line as:

```
f = open(filename, 'r')
```

Because we read from files more often than we write to them, `r` is the default value for the second argument. It's quite usual for Python programs not to specify `r` if reading from a file.

As you can see here, we've put the resulting object into the variable `f`. And because file-like objects are all iterable, returning one line per iteration, it's typical to then say:

```
for line in f:
    print(line)
```

But if you're just planning to iterate over `f` once, then why create it as a variable at all? We can avoid the variable definition, and simply iterate over the (anonymous) file object that `open` returned:

```
for line in open(filename):
    print(line)
```

With each iteration over a file-like object, we get the next line from the file, up to and including the `\n` newline character. Thus, in the above code, `line` is always going to be a string, and always containing a single `\n' character, at the end of the string.

A blank line in a file will contain just the `\n` newline character.

In theory, files should end with a `\n`, such that you'll never finish the file in the middle of a line. In practice, I've seen many files that don't end with a `\n`. Keep this in mind whenever you're printing out a file; assuming that a file will always end with a newline character can cause trouble.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

NOTE

What happens if we open a non-text file, such as a PDF or a JPEG, with `open`, and then try to iterate over it, one line at a time?

First: You'll likely get an error right away. That's because Python expects the contents of a file to be valid UTF-8 formatted Unicode strings. Binary files, by definition, don't use Unicode. When Python tries to read a non-Unicode string, it'll raise an exception, complaining that it cannot define a string with such content.

To avoid that problem, you can and should open the file in "binary" or "bytes" mode, adding a `b` to `r`, `w`, or `a` in the second argument to `open`. For example:

```
for line in open(filename, 'rb'):     ❶
    print(line)     ❷
```

❶ We open the file in `r` (read) and `b` (binary) mode.

❷ The type of `line` here is `bytes`, similar to a string but without Unicode characters

Now you won't be constrained by a lack of Unicode characters.

But wait: Remember that with each iteration, Python will return everything up to and including the next `\n` character. In a binary file, such a character won't appear at the end of every line, because there are no "lines" to speak of. Which means that what you get back from each iteration will probably be nonsense.

The bottom line is that if you're reading from a binary file, then you shouldn't forget to use the `b` flag. But when you do that, you'll find that you don't want to read the file per line anyway. Instead, you should be using the `read` method to retrieve a fixed number of bytes; when `read` returns 0 bytes, you'll know that you're at the end of the file. For example:

```
f = open(filename, 'rb')
while True:
    one_chunk = f.read(1000)     ❶
    if not one_chunk:
        //GH 'break' should be indented
        break
    print(f"This chunk contains {len(one_chunk)} bytes")
```

❶ Read up to 1,000 bytes and return them as a `bytes` object

In this particular exercise, you were asked to print the final line of a file. One way to do this might be:

```
for line in open(filename):
    pass    ❶
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

```
print(line)
```

❶ `pass` simply means, "Don't do anything." It exists because Python requires that we have at least one line in an indented block, such as the body of a `for` loop.

The above trick works because we iterate over the lines of the file and assign `line` in each iteration—but we don't actually do anything in the body of the `for` loop. Rather, we use `pass`, which is a way of telling Python to do nothing. The reason that we execute this loop is for its side effect—namely, the fact that the final value assigned to `line` remains in place after the loop exits.

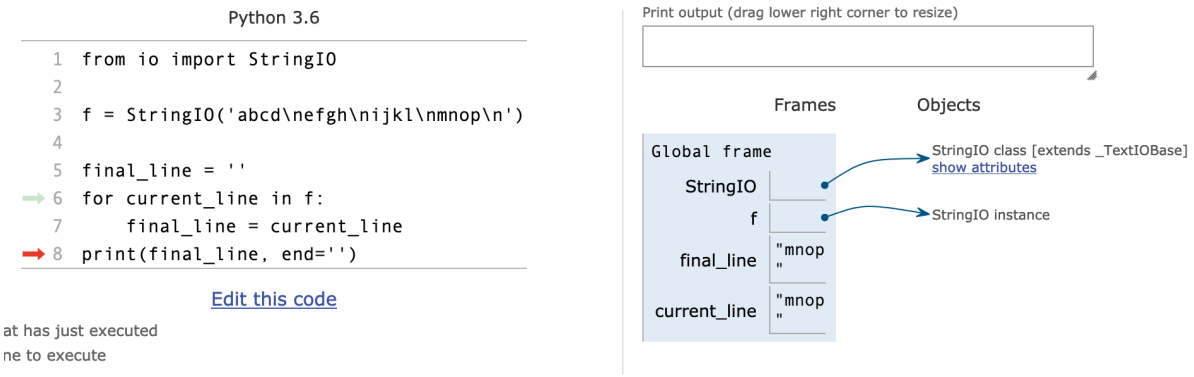


Figure 5.1 Immediately before printing the final line

However, looping over the rows of a file just to get the final one strikes me as a bit strange, even if it works. My preferred solution, as outlined above, is to iterate over each line of the file, getting the current line but immediately assigning it to `final_line`.

When we exit from the loop, `final_line` will contain whatever was in the most recent line. We can thus print it out afterwards.

Normally, `print` adds a newline after printing something to the screen. However, when we iterate over a file, each line already ends with a newline character. This can lead to doubled whitespace between printed output. The solution is to stop `print` from displaying anything, by overriding the default `\n` value in the `end` parameter. By passing `end=''`, we tell `print` to add `''`, the empty string, after printing `final_line`.

5.1.3 Beyond the exercise

Iterating over files, and understanding how to work with the content as (and after) you iterate over them, is an important skill to have when working with Python. It's also important to understand how to turn the contents of a file into a Python data structure, something we'll look at several more times in this chapter. Here are a few ideas for things you can do when iterating through files in this way:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

1. Iterate over the lines of a text file. Find all of the words (i.e., non-whitespace surrounded by whitespace) that contain only integers, and sum them.
2. Create a text file (using an editor, not necessarily Python) containing two tab-separated columns, with each column containing a number. Then read through the file you have created with Python. For each line, multiply each number by the second, and then sum the results. Ignore any line that doesn't contain two numeric columns.
3. Read through a text file, line by line. Use a dict to keep track of how many times each vowel (a, e, i, o, and u) appears in the file. Print the resulting tabulation.

5.2 /etc/passwd to dict

It's both common and useful to think of files as sequences of strings. After all, when you iterate over a file object, you get each of the file's lines as a string, one at a time. But there are many occasions when it makes more sense to turn a file into a more complex data structure, such as a dictionary.

In this exercise, we'll be working with `/etc/passwd`, the file in which Unix and Linux systems store information about their users. If you don't have access to such a file, you can download one that I've uploaded, at https://gist.github.com/reuven/7647d1af56cc8c6a9744 .

Here's a sample of what the file looks like:

```
nobody:*:-2:-2::0:0:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0::0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1::0:0:System Services:/var/root:/usr/bin/false
```

Each line is one user record, divided into colon-separated fields. The first field (index 0) is the username, and the third field (index 2) is user's unique ID number. In the system from which I took the above `/etc/passwd` file, `nobody` has ID `-2`, `root` has ID `0`, and `daemon` has ID `1`. For our purposes, you can ignore all but these two fields.

Sometimes, the file will contain lines that fail to adhere to this format. For example, we generally ignore lines containing nothing but whitespace. Some vendors (e.g., Apple) include comments in their `/etc/passwd` files, in which the line starts with a `#` character.

For this exercise, create a dictionary based on `/etc/passwd`, in which the dict's keys are usernames and the values are the users' IDs. Once you have created the dictionary, iterate over it, displaying the usernames and associated IDs in alphabetical order.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

NOTE The string methods `str.startswith`, `str.endswith`, and `str.strip` are helpful when doing this kind of analysis and manipulation.

For example, `str.startswith` returns `True` or `False`, depending on whether the string starts with a string.

```
s = 'abcd'
s.startswith('a')    # returns True
s.startswith('abc')  # returns True
s.startswith('b')    # returns False
```

Similarly, `str.endswith` tells us whether a string ends with a particular string:

```
s = 'abcd'
s.endswith('d')    # returns True
s.endswith('cd')   # returns True
s.endswith('b')    # returns False
```

`str.strip` removes the whitespace—the space character, as well as `\n`, `\r`, `\t`, and even '\v`—on either side of the string. The `str.lstrip` and `str.rstrip` methods only remove whitespace on the left and right, respectively. All of these methods return strings:

```
s = '   \t\t\ta  b  c  \t\t\n'
s.strip()    # returns 'a  b  c'
s.lstrip()   # returns 'a  b  c  \t\t\n'
s.rstrip()   # returns '   \t\t\ta  b  c'
```

5.2.1 Solution

```
users = {}
with open('/etc/passwd') as f:
    for line in f:
        if not line.startswith("#") and line.strip():    ❶
            user_info = line.split(":")    ❷
            users[user_info[0]] = user_info[2]

for username, user_id in sorted(users.items()):    ❸
    print(f"{username}: {user_id}")
```

❶ Ignore comment and blank lines

❷ Turn the line into a list of strings

❸ Display the users and IDs, sorted by username

You can work through a version of this code in the Python Tutor at https://tinyurl.com/y6228ht9

5.2.2 Discussion

Once again, we're opening a text file and iterating over its lines, one at a time. Here, we assume that we know the file's format, and that we can extract fields from within each record.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

In this case, we're splitting each line across the `:` character, using the `str.split` method. `str.split` always returns a list of strings, although the length of that list depends on the number of times that `:` occurs in the string. In the case of `/etc/passwd`, we will assume that any line containing `:` is a legitimate user record, and thus has all of the necessary fields.

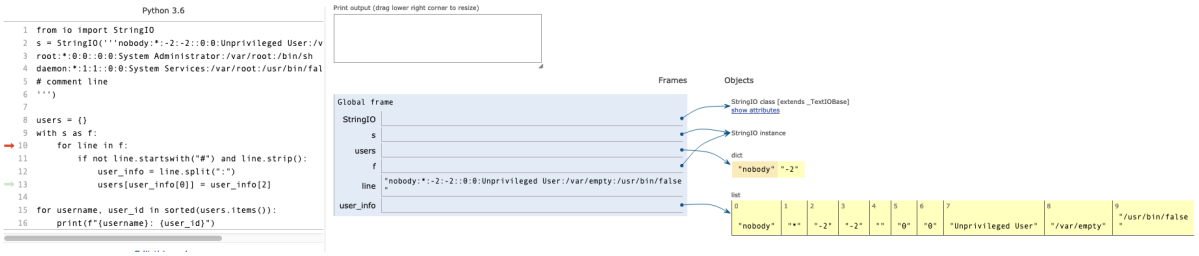


Figure 5.2 Processing one line in /etc/passwd

However, the file might contain comment lines beginning with `#`. If we were to invoke `str.split` (https://docs.python.org/3/library/stdtypes.html?#str.split) on those lines, we would get back a list, but one containing only a single element—leading to an `IndexError` exception if we would try to retrieve `user_info[2]`.

It is thus important that we ignore those lines that begin with `#`. Fortunately, there is a `str.startswith` (https://docs.python.org/3/library/stdtypes.html?#str.startswith) method, which returns `True` if the line starts with the string passed as an argument. By negating this (`if not line.startswith("#")`), we can be sure that we're only splitting `line` for legitimate lines.

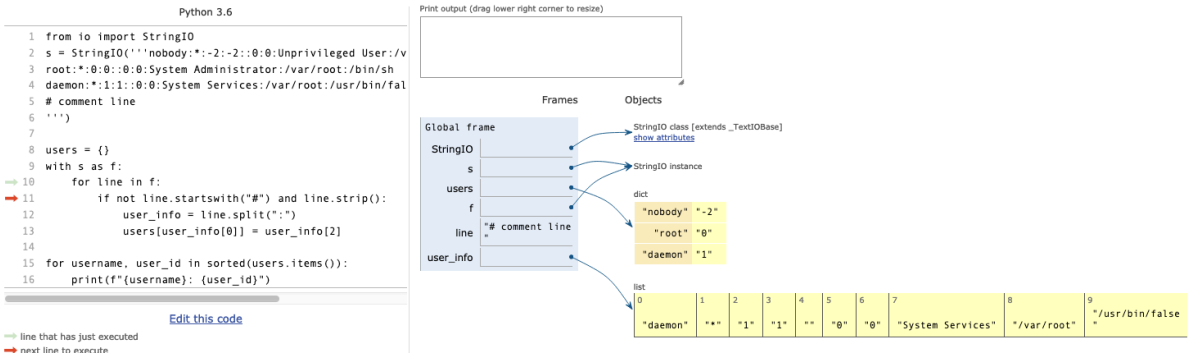


Figure 5.3 Processing a comment line

I also want to ignore any lines that consist solely of whitespace. I use `str.strip` to remove all whitespace from the start and end of the string—and if we get an empty string back, then we know that the line was blank, and ignore it.

Assuming that it has found a user record, our program then adds a new key-value pair to `users`. The key is `user_info[0]`, and the value is `user_info[2]`. Notice how we can use `user_info[0]` as the name of a key; so long as the value of that variable contains a string, we may use it as a dictionary key.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

I use `with` (https://docs.python.org/3/reference/compound_stmts.html#with) here to open the file, thus ensuring that it's closed when the block ends. (See sidebar about `with` and context managers.)

You might have noticed a slight stylistic issue with the above program: I used a `for` loop in order to iterate over the file and extract fields. I could have instead used a dictionary comprehension. In other words, I could have compacted my code greatly by using the following:

```
users =  { line.split(':')[0] : line.split(':')[2]     ❶
           for line in open('/etc/passwd')     ❷
           if not line.startswith('#') and line.strip() }     ❸
```

❶ Return a key-value pair, consisting of the username and user ID

❷ Iterate over the file, one line at a time

❸ Don't return a key-value pair if the line contains only whitespace or starts with #

I love dictionary comprehensions, but recognize that it's sometimes hard to understand all of the functionality inside of them. Furthermore, their syntax can be a bit off-putting to less experienced Python developers. This dict comprehension takes advantage of the fact that we can iterate over the lines of `/etc/passwd`. Each line can then be checked whether it begins with `#` and contains non-whitespace; if it passes these tests, then we split the line across `:`, and return the dictionary with name-ID pairs.

Once we have finished creating our dictionary, we iterate over it and print each key-value pair.

We could, in theory, iterate over the dict itself, and thus get the keys. But I generally prefer to invoke `dict.items`, and get a tuple containing a key-value pair with each iteration. Then I can capture the keys and values in variables (via unpacking), giving me clearer names with which to work. I can also pass the output from `dict.items` to `sorted`, which returns the pairs sorted by key.

Wait a second — does `sorted` know how to sort a dictionary? No, but remember that what we're passing to `sorted` is effectively a list of tuples, output from `dict.items`. When `sorted` encounters a sequence of sequences (e.g., a list of strings or a list of tuples), then it sorts them first by the item at index 0, then by index 1, and so forth. When sorting strings, this makes sense, because we're used to sorting words alphabetically. But it also works for tuples or lists. The output from `dict.items` is a list of tuples, in which each tuple's first element is the dictionary key. Since those are guaranteed to be unique, we know that we'll never need to check the value. The result of `sorted` is thus a list of tuples, in increasing order by key.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

5.2.3 Beyond the exercise

At a certain point in your Python career, you'll stop seeing files as sequences of characters on a disk, and start seeing them as raw material you can transform into Python data structures. Our programs have more semantic power with structured data (e.g., dictionaries) than strings. We can similarly do more and think in deeper ways if we read a file into a data structure rather than just into a string.

For example, imagine a CSV (comma-separated values) file, in which each line contains the name of a country and its population. Reading this file as a string, it would be possible — but frustrating — to compare the populations of France and Thailand. But reading this file into a dictionary, it would be trivial to make such a comparison.

Indeed, I'm a particular fan reading files into dictionaries, in no small part because many file formats lend themselves to this sort of translation — but you can also use more complex data structures. Here are some additional exercises you can try to help you see that connection, and make the transformation in your code:

1. Read through `/etc/passwd`, creating a dict in which user login shells (the final field on each line) are the keys. Each value will be a list of the users for whom that shell is defined as their login shell.
2. Ask the user to enter integers, separated by spaces. From this input, create a dictionary whose keys are the factors for each number, and the values are lists containing those of the users' integers that are multiples of those factors.
3. From `/etc/passwd`, create a dict in which the keys are the usernames (as in the main exercise), and the values are themselves dicts with keys (and appropriate values) for user ID, home directory, and shell.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

SIDEBAR **`with` and context managers**

It's common to use `with` when opening files in Python. As we've seen, you can say:

```
with open('myfile.txt', 'w') as f:
    f.write('abc\n')
    f.write('def\n')
```

Most people believe, correctly, that using `with` ensures that the file `f` will be flushed and closed at the end of the block.

But because `with` is overwhelmingly used with files, many developers believe that there is some inherent connection between `with` and files. The truth is that `with` is a much more general Python construct, known as a "context manager."

The basic idea is as follows:

1. You use `with`, along with an object and a variable to which you want to assign the object,
2. The object should know how to behave inside of the context manager,
3. When the block starts, `with` turns to the object. If a *`_enter_`* method is defined on the object, then it runs. In the case of files, the method is defined, but does nothing.
4. When the block ends, `with` once again turns to the object, executing its *`_exit_`* method. This method gives the object a chance to change or restore whatever state it was using.

It's pretty obvious, then, how `with` works with files: Perhaps the *`_enter_`* method isn't important and doesn't do much, but the *`_exit_`* method certainly is important, and does a lot—specifically, in flushing and closing the file.

If you pass two or more objects to `with`, then the *`_enter_`* and *`_exit_`* methods are invoked on each of them, in turn.

Other objects can and do adhere to this protocol. Indeed, if you want, you can write your own classes such that they'll know how to behave inside of a `with` statement. (Details of how to do so are in the "resources" table at the start of the chapter.)

Are context managers only used in the case of files? No, but that's the most common case, by far. Two other common cases are (a) when processing database transactions and (b) when locking certain sections in multi-threaded code. In both situations you want to have a section of code that is executed within a certain context—and thus, Python's context management, via `with`, comes to the rescue.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

5.3 Word count

Unix systems contain many utility functions. One of the most useful to me is `wc` (http://www.computerhope.com/unix/uwc.htm), the "word count" program. If you run `wc` against a text file, it'll count the characters, words, and lines that the file contains.

The challenge for this exercise is to write a version of `wc` in Python. However, your version of `wc` will return four different types of information about the files:

1. Number of characters (including whitespace)
2. Number of words (separated by whitespace)
3. Number of lines
4. Number of unique words (case sensitive, so "NO" is different from "no")

The program should ask the user for the name of an input file, and then produce output for that file.

I have placed a test file (`wcfile.txt`) at https://gist.github.com/reuven/5660728. You may download and use that file in order to test your implementation of `wc`. Any file will do, but if you use this one, your results will match up with mine.

This exercise, like many others in this chapter, tries to help you see the connections between text files and Python's built-in data structures. It's very common to use Python to work with logfiles and configuration files, collecting and reporting that data in a human-readable format.

5.3.1 Solution

```
filename = input("Enter a filename: ")

counts = {'characters':0,
          'words':0,
          'lines':0}
unique_words = set()    ❶

for one_line in open(filename):
    counts['lines'] += 1
    counts['characters'] += len(one_line)
    counts['words'] += len(one_line.split())

    unique_words.update(one_line.split())    ❷

counts['unique words'] = len(unique_words)    ❸
for key, value in counts.items():
    print(f"{key}: {value}")
```

❶ You can create sets with curly braces, but not if they're empty! Use `set()` to create a new, empty set.

❷ `set.update` adds all of the elements of an iterable to a set.

❸ Stick the set's length into `counts` for a combined report

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

You can work through a version of this code in the Python Tutor at https://tinyurl.com/y4fn3qgz

5.3.2 Discussion

This program demonstrates a number of aspects of Python that many programmers use on a daily basis.

First and foremost: Many people who are new to Python believe that if they have to measure four aspects of a file, then they should read through the file four times. That might mean opening the file once and reading through it four times. Or even opening it four separate times.

But it's more common in Python to loop over the file once, iterating over each line and accumulating whatever data it can find from that line.

How will we accumulate this data? We could use separate variables, and there's nothing wrong with that. But I prefer to use a dictionary, since the counts are closely related, and because it also reduces the code I need to produce a report.

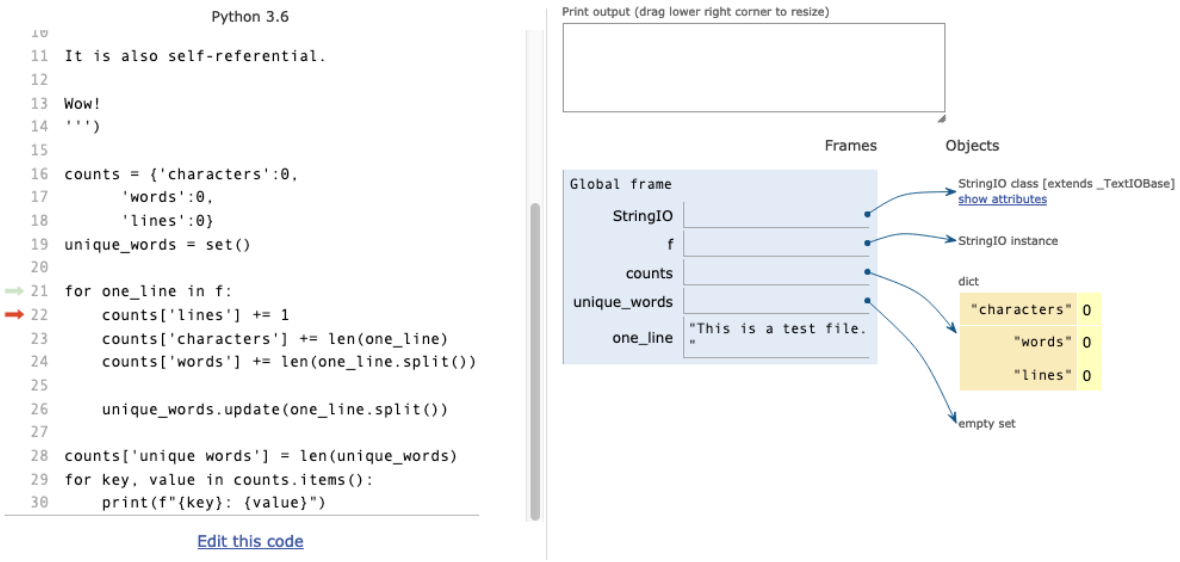


Figure 5.4 Initialized counts in the dict.

So, once we're iterating over the lines of the file, how can we count the various elements?

Counting lines is the easiest part: Each iteration goes over one line, so we can simply add 1 to `counts['lines']` at the top of the loop.

Next, we want to count the number of characters in the file. Since we're already iterating over the file, there's not that much work to do: We get the number of characters in the current line by calculating `len(one_line)`, and then adding that to `counts['characters']`.

Many people are surprised that this includes whitespace characters, such as space and tab, as well as newline. Yes, even an "empty" line contains a single newline character. But if we didn't have newline characters, then it wouldn't be obvious to the computer when it should start a new

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

line. So such characters are necessary, and take up some space.

Next, we want to count the number of words. In order to get this count, we turn `line` into a list of words, invoking `one_line.split`. I invoke `split` without any arguments, which causes it to use all whitespace — spaces, tabs, and newlines — as delimiters. The result is then put into `counts['words']`.

The final item to count is unique words. We could, in theory, use a list to store new words. But it's much easier to let Python do the hard work for us, using a `set` to guarantee the uniqueness. Thus, we create the `unique_words` set at the start of the program, and then use `unique_words.update` (https://docs.python.org/3/library/stdtypes.html?#frozenset.update) to add all of the words in the current line into the set. In order for the report to work on our dictionary, we then add a new key-value pair to `counts`, using `len(unique_words)` in order to count the number of words in the set.

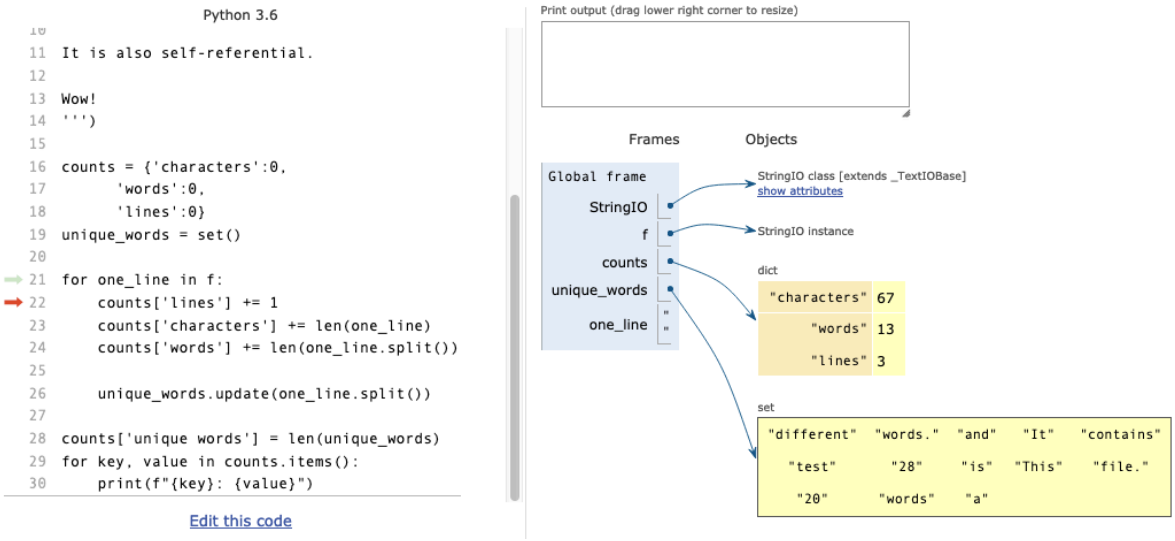


Figure 5.5 The data structures, including unique words, after several lines.

5.3.3 Beyond the exercise

Creating reports based on files is a common use for Python, and using dictionaries to accumulate information from those files is also common. Here are some additional things you can try to do, similar to what we did here:

1. Ask the user to enter the name of a text file, and then (on one line, separated by spaces) words whose frequencies should be counted in that file. Count how many times those words appear in a dictionary, using the user-entered words as the keys, and the counts as the values.
2. Create a dictionary in which the keys are the names of files on your system, and the values are the sizes of those files. To calculate the size, you can use `os.stat` (https://docs.python.org/3/library/os.html?#os.stat).
3. Given a directory, read through each file and count the frequency of each letter. (Force

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

letters to be lowercase, and ignore non-letter characters.) Use a dictionary to keep track of the letter frequencies. What are the five most common letters across all of these files?

5.4 Longest word per file

So far, we have worked with individual files. Many tasks, however, require you to analyze data in multiple files — such as all of the files in a dictionary. This exercise will give you some practice working with multiple files, aggregating measurements across all of them.

In this exercise, the user will provide the name of a directory. You are to find the longest word for each file in the dictionary, returning a dict whose keys are the filenames and whose values are the longest words.

If you don't have any text files that you can use for this exercise, then you can download and use a zipfile I've created from the five most popular books at Project Gutenberg (https://gutenberg.org/). You can download the zipfile from https://gist.github.com/reuven/books.zip .

5.4.1 Solution

```
import os
dirname = input("Enter a directory name: ")


def find_longest_word(filename):
    longest_word = ''
    for one_line in open(filename):
        //GH The following line should be further indented
        for one_word in one_line.split():
            if len(one_word) > len(longest_word):
                //GH The following line should be further indented
                longest_word = one_word
    return longest_word

print({filename : find_longest_word(os.path.join(dirname, filename))    ❶
       for filename in os.listdir(dirname)    ❷
       if os.path.isfile(os.path.join(dirname, filename))})    ❸
```

❶ Get the filename and its full path

❷ Iterate over all of the files in `dirname`

❸ We're only interested in files, not directories or special files

5.4.2 Discussion

In this case, we are being asked to take a directory name, and then find the longest word in each plain-text file in that directory. Our function should return a dictionary in which the dict's keys are the filenames and the dict's values are the longest words in each file.

Whenever you hear that you need to transform a collection of inputs into a collection of outputs,

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

you should immediately think about comprehensions—most commonly, list comprehensions, but set comprehensions and dictionary comprehensions are also useful. In this case, we'll use a dict comprehension — which means that we'll create a dictionary based on iterating over a source. The source, in our case, will be a list of filenames. The filename will also provide the dictionary key, while the value will be the result of passing the filename to a function.

In other words, our dict comprehension will:

1. Iterate over the list of files in the named directory, putting the filename in the variable `filename`.
2. For each file, run the function `find_longest_word`, passing `filename` as an argument. The return value is a string, the longest string in the file.
3. Each filename - longest-word combination becomes a key-value pair in the dictionary we create.

How can we implement `find_longest_word`? We could read the file's entire contents into a string, turn that string into a list, and then find the longest word in the list with `sorted`. While this will work well for short files, it's going to use a lot of memory for even medium-sized files.

My solution is thus to iterate over every line of a file, and then over every word in the line. If we find a word that's longer than the current `longest_word`, then we replace the old word with the new one.

When we're done iterating over the file, we can return the longest word that we found.

Note my use of `os.path.join` (https://docs.python.org/3/library/os.path.html?#os.path.join), to combine the directory name with a filename. You can think of `os.path.join` as a filename-specific version of `str.join`. It has additional advantages, as well, such as taking into account the current operating system. On Windows, `os.path.join` will use backslashes, whereas on Macs and Unix/Linux systems, it'll use a forward slash.

5.4.3 Beyond the exercise

Producing reports about files and file contents using dicts and other basic data structures is commonly done in Python. Here are a few possible exercises to practice these ideas further:

1. Use the `hashlib` module in the Python standard library, and the `md5` function within it, to calculate the MD5 hash for the contents of every file in a user-specified directory. Then print all of the filenames and their MD5 hashes.
2. Ask the user for a directory name. Show all of the files in the directory, as well as how long ago the directory was modified. You will probably want to use a combination of `os.stat` and the "Arrow" package on PyPI (https://pypi.org/project/arrow) to do this easily.
3. Open an HTTP server's logfile. (If you lack one, then you can read one from me, at https://gist.github.com/reuven/5875971 .) Summarize how many requests resulted in numeric response code—202, 304, etc.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

SIDEBAR **Directory listings**

For a language that claims "there's one way to do it," Python has too many ways to list files in a directory. The two most common are `os.listdir` and `glob.glob`, both of which I mentioned in this chapter. A third way is to use `pathlib`, a relatively new addition to Python which provides us with an object-oriented API to the filesystem.

The easiest and most standard of these is `os.listdir`, a function in the `os` module. It returns a list of strings, the names of files in the directory. For example:

```
filenames = os.listdir('/etc/')
```

The good news is that it's easy to understand and work with `os.listdir`. The bad news is that it returns a list of filenames without the directory name, which means that in order to open or work with the files, you'll need to add the directory name at the beginning—ideally with `os.path.join`, which works cross-platform.

The other problem with `os.listdir` is that you cannot filter the filenames by a pattern. You get everything, including subdirectories and hidden files. So if you want all of the `.txt` files in a directory, `os.listdir` won't be enough.

That's where the `glob` module comes in: It lets you use patterns, sometimes known as "globbing," to describe the files that you want. Moreover, it returns a list of strings—with each string containing the complete path to the file. For example, I can get the full paths of the configuration files in `/etc/` on my computer with

```
filenames = glob.glob('/etc/*.conf')
```

I don't need to worry about other files or subdirectories in this case, which makes it much easier to work with. For a long time, `glob.glob` was thus my go-to function for finding files.

But I have recently discovered `pathlib`, a module that comes with the Python standard library and which makes things easier in many ways. You start by creating a `pathlib.Path` object, which represents a file or directory:

```
import pathlib
p = pathlib.Path('/etc/')
```

Once you have this `Path` object, you can do lots of things with it that previously required separate functions—including the ones I've described above. For example, we can get an iterator that returns files in the directory with `iterdir`:

```
for one_filename in p.iterdir():
    print(one_filename)
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

In each iteration, we don't get a string, but rather a `Path` object (or more specifically, on my Mac I get a `PosixPath` object). This allows us to do lots more than just print the filename; we can open it and inspect it with the `Path` object.

If you want to get a list of files matching a pattern, as I did with `glob.glob`, then you can use the `glob` method:

```
for one_filename in p.glob('*.conf'):
    print(one_filename)
```

`pathlib` is a great addition to recent Python versions. If you have a chance to use it, you should do so; I've found that it clarifies and shortens quite a bit of my code.

5.5 Reading and writing CSV

CSV ("comma-separated values") files are extremely common, and especially so in the world of data science. Thus, knowing how to work with them can be handy.

In a CSV file, each record is stored on one line, and fields are separated by commas. CSV is commonly used for exchanging information, especially (but not only) in the world of data science. For example, a CSV file might contain information about different vegetables:

```
lettuce,green,soft
carrot,orange,hard
pepper,green,hard
eggplant,purple,soft
```

Each line in this CSV file contains three fields, separated by commas. There aren't any headers describing the fields, although many CSV files do have them.

Sometimes, the comma is replaced by another character, so as to avoid potential ambiguity; my personal favorite is to use a `TAB` character (`\t` in Python strings).

Python comes with a `csv` module (https://docs.python.org/3/library/csv.html?module-csv) that handles writing to and reading from CSV files. For example, you can write to a CSV file with the following code:

```
import csv

with open('/tmp/stuff.csv', 'w') as f:
    o = csv.writer(f)    ❶
    o.writerow(range(5))    ❷
    o.writerow(['a', 'b', 'c', 'd', 'e'])    ❸
```

❶ Create a `csv.writer` object, wrapping our file-like object `f`

❷ Write the integers from 0-4 to the file, separated by commas

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

❸ Write this list of strings as a record to the CSV file, separated by commas

Not all CSV files necessarily look like CSV files. For example, the standard Unix `/etc/passwd` file, which contains information about users on a system (but no longer users' passwords, despite its name), separates fields with `:` characters.

For this exercise, create a program that creates a subset of `/etc/passwd` in a new file. You should read from `/etc/passwd` and produce a new file whose contents are the username (index 0) and the user ID (index 2). Note that a record may contain a comment, in which case it will not have anything at index 2; you should take that into consideration when writing the file. The output file should use `TAB` characters to separate the elements.

Thus, the input will look like:

```
root:*:0:0::0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1::0:0:System Services:/var/root:/usr/bin/false
# I am a comment line
_ftp:*:98:-2::0:0:FTP Daemon:/var/empty:/usr/bin/false
```

and the output will look like:

```
root    0
daemon  1
_ftp    98
```

Notice that the comment line in the input file is not placed in the output file. You can assume that any line with at least 2 colon-separated fields is legitimate.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

NOTE Different operating systems have different ways of indicating that we have reached the end of the line. Unix systems, including the Mac, use ASCII 10 ("line feed," or "LF"). Windows systems use two characters, namely ASCII 13 ("carriage return," or "CR") + ASCII 10. Old-style Macs used just ASCII 13.

Python tries to bridge these gaps by being flexible, and making some good guesses, when it reads files. I've thus rarely had problems using Python to read text files that were created using Windows. By the same token, my students (who typically use Windows) generally have no problem reading the files that I created on the Mac. Python figures out what line ending is being used, so we don't need to provide any more hints. And inside of the Python program, the line ending is symbolized by `\n`.

Writing to files, by contrast, is a bit trickier: Python will try to use the line ending appropriate for that operating system. So if you're writing to a file on Windows, it'll use CR+LF (sometimes shown as `\r\n`). If you're writing to a file on a Unix machine, then it'll just use LF.

This typically works just fine. But sometimes, you'll find yourself seeing too many or too few newlines when you read from a file. This might mean that Python has guessed incorrectly, or that the file uses a few different line endings, confusing Python's guessing algorithm.

In such cases, you can pass a value to the `newline` parameter in the `open` function, used to open files. You can try to explicitly use `newline='\n'` to force Unix-style newlines, or `newline='\r\n'` to force Windows-style newlines. If this doesn't fix the problem, then you might need to examine the file further, to see how it was defined.

5.5.1 Solution

```
import csv

with open('/etc/passwd') as passwd, open('/tmp/output.csv', 'w') as output:
    r = csv.reader(passwd, delimiter=':')    ❶
    w = csv.writer(output, delimiter='\t')    ❷
    for record in r:
        if len(record) > 1:
            w.writerow((record[0], record[2]))
```

❶ Fields in the input file are separated by colons (`:`)

❷ Fields in the output file are separated by tabs (`\t`)

5.5.2 Discussion

The above program uses a number of aspects of Python that are useful when working with files.

We have already seen and discussed `with` earlier in this chapter. Here, you can see how `with`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

can be used to open two separate files, or generally to define any number of objects. As soon as our block exits, both of the files are automatically closed.

We define two variables in the `with` statement, for the two files with which we will be working: The `passwd` file is opened for reading from `/etc/passwd`. The `output` file is opened for writing, and writes to `/tmp/output.csv`. Our program will act as a go-between, translating from the input file and placing a reformatted subset into the output file.

We do this by creating one instance of `csv.reader`, which wraps `passwd`. However, because `/etc/passwd` uses colons (`:`) to delimit fields, we must tell this to `csv.reader`. Otherwise, it will try to use commas, which is almost certainly not going to work well. Similarly, we define an instance of `csv.writer`, wrapping our `output` file, and indicating that we want to use `TAB` as the delimiter.

Now that we have our objects in place for reading and writing CSV data, we can run through the input file, writing a row (line) to the output file for each of those inputs. We take the username (from index 0) and the user ID (from index 2), create a tuple, and pass that tuple to `csv.writerow`. Our `csv.writer` object knows how to take our fields, and print them to the file, separated by `\t`.

Perhaps the trickiest thing here is to ensure that we do not try to transform lines which contain comments—that is, those which begin with a hash (`#`) character. There are a number of ways to do this, but the method that I have employed here is simply to check the number of fields we got for the current input line. If there is only one field, then it must be a comment line, or perhaps another type of malformed line. In such a case, we ignore the line altogether. Another good technique would be to check for `\#` at the start of the line, perhaps using `str.startswith`.

5.5.3 Beyond the exercise

CSV files are extremely useful and common, and the `csv` module that comes with Python works with them very well. If you need something more advanced, then you might want to look into `pandas` (http://pandas.pydata.org/), which handles a wide array of CSV variations, as well as many other formats.

1. Extend this exercise by asking the user to enter a space-separated list of integers, indicating which fields should be written to the output CSV file. Also ask the user which character should be used as a delimiter in the output file. Then read from `/etc/passwd`, writing the user's chosen fields, separated by the user's chosen delimiter.
2. Write a dictionary to a CSV file. Each line in the CSV file should be: (1) the key, which we'll assume to be a string, (2) the value, (3) the type of the value (e.g., `str` or `int`).
3. Create a CSV file, in which each line contains 10 random integers between 10 and 100. Now read the file back, and print the sum and mean of the numbers on each line.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

5.6 JSON

JSON ("JavaScript Object Notation," described at http://json.org/) is a popular format for data exchange. In particular, many Web services and APIs send and receive data using JSON.

JSON-encoded data can be read into a very large number of programming languages, including Python. The Python standard library comes with the `json` module (https://docs.python.org/3/library/json.html?#module-json), which can be used to turn JSON-encoded strings into Python objects, and vice versa. The `json.load` method reads a JSON-encoded string from a file, and returns a combination of Python objects.

In this exercise, you are analyzing test data in a high school. There is a `scores` directory on the filesystem containing a number of files in JSON format. Each file represents the scores for one class. Thus, if we are trying to analyze the scores from class 9a, the scores would be in a file called `9a.json` that looks like this:

```
[{"math" : 90, "literature" : 98, "science" : 97},
 {"math" : 65, "literature" : 79, "science" : 85},
 {"math" : 78, "literature" : 83, "science" : 75},
 {"math" : 92, "literature" : 78, "science" : 85},
 {"math" : 100, "literature" : 80, "science" : 90}
]
```

The directory may also contain files for 10th grade (`10a.json`, `10b.json`, and `10c.json`), and other grades and classes in the high school. Each file file contains the JSON equivalent of a list of dicts, with each dict containing scores for several different school subjects.

> **NOTE** Valid JSON uses double quotes (`"`), not single quotes (`'`). This can be surprising and frustrating for Python developers to discover.

For this exercise, you must produce the highest, lowest, and average test scores for each subject in each class. Given two files (`9a.json` and `9b.json`) in the `scores` directory, we would see the following output:

```
scores/9a.json
    science: min 75, max 97, average 86.4
    literature: min 78, max 98, average 83.6
    math: min 65, max 100, average 85.0
scores/9b.json
    science: min 35, max 95, average 82.0
    literature: min 38, max 98, average 72.0
    math: min 38, max 100, average 77.0
```

You can download a zipfile with these JSON files from https://gist.github.com/reuven/SOMETHING

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

5.6.1 Solution

```
import json
import glob

scores = { }

for filename in glob.glob("scores/*.json"):
    scores[filename] = { }

    f = open(filename)
    for result in json.load(f):    ❶
        //GH The following few lines should be further indented
        for subject, score in result.items():
            scores[filename].setdefault(subject, [])    ❷
            scores[filename][subject].append(score)

for one_class in scores:    ❸
    print(one_class)
    for subject, subject_scores in scores[one_class].items():
        //GH The following few lines should be further indented
        min_score = min(subject_scores)
        max_score = max(subject_scores)
        average_score = float(sum(subject_scores)) / len(subject_scores)

        //GH We're missing the 'f' below to make it an f-string.
        print("\t{subject}: min {min_score}, max {max_score}, average {average_score}")
```

❶ Read from the file `f`, and turn it from JSON into Python objects

❷ Make sure that `subject` exists as a key in `scores[filename]`

❸ Summarize the scores

5.6.2 Discussion

In many languages, the first response to this kind of problem would be: Let's create our own class! But in Python, while we can (and often do) create our own classes, it's often easier and faster to make use of built-in data structures—lists, tuples, and dicts.

In this particular case, we are reading from a JSON file. JSON is a data representation, much like XML; it isn't a data type per se. Thus, if we want to create JSON, we must use the `json` module to turn our Python data into JSON-formatted strings. And if we want to read from a JSON file, we must read the contents of the file, as strings, into our program, and then turn it into Python data structures.

In this exercise, though, you are being asked to work on multiple files in one directory. We know that the directory is called `scores` and that the files all have a `.json` suffix. We could thus use `os.listdir` on the directory, filtering (perhaps with a list comprehension) through all of those filenames such that we only work on those ending with `.json`.

However, this seems like a more appropriate place to use `glob` (https://docs.python.org/3/library/glob.html#glob.glob), which takes a Unix-style filename pattern with (among others) `*` and `?` characters, and returns a list of those filenames matching the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

pattern. Thus, by invoking `glob.glob('scores/*.json')`, we get all of the files ending in `.json` within the `scores` directory. We can then iterate over that list, assigning the current filename (a string) to `filename`.

Next, we create a new entry in our `scores` dictionary, which is where we will store the scores. This will actually be a dictionary of dictionaries, in which the first level will be the name of the file—and thus the class—from which we have read the data. The second-level keys will be the subjects; its values will be a list of scores, from which we can then calculate the statistics we need. Thus, once we have defined `filename`, we immediately add the filename as a key to `scores`, with a new, empty dictionary as the value.

Sometimes, you will need to read each line of a file into Python, and then invoke `json.loads` to turn that line into data. In our case, however, the file contains a single JSON array. We must thus use `json.load` to read from the file object `f`, which turns the contents of the file into a Python list of dictionaries.

Because `json.parse` returns a list of dicts, we can iterate over it. Each test result is placed in the `result` variable, which is a dictionary, in which the keys are the subjects and the values are the scores. Our goal is to reveal some statistics for each of the subjects in the class, which means that while the input file reports scores on a per-student basis, our report will ignore the students, in favor of the subjects.

Given that `result` is a dict, we can iterate over its key-value pairs with `result.items()`, using parallel assignment to iterate over the key and value (here called `subject` and `score`). Now, we don't know in advance what subjects will be in our file, nor do we know how many tests there will be. Thus, it is easiest for us to store our scores in a list. This means that our `scores` dict will have one top-level key for each filename, and a one second-level key for each subject. The second-level value will be a list, to which we will then append with each iteration through the JSON-parsed list.

Thus, we will want to add our score to the list

```
scores[filename][subject]
```

Before we can do that, we need to make sure that the list exists. One easy way to do this is with `dict.setdefault`, which assigns a key-value pair to a dictionary, but only if the key does not already exist. In other words, `d.setdefault(k, v)` is the same as saying:

```
if k not in d:
    d[k] = v
```

We use `dict.setdefault` (https://docs.python.org/3/library/stdtypes.html?#dict.setdefault) to create the list if it doesn't yet exist, and then in the next line, we add the score to the list for this subject, in this class.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

When we have completed our initial `for` loop, we have all of the scores for each class. We can now iterate over each class, printing the name of the class.

Then, we iterate over each subject for the class. We once again use the method `dict.items` to return a key-value pair—in this case, calling them `subject` (for the name of the class) and `subject_scores` (for the list of scores for that subject).

Now, we use an f-string to produce some output, using the built-in `min` (https://docs.python.org/3/library/functions.html?#min) and `max` (https://docs.python.org/3/library/functions.html?#max) functions, and then combining `sum` (https://docs.python.org/3/library/functions.html?#sum) and `len` to get the average score.

While this program read from a file containing JSON and then produced output on the user's screen, it could just as easily have read from a network connection containing JSON, and/or written to a file or socket in JSON format. So long as we use built-in and standard Python data structures, the `json` module will be able to take our data and turn it into JSON.

5.6.3 Beyond the exercise

Here are some more tasks you can try that use JSON:

1. Convert `/etc/passwd` from a CSV-style file into a JSON-formatted file. The JSON will contain the equivalent of a list of Python tuples, with each tuple representing one line from the file.
2. For a slightly different challenge, turn each line in the file into a Python dict. This will require identifying each field with a unique column or key name; if you're not sure what each field in `/etc/passwd` does, then you can give it an arbitrary name.
3. Ask the user for the name of a directory. Iterate through each file in that directory (ignoring subdirectories), getting (via `os.stat`) the size of the file and when they were last modified. Create a JSON-formatted file on disk listing each filename, size, and modification timestamp. Then read the file back in, and identify which files were modified most and least recently in that directory, and which files are largest and smallest in that directory.

5.7 Reverse lines

In many cases, we want to take a file in one format and save it to another format. This exercise tries to do a simple version of such transformation: You are to create a new text file whose contents are identical to an existing text file, but with each line's content reversed. For example, if a file looks like

```
abc def
ghi jkl
```

then the output file will be

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

```
fed cba
lkj ihg
```

Notice that the newline remains at the end of the string, while the rest of the characters are all reversed.

Transforming files from one format into another, or taking data from one file and creating another one based on it, are common tasks. For example, you might need to translate dates to a different format, or move timestamps from Eastern Daylight Time into Greenwich Mean Time, or transform prices from euros into dollars. You might also want to extract only some data from an input file, such as for a particular date or location.

5.7.1 Solution

```
infilename = 'input.txt'
outfilename = 'output.txt'

with open(infilename) as infile, open(outfilename, 'w') as outfile:
    for one_line in infile:
        //GH The next line should be further indented. Also, has 'rstrip' been mentioned before?
          Perhaps it should be treated in the same way as my previous comment about
          startswith and strip.
        outfile.write(f"{one_line.rstrip()[::-1]}\n")    ❶
```

❶ `str.rstrip` removes all whitespace from the right side of a string

5.7.2 Discussion

This solution depends not only on the fact that we can iterate over a file one line at a time, but also that we can work with more than one object in a `with` statement. Remember that `with` takes one or more objects, and allows us to assign variables to those objects. I particularly like the fact that when I want to read from one file and write to another, I can just use `with` to open one for reading, open a second for writing, and then do what I've shown here.

I then read through each line of the input file. I then reverse the line using Python's slice syntax—remember that `s[::-1]` means that we want all of the elements of `s`, from the start to the end, but with a step size of -1, which returns a reversed version of the string. But before we can reverse the string, we first want to remove the newline character that is the final character in the string. So we first run `str.rstrip()` on the current line, and then we reverse it. We then write it to the output file, adding a newline character so that we will actually descend by one line.

The use of `with` guarantees that both files will be closed when the block ends. When we close a file that we opened for writing, it is automatically flushed, which means that we don't need to worry about whether the data has actually been saved to disk.

I should note that people often ask me how to read from and write to the same file. Python does support that, with the `r+` mode. But I find that this opens the door to many potential problems,

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

because of the chance that you'll overwrite the wrong character, and thus mess up the format of the file you're editing. I suggest that people use this sort of read-from-one, write-to-the-other code, which has roughly the same effect, without the potential danger of messing up the input file.

5.7.3 Beyond the exercise

Here are some more exercise ideas for translating files from one format to another using `with` and this kind of technique:

1. "Encrypt" a text file by turning all of its characters into their numeric equivalents (with the built-in `ord` function), and writing that file to disk. Now "decrypt" the file (using the built-in `chr` function), turning the numbers back into their original characters.
2. Given an existing text file, create two new text files. The new files will each contain the same number of lines as the input file. In one, you'll write all of the vowels (a, e, i, o, and u) from the input file. In the other output file, you'll write all of the consonants. (You can ignore punctuation and whitespace.)
3. The final field in `/etc/passwd` is the "shell," the Unix command interpreter that is invoked when a user logs in. Create a file containing one line per shell, in which the shell's name is written, followed by all of the usernames that use the shell. For example:

```
/bin/bash:root, jci, user, reuven, atara
/bin/sh:spamd, gitlab
```

5.8 Conclusion

It's almost impossible to imagine writing programs without using files. And while there are many different types of files, Python is especially well suited for working with text files—especially, but not only, including logfiles and configuration files, as well those formatted in such standard ways as JSON and CSV.

It's important to remember a few things when working with files:

1. You will typically open files for either reading or writing
2. You can (and should) iterate over files one line at a time, rather than reading the whole thing into memory at once
3. Using `with` when opening a file for writing ensures that the file will be flushed and closed
4. The `csv` module makes it easy to read from and write to CSV files
5. The `json` module's `dump` and `load` functions allow us to move between Python data structures and JSON-formatted strings
6. Reading from files into builtin Python data types is a common and powerful technique

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

6 Functions

Functions are one of the cornerstones of programming—but not because there is a technical need for them. We could program without functions, if we really had to do. But functions provide a number of great benefits.

First, they allow us to avoid repetition in our code. Many programs have instructions that are repeated: Asking a user to log in, reading data from a particular type of configuration file, or calculating the length of an MP3. While the computer won't mind (or even complain) if the same code appears in multiple places, we — and the people who have to maintain the code after we're done with it — will suffer, and likely complain. Such repetition is hard to remember and keep track of. Moreover, you'll likely find that the code needs improvement and maintenance; if it occurs multiple times in your program, then you'll need to find and fix it each of those times.

The term "don't repeat yourself," often abbreviated as DRY, is a good thing to keep in mind when programming. And writing functions is a great way to apply the phrase, "DRY up your code."

A second benefit of functions is that they let us (as developers) think at a higher level of abstraction. Just as you can't drive if you're constantly thinking about all of the parts of your car and what you're doing, you can't program if you're constantly thinking about all of the parts of your program and what they're doing. It helps, semantically and cognitively, to wrap functionality into a named package, and then to use that name.

In natural language, we create new verbs all of the time, such as "programming" and "texting." We don't have to do this; we could describe the action using many more words, and with much more detail. But doing so becomes tedious, and draws attention away from the point that we're making. Functions are the verbs of programming; they let us define new actions based on old ones, and thus let us think in more sophisticated terms.

For all of these reasons, functions are a useful tool, and available in all programming languages.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

But Python's functions add a twist to this: They are objects, meaning that they can be treated as data. We can store functions in data structures, and retrieve from them there, as well. Using functions in this way seems odd to many newcomers to Python, but it provides a powerful technique that can reduce how much code we write and increase our flexibility.

Moreover, Python doesn't allow for multiple definitions of the same function. In some languages, you can define a function multiple times, each time having a different signature. So you could, for example, define the function once taking a single string argument, a second time taking a list argument, a third time taking a dict argument, and a fourth time taking three float arguments.

In Python, this functionality doesn't exist: When you define a function, you're assigning to a variable. And just as you cannot expect that `x` will simultaneously contain the values 5 and 7, you similarly cannot expect that a function will contain multiple implementations.

The way that we get around this problem in Python is with flexible parameters: Between default values, variable numbers are of arguments (`*args`), and keyword arguments (`**kwargs`), we can write functions that handle a variety of situations.

This chapter contains exercises that prod you to use all of these techniques, so that you can write better, more powerful, and more flexible functions in your own work. This will not only allow you to write code once and use it numerous times, but also to build up a hierarchy of new verbs, describing increasingly complex and higher-level tasks.

Table 6.1 Reference table

Name	Description	Example	Link
Functions	Python documentation	`def double(x): return x * 2`	#function-definitions
`global`	Meaning and usage of `global`	`global x`	#global
`nonlocal`	Meaning and usage of `nonlocal`	`nonlocal x`	#nonlocal
`operator` module	collection of methods that implement built-in operators	`operator.add(2,4)`	docs.python.org/3/library/operator

SIDEBAR Default parameter values

Let's say that I can write a simple function that returns a friendly greeting:

```
def hello(name):
    return f'Hello, {name}!'
```

This will work fine, if I provide a value for `name`. But what if I don't?

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

```
>>> hello('world')
'Hello, world!'

>>> hello()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: hello() missing 1 required positional argument: 'name'
```

In other words: Python knows that the function takes a single argument. So if you call the function with one argument, you're just fine. Call it with no arguments (or with two arguments, for that matter), and you'll get an error message.

How does Python know how many arguments the function should take? Because the function object, which we created when we defined the function with `def`, keeps track of that sort of thing. Instead of invoking the function, we can look inside of the function object. The `__code__` attribute contains the core of the function, including the bytecodes into which your function was compiled. Inside of that object are a number of hints that Python keeps around, including this one:

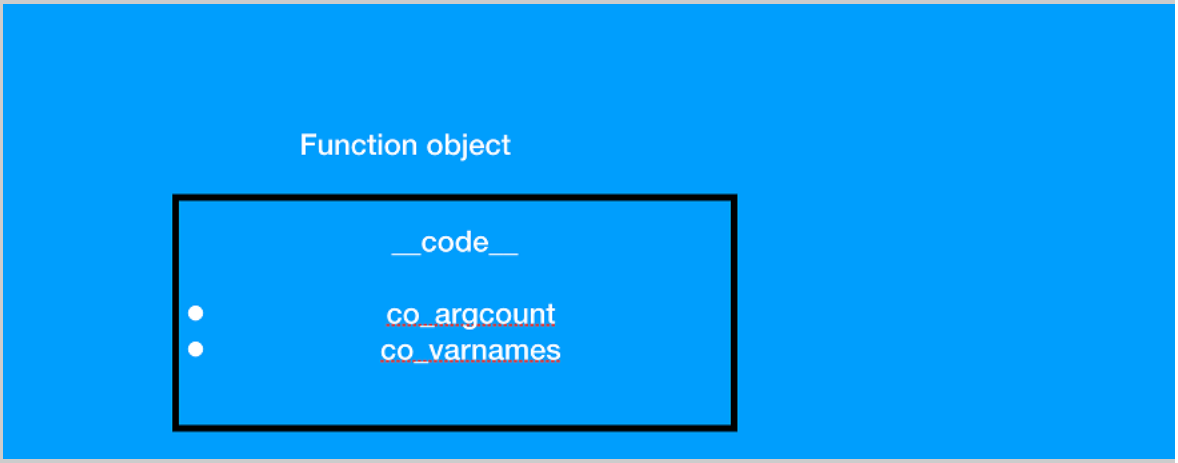


Figure 6.1 A function object, along with its __code__ section

```
>>> hello.__code__.co_argcount
1
```

In other words: When we define our function with a parameter, the function object keeps track of that in `co_argcount`. And when we invoke the function, Python compares the number of arguments with `co_argcount`. If there's a mismatch, then we get an error, as we saw above.

However, there is still a way that we can define the function such that an argument is optional: We can add a default value to the parameter:

```
def hello(name='world'):
    return f'Hello, {name}!'
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

When we run the function now, Python gives us more slack: If we pass an argument, then that value is assigned to the `name` parameter. But if we don't pass an argument, then the string `world` is assigned to `name`, as per our default. In this way, we can call our function with either no arguments or one argument. Two arguments aren't allowed, though.

Table 6.2 Calling hello

Call	Value of `name`	Return value
`hello()`	`world`, thanks to the default	`Hello, world!`
`hello('out there')`	`out there`	`Hello, out there!`
`hello('a', 'b')`	Error: Too many arguments	No return value

NOTE Parameters with defaults must come after those without defaults.

WARNING Never use a mutable value, such as a list of dictionary, as a parameter's default value. This is because default values are stored and reused across calls to the function. This means that if you modify the default value in one call, that modification will be visible in the next call. Most code checkers and IDEs will warn you about this, but it's important to keep in mind.

6.1 XML generator

Python is often used not just to parse data, but to format it, as well. In this exercise, you'll write a function that uses a combination of different parameters and parameter types to produce a variety of outputs.

You are to Write a function, `myxml`, that allows you to create simple XML output. The output from the function will always be a string.

The function can be invoked in a number of ways:

Table 6.3 Calling myxml

Call	Return value
`myxml('foo')`	`<foo></foo>`
`myxml('foo', 'bar')`	`<foo>bar</foo>`
`myxml('foo', 'bar', a=1, b=2, c=3)`	`<foo a="1" b="2" c="3">bar</foo>`

Notice that in all cases, the first argument is the name of the tag. In the latter two cases, the second argument is the content (text) placed between the opening and closing tags. And in the third case, the name-value pairs will be turned into attributes inside of the opening tag.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

6.1.1 Solution

```
def myxml(tagname, content='', **kwargs):     ❶
    attrs = ''.join([f' {key}="{value}"'
                     for key, value in kwargs.items()])     ❷
    return f"<{tagname}{attrs}>{content}</{tagname}>"     ❸
```

❶ The function has one mandatory parameter, one with a default, and `**kwargs`

❷ Use a list comprehension to create a string from `kwargs`

❸ Return the XML-formatted string

You can work through a version of this code in the Python Tutor at https://tinyurl.com/y4ef7pbt

6.1.2 Discussion

Let's start by assuming that we only want our function to take a single argument, the name of the tag. That would be easy to write; we could say

```
def myxml(tagname):
    return f"<{tagname}></{tagname}>"
```

But of course, this will fail when we want to pass the second (optional) argument. Some people thus assume that our function should take `*args`, meaning any number of arguments, all of which will be put in a tuple. But as a general rule, `*args` is meant for situations in which you don't know how many values you'll be getting, and you can accept any number.

> **NOTE** My general rule with `*args` is that it should be used when you'll put its value into a `for` loop, and that if you're grabbing elements from `*args` with numeric indexes, then you're probably doing something wrong.

The other option, though, is to use a default. And that's what I've gone with: The first parameter is mandatory, but the second is optional. If I make the second one (which I call `content` here) an empty string, then I know that either the user passes content or the content is empty. In either case, the function works. I can thus define it as follows:

```
def myxml(tagname, content=''):
    return f"<{tagname}>{content}</{tagname}>"
```

But what about the key-value pairs that we can pass, and which are then placed as attributes in the opening tag?

When we define a function with `**kwargs`, we are effectively telling Python that we might pass **any** name-value pair in the style `name=value`. These arguments aren't passed in the normal way,

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

but are treated separately, as "keyword arguments." Those are used to create a dictionary, traditionally called `kwargs`, whose keys are the keyword names and the values are the keyword values. Thus, we can say:

```
def myxml(tagname, content='', **kwargs):
    attrs = ''.join([f' {key}="{value}"'
                     for key, value in kwargs.items()])
    return f"<{tagname}{attrs}>{content}</{tagname}>"
```

As you can see, I'm not just taking the key-value pairs from `**kwargs` and putting them into a string. I first have to take that dict and turn it into name-value pairs in XML format. I do this with a list comprehension, running on the dict. For each key-value pair, I create a string, making sure that the first character in the string is a space—to make sure that we don't bump up against the tagname in the opening tag.

There's a lot going on in these lines of code. and it uses a few common Python paradigms. So it's probably useful to go through it, step by step, just to make things clearer:

1. In the body of `myxml`, we know that `tagname` will be a string (the name of the tag), `content` will be a string (whatever content should go between the tags), and `kwargs` will be a dict (with the attribute name-value pairs).
2. Both `content` and `kwargs` might be empty, if the user didn't pass any values for those parameters.
3. We use a list comprehension to iterate over `kwargs.items`. This will provide us with one key-value pair in each iteration.
4. We use the key-value pair, assigned to the variables `key` and `value`, to create a string of the form `key="value"`. We get one such string for each of the attribute key-value pairs passed by the user.
5. The result of our list comprehension is a list of strings. We join these strings together with `str.join`, with an empty string between the elements.
6. Finally, we return the combination of the opening tag (with any attributes we might have gotten), the content, and the closing tag.

6.1.3 Beyond the exercise

Learning to work with functions, and the types of parameters that you can define, takes some time, but is well worthwhile. Here are some exercises you can use to sharpen your thinking when it comes to function parameters:

1. Write a `copyfile` function that takes one mandatory argument, the name of an input file, and any number of additional arguments, the names of files to which the input should be copied. Calling copyfile('myfile.txt', 'copy1.txt', 'copy2.txt', 'copy3.txt') will create three copies of `myfile.txt`, in `copy1.txt`, `copy2.txt`, and `copy3.txt`.
2. Write a "factorial" function that takes any number of numeric arguments, and returns the result of multiplying them all by one another.
3. Write a `anyjoin` function that works similarly to `str.join`, except that the first argument is a sequence of any types (not just of strings), and the second argument is the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

"glue" that we put between elements, defaulting to " " (a space). So `anyjoin([1,2,3])` will return `1 2 3`, and `anyjoin('abc', '**')` will return `a**b**c`.

SIDEBAR **Variable scoping in Python**

Variable scoping is one of those topics that many people ignore—first because it's dry, and then because it's obvious. The thing is, Python's scoping is very different from what I've seen in other languages. Moreover, it explains a great deal about how the language works, and why certain decisions were made.

The term "scoping" refers to the visibility of variables (and all names) from within the program. If I set a variable's value within a function, have I affected it outside of the function, as well? What if I set a variable's value inside of a `for` loop?

Python has four levels of scoping: Local, enclosing function, global, and builtins, often known by the abbreviation LEGB. They are searched in that order whenever an identifier is referenced. Actually, that's not true; all four are only searched if you're inside of a function body. If you're outside of a function, then Python only searches through the final two namespaces—globals and builtins.

That's an important consideration to keep in mind: **If you haven't defined a function, then you're operating at the global level.** Indentation might be pervasive in Python, but it doesn't affect variable scoping at all.

But what if you run `int('s')`? Is `int` a global variable? No, it is in the "builtins" namespace. Python has very few reserved words; many of the most common types and functions we run are neither globals nor reserved keywords. Rather, they are in the "builtins" namespace, searched after the global one, if a name doesn't exist in the global namespace.

What if you define a global name that's identical to one in builtins? Then you have effectively "shadowed" that value. I see this all the time in my courses, when people write something like:

```
sum = 0
for i in range(5):
    sum += i
print(sum)

print(sum([10, 20, 30]))

TypeError: 'int' object is not callable
```

Why do we get this weird error? Because in addition to the `sum` function defined in "builtins", we have now defined a global variable named `sum`. And because globals come before builtins in Python's search path, Python discovers that `sum` is an integer, and refuses to invoke it.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

It's a bit frustrating that the language doesn't bother to check or warn you about redefining words in builtins. However, there are tools (e.g., PyLint) that will tell you if you've accidentally (or not) created a clashing name.

Local variables

If I define a variable inside of a function, then it's considered to be a "local" variable. Local variables exist only so long as the function does; when the function goes away, so do the local variables it defined. For example:

```
x = 100

def foo():
    x = 200
    print(x)

print(x)
foo()
print(x)
```

The above will print 100, 200, and then 100 again.

In the above code, we have defined two variables: `x` in the global scope is defined to be 100, and never changes. `x` in the local scope, available only within the function `foo`, is 200, and never changes. The fact that both are called `x` doesn't confuse Python, because from within the function, it'll see the local `x` and ignore the global one entirely.

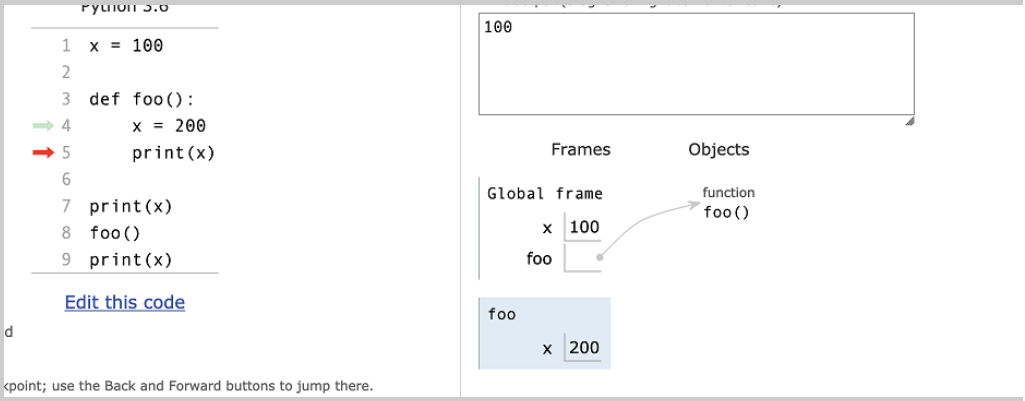


Figure 6.2 Inner vs. outer x

The `global` statement

What if, from within the function, I want to change the global variable? That requires the use of the `global` declaration, which tells Python that you're not interested in creating a local variable in this function. Rather, any retrievals or assignments should affect the global variable. For example:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

```
x = 100

def foo():
    global x
    x = 200
    print(x)

print(x)
foo()
print(x)
```

The above will print 100, 200, and then 200—because there is only one `x`, thanks to the `global` declaration.

Now, changing global variables from within a function is almost always a bad idea. And yet, there are rare times when it's necessary. For example, you might need to update a configuration parameter that's set as a global variable.

Enclosing

Finally, let's consider inner functions via the following code:

```
def foo(x):
    def bar(y):
        return x * y
    return f

bar = foo(10)
print(f(20))
```

Already, this code seems a bit weird. What are we doing, defining `bar` inside of `foo`? This inner function, sometimes known as a "closure," is a function that's defined when `foo` is executed. Indeed, every time that we run `foo`, we get a new function named `bar` back. But of course, the name `bar` is a local variable inside of `foo`; we can call the returned function whatever we want.

When we run the above code, the result is 200. It makes sense that when we invoke `f`, we're executing `bar`, which was returned by `foo`. And we can understand how `bar` has access to `y`, since it's a local variable. But what about `x`? How does the function `bar` have access to `x`, a local variable in `foo`?

The answer, of course, is LEGB:

1. First, Python looks for `x` locally, in the local function `bar`
2. Next, Python looks for `x` in the enclosing function `foo`
3. If `x` were not in `foo`, then Python would continue looking at the global level
4. And if `x` were not a global variable, then Python would look in the `builtins` namespace

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

What if I want to change the value of `x`, a local variable in the enclosing function? It's not global, so the `global` declaration won't work. In Python 3, though, we have the `nonlocal` keyword, which tells Python: Any assignment we do to this variable should go to the outer function, not to a (new) local variable.

For example:

```
def foo():
    call_counter = 0    ❶
    def bar(y):
        nonlocal call_counter    ❷
        call_counter += 1    ❸
        return f'y = {y}, call_counter = {call_counter}'
    return bar

b = foo()
for i in range(10, 100, 10):    ❹
    print(b(i))    ❺
```

❶ Initialize `call_counter` as a local variable in `foo`

❷ Tell `bar` that assignments to `call_counter` should affect the enclosing variable in `foo`

❸ Increment `call_counter`, whose value sticks around across runs of `bar`

❹ Iterate over the numbers 10, 20, 30, … 100

❺ Call `b` with each of the numbers in that range

The output from the above program is:

```
y = 10, call_counter = 1
y = 20, call_counter = 2
y = 30, call_counter = 3
y = 40, call_counter = 4
y = 50, call_counter = 5
y = 60, call_counter = 6
y = 70, call_counter = 7
y = 80, call_counter = 8
y = 90, call_counter = 9
```

So any time you see Python accessing or setting a variable—which is often!—consider the "LEGB" scoping rule, and how it is always, without any exception, used to find all identifiers, including data, functions, classes, and modules.

6.2 Prefix notation calculator

In Python, as in real life, We normally write mathematics using "infix" notation, as in `2+3`. But there is also something known as "prefix notation," in which the operator precedes the arguments. (There is also "postfix notation," sometimes known as "reverse Polish notation," which is still in use on HP brand calculators.)

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

Prefix and postfix notation are both useful, in that they allow us to do sophisticated operations without parentheses. For example, if you write `2 3 4 + *` in RPN, you're telling the system to first add `3+4` and then multiply `2*7`. This is why HP calculators have an `enter` key but no = key, which confuses newcomers greatly. I've heard that back before calculators commonly had parentheses, RPN was a huge competitive advantage. And in the Lisp programming language, prefix notation allows you to apply an operator to many numbers (e.g., `(+ 1 2 3 4 5)`) rather than get caught up with lots of + signs.

For this exercise, I want you to write a program that asks the user to enter a simple math expression in prefix notation, with an operator and two numbers. Your program will parse the input and produce the appropriate output. For our purposes, it's enough to handle the six basic arithmetic operations in Python—addition, subtraction, multiplication, division (/), modulus (%), and exponentiation (**). The normal Python math rules should work, such that division always results in a floating-point number. We will assume, for our purposes, that the user will enter one of our six operators and two valid numbers, albeit all in one line and one input string.

But wait, there's a catch — or a hint, if you prefer: You should implement each of the operations as a separate function, and you shouldn't use an `if` statement to decide which function should be run.

6.2.1 Solution

```
import operator    ❶
operations = {'+' : operator.add,    ❷
              '-' : operator.sub,
              '*' : operator.mul,
              '/' : operator.truediv,    ❸
              '**' :operator.pow,
              '%' : operator.mod}

to_solve = input("Enter a two-operand math problem, with prefix notation: ")

operator, first_s, second_s = to_solve.split()    ❹
first = int(first_s)
second = int(second_s)

print(operations[operator](first, second))    ❺
```

❶ The `operator` module provides functions that implement all builtin operators

❷ Yes, functions can be the values in a dict!

❸ You can choose between `truediv` that returns a float, as with the `/` operator, or `floordiv` that returns an integer, as with the `//` operator.

❹ We split the line, assigning via unpacking

❺ We call the function retrieved via `operator`, passing `first` and `second` as arguments

You can work through a version of this code in the Python Tutor at https://tinyurl.com/y5bj2y4v

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

6.2.2 Discussion

The above solution uses a technique known as a "dispatch table," along with the "operator" module that comes with Python. It's my favorite solution to this problem, but it's not the only one—and it's likely not the one that you first thought of.

Let's start with the simplest solution, and work our way up to what I wrote above. We will need a function for each of the operators. But then we'll somehow need to translate from the operator string (e.g., `+` or `**`) to the function we want to run. We could use `if` statements to make such a decision, but a more common way to do this in Python is with dictionaries. After all, it's pretty standard to have keys that are strings, and since we can store anything in the value, this includes functions.

> **NOTE** Many of my students ask me how to create a `switch-case` statement in Python. They are surprised to hear that they already know the answer, namely that Python doesn't have such a statement, and that we use `if` instead. This is part of Python's philosophy of having one, and only one way to do something. It reduces programmers' choices, but makes the code clearer and easier to maintain.

We can then retrieve the function from the dict and invoke it with parentheses:

```
def add(a,b):
    return a + b

def sub(a,b):
    return a - b

def mul(a,b):
    return a * b

def div(a,b):
    return a / b

def pow(a,b):
    return a ** b

def mod(a,b):
    return a % b

operations = {'+' : add,      ❶
              '-' : sub,
              '*' : mul,
              '/' : div,
              '**' : pow,
              '%' : mod}

to_solve = input("Enter a two-operand math problem, with prefix notation: ")

operator, first_s, second_s = to_solve.split()     ❷
first = int(first_s)     ❸
second = int(second_s)

print(operations[operator](first, second))     ❹
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

❶ The keys in the `operations` dict are the operator strings that a user might enter, while the values are our functions, associated with those strings.

❷ Break the user's input apart

❸ Turn each of the user's inputs from strings into integers

❹ Apply the user's chosen operator as a key in `operations`, returning a function—which we then invoke, passing it `first` and `second` as arguments

Perhaps my favorite part of the code is the final line: We have a dict in which the functions are the values. We can thus retrieve the function we want with `operations[operator]`, where `operator` is the first part of the string that we broke apart with `str.split`. Once we have a function, we can call it with parentheses, passing it our two operands, `first` and `second`.

But how do we get `first` and `second`? From the user's input string, in which we assume that there are three elements. We use `str.split` to break them apart, and immediately use unpacking to assign them to three variables.

NOTE If you're uncomfortable with the idea of invoking `str.split` and simply assuming that we'll get three results back, there is an easy way to deal with this.

When you invoke `str.split`, pass a value to its optional `maxsplit` parameter. This parameter indicates how many splits will actually be performed; another way to think about it is that it's the index of the final element in the returned list. For example, if I write:

```
>>> s = 'a b c d e'
>>> s.split()
['a', 'b', 'c', 'd', 'e']
```

As you can see, I get (as always) a list of strings. Because I invoked `str.split` without any arguments, Python used any whitespace characters as separators.

But if I pass a value of 3 to `maxsplit`, I get the following:

```
>>> s = 'a b c d e'
>>> s.split(maxsplit=3)
['a', 'b', 'c', 'd e']
```

Notice that the returned list now has four elements. The Python documentation says that `maxsplit` tells `str.split` how many cuts to make. I prefer to think of that value as the largest index in the returned list—that is, because the returned list contains four elements, the final element will have an index of 3.

Either way, `maxsplit` ensures that when we use unpacking on the result from `maxsplit`, we're not going to encounter an error.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

All of this is fine, but this code doesn't seem very DRY — that is, the fact that we have to define each of our functions, even when they're so similar to one another and are re-implementing existing functionality, is a bit frustrating and out of character for Python.

Fortunately, the `operator` module, which comes with Python, can help us. By importing `operator`, we get precisely the functions we need—`add`, `sub`, `mul`, `truediv` and `floordiv`, `mod`, and `pow`. We no longer need to define our own functions, because we can use those provided by the module. The `add` function in `operators` does what we would normally expect from the `+` operator: It looks to its left, determines the type of the first parameter, and uses that in order to know what to invoke. `operator.add`, as a function, doesn't need to look to its left; it checks the type of its first argument, and uses that to determine which version of `+` to run.

In this particular exercise, we restricted the user's inputs to integers, so we didn't do any type checking. But you can imagine a version of this exercise in which we could handle a variety of different types, not just integers. In such a case, the various `operator` functions would know what to do with whatever types we would hand it.

6.2.3 Beyond the exercise

Treating functions as data, and storing them in data structures, is odd for many newcomers to Python. But it enables techniques that are possible, but far more complex, in other languages. Here are three more exercises that extend this idea even further:

1. Expand the program you wrote, such that the user's input can contain any number of numbers, not just two. The program will thus handle `+ 3 5 7` or `/ 100 5 5`, and will apply the operator from left to right—giving the answers `15` and `4`, respectively.
2. Write a function, apply_to_each, which takes two arguments: A function that takes a single argument, and an iterable. Return a list whose values are the result of applying the function to each element in the iterable. (If this sounds familiar, then maybe it is: This is an implementation of the classic `map` function, still available in Python, but not used very often any more.)
3. Write a function, transform_lines, that takes three arguments: A function that takes a single argument, the name of an input file, and the name of an output file. Calling the function will run the function on each line of the input file, with the results written to the output file. (Hint: The previous exercise and this one are closely related.)

6.3 Password generator generator

For far too long, many people used (and probably still use) the same password for many different systems. This means that if someone figures out your password on system A, then they can log into systems B, C, and D where you used the same password. For this reason, many people (including me) use software that creates (and then remembers) long, random passwords. If you use such a system, then even if system A is compromised, your logins on systems B, C, and D are all safe.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

In this exercise, we're going to create a password-generation function. Actually, we're going to create a factory for password-generation functions. That is, I might need to generate a large number of passwords, all of which use the same set of characters. (You know how it is: Some applications require a mix of capital letters, lowercase letters, numbers, and symbols, while others require that you only use letters, and still others allow both letters and digits.) You'll thus call the function `create_password_generator` with a string. That string will return a function, which itself takes an integer argument. Calling this function will return a password of the specified length, using the string from which it was created.

For example:

```
alpha_password = create_password_generator('abcdef')
symbol_password = create_password_generator('!@#$%')

print(alpha_password(5))    # efeaa
print(alpha_password(10))   # cacdacbada

print(symbol_password(5))   # %#@%@
print(symbol_password(10))  # @!%%$%$%%#
```

A useful function to know about in implementing this function is the `random` module (https://docs.python.org/3/library/random.html), and more specifically the `random.choice` function in that module, which returns one (randomly chosen) element from a sequence.

The point of this exercise is to understand how to work with inner functions: Defining them, returning them, and using them to create numerous, similar functions.

6.3.1 Solution

```
import random

def create_password_generator(characters):    ❶
    def create_password(length):    ❷
        output = []

        for i in range(length):    ❸
            output.append(random.choice(characters))    ❹
        return ''.join(output)    ❺
    return create_password    ❻
```

❶ Define the outer function

❷ Define the inner function; this `def` runs each time we run the outer function

❸ How long do we want the password to be?

❹ Add a new, random element from `characters` to `output`

❺ Return a string based on the elements of `output`

❻ Return the inner function to the caller

You can work through a version of this code in the Python Tutor at https://tinyurl.com/y6o2s5au

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

6.3.2 Discussion

This is an example of where you might want to use an inner function, sometimes known as a "closure." The idea is that we're invoking a function (`create_password_generator`) that returns a function (`create_password`). The returned, inner function knows what we did on our initial invocation, but also has some functionality of its own—so it needs to be defined as an inner function, so that it can access variables from the initial (outer) invocation.

The inner function is defined not when Python first executes the program, but rather when the outer function (`create_password_generator`) is executed. Indeed, we create a new inner function once for each time that `create_password_generator` is invoked.

If it is not used or returned from within the outer function, the inner function remains local to the outer function. So it's typical for the outer function to return the inner one to its caller, much as it might return a list, dictionary, or other object.

In this particular case, we want to eventually end up with a function to which we can pass an integer, and from which we can get a randomly generated password. But of course, the password must contain certain characters, and different programs (as I know all too well) have different restrictions on what characters can be used for those passwords. Thus, we might want 5 alphanumeric characters, or 10 numbers, or 15 characters that are either alphanumeric or punctuation.

To achieve this, we define our outer function such that it takes a single argument, a string containing the characters from which we want to create a new password. The result of invoking this function is, as was indicated, a function—the dynamically defined `create_password`. This inner function has access to the original `characters` variable in the outer function because of Python's LEGB (local, enclosing, global, and builtin) precedence rule for variable lookup. (See sidebar, "Scoping in Python.") When, inside of `create_password`, we look for the variable `characters`, it is found in the enclosing function's scope.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

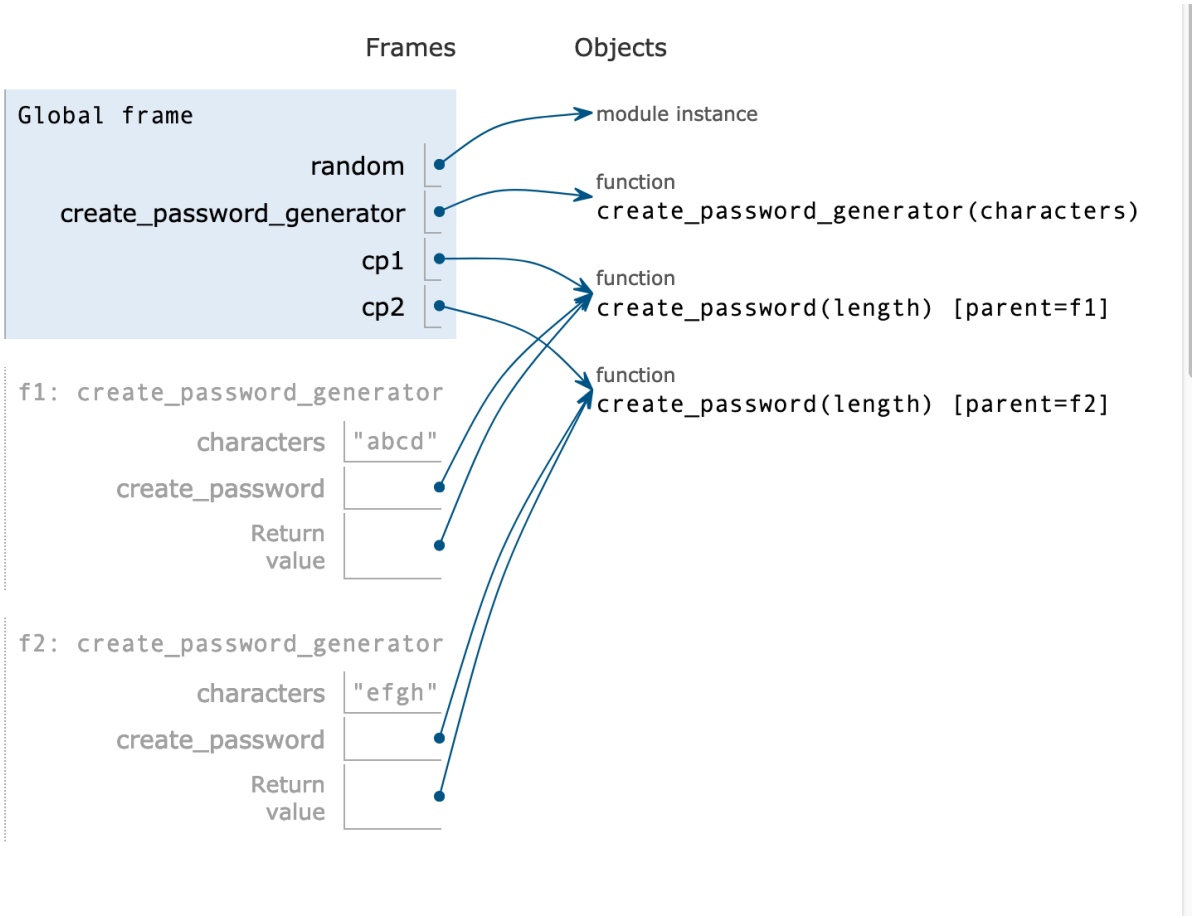


Figure 6.3 Python Tutor's depiction of two password-generating functions

If we invoke `create_password_generator` twice, as shown in this visualization via the Python Tutor, each invocation will return a separate version of `create_password`, with a separate value of `characters`. Each invocation of the outer function returns a new function, with its own local variables. At the same time, each of the returned inner functions has access to the local variables from its enclosing function. When we invoke one of the inner functions, we thus get a new password based on the combination of the inner function's local variables and the outer (enclosing) function's local variables.

6.3.3 Beyond the exercise

Thinking of functions as data lets you work at even higher levels of abstraction than usual functions, and thus solve even higher-level problems without worrying about the low-level details. . However, it can take some time to internalize and understand how to pass functions as arguments to other functions, or to return functions from inside of other functions. Here are some additional exercises you can try in order to better understand and work with them:

1. Write a function, `getitem`, that takes a single argument and return a function `f`. The returned `f` can then be invoked on any data structure whose elements can be selected via

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion
Licensed to Shai Asher <shaia@outlook.com>

square brackets, and returns that item. So if I invoke `f = getitem('a')`, and if I have a dictionary `d = {'a':1, 'b':2}`, then `f(d)` will return 1. (PS: This is very similar to `operator.itemgetter`, a very useful function in many circumstances.)

2. Write a function, `doboth`, that takes two functions as arguments (`f1` and `f2`) and returns a single function, `g`. Invoking `g(x)` should return the same result as invoking `f2(f1(x))`.
3. Write a function, `withfiles`, that takes a `glob`-style pattern of filenames. It should return a function `f` that takes a single argument `g`, another function. When we invoke `f`, it iterates over each of the filenames returned by `glob`, opening each file and passing it to `g`. The output from `f` is a dictionary in which the filenames are the keys and the values are the results of `g` being invoked on each file.

6.4 Conclusion

Writing simple Python functions isn't hard. But where Python's functions really shine is in their flexibility—especially when it comes to parameter interpretation—and in the fact that functions are data, too. In this chapter, we explored all of these ideas, which should give you some thoughts about how to take advantage of functions in your own programs.

If you ever find yourself writing similar code multiple times, then you should seriously consider generalizing it into a function that you can call from those locations. Moreover, if you find yourself implementing something that you might want to use in the future, implementing it as a function . Besides, it's often easier to understand, maintain, and test code that has been broken into functions — so even if you aren't worried about reuse or higher levels of abstraction, it might still be beneficial to write your code as functions.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/python-workout/discussion