

Chapters *To Go*



Python Crash Course: A Hands-On, Project-Based Introduction to Programming, 2nd Edition

by Eric Matthes

No Starch Press. (c) 2019. Copying Prohibited.

Reprinted for Richard Harold, Training

none@books24x7.com

Reprinted with permission as a subscription benefit of **Skillport**,

All rights reserved. Reproduction and/or distribution in whole or in part in electronic, paper or other forms without written permission is prohibited.



Chapter 4: Working with Lists

Overview

In Chapter 3 you learned how to make a simple list, and you learned to work with the individual elements in a list. In this chapter you'll learn how to *loop* through an entire list using just a few lines of code regardless of how long the list is. Looping allows you to take the same action, or set of actions, with every item in a list. As a result, you'll be able to work efficiently with lists of any length, including those with thousands or even millions of items.

Looping Through an Entire List

You'll often want to run through all entries in a list, performing the same task with each item. For example, in a game you might want to move every element on the screen by the same amount, or in a list of numbers you might want to perform the same statistical operation on every element. Or perhaps you'll want to display each headline from a list of articles on a website. When you want to do the same action with every item in a list, you can use Python's `for` loop.

Let's say we have a list of magicians' names, and we want to print out each name in the list. We could do this by retrieving each name from the list individually, but this approach could cause several problems. For one, it would be repetitive to do this with a long list of names. Also, we'd have to change our code each time the list's length changed. A `for` loop avoids both of these issues by letting Python manage these issues internally.

Let's use a `for` loop to print out each name in a list of magicians:

magicians.py

```
❶ magicians = ['alice', 'david', 'carolina']
❷ for magician in magicians:
❸     print(magician)
```

We begin by defining a list at ❶, just as we did in Chapter 3. At ❷, we define a `for` loop. This line tells Python to pull a name from the list `magicians`, and associate it with the variable `magician`. At ❸ we tell Python to print the name that's just been assigned to `magician`. Python then repeats lines ❷ and ❸, once for each name in the list. It might help to read this code as "For every magician in the list of magicians, print the magician's name." The output is a simple printout of each name in the list:

```
alice
david
carolina
```

A Closer Look at Looping

The concept of looping is important because it's one of the most common ways a computer automates repetitive tasks. For example, in a simple loop like we used in *magicians.py*, Python initially reads the first line of the loop:

```
for magician in magicians:
```

This line tells Python to retrieve the first value from the list `magicians` and associate it with the variable `magician`. This first value is `'alice'`. Python then reads the next line:

```
    print(magician)
```

Python prints the current value of `magician`, which is still `'alice'`. Because the list contains more values, Python returns to the first line of the loop:

```
for magician in magicians:
```

Python retrieves the next name in the list, `'david'`, and associates that value with the variable `magician`. Python then executes

the line:

```
print(magician)
```

Python prints the current value of `magician` again, which is now `'david'`. Python repeats the entire loop once more with the last value in the list, `'carolina'`. Because no more values are in the list, Python moves on to the next line in the program. In this case nothing comes after the `for` loop, so the program simply ends.

When you're using loops for the first time, keep in mind that the set of steps is repeated once for each item in the list, no matter how many items are in the list. If you have a million items in your list, Python repeats these steps a million times—and usually very quickly.

Also keep in mind when writing your own `for` loops that you can choose any name you want for the temporary variable that will be associated with each value in the list. However, it's helpful to choose a meaningful name that represents a single item from the list. For example, here's a good way to start a `for` loop for a list of cats, a list of dogs, and a general list of items:

```
for cat in cats:
for dog in dogs:
for item in list_of_items:
```

These naming conventions can help you follow the action being done on each item within a `for` loop. Using singular and plural names can help you identify whether a section of code is working with a single element from the list or the entire list.

Doing More Work within a for Loop

You can do just about anything with each item in a `for` loop. Let's build on the previous example by printing a message to each magician, telling them that they performed a great trick:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
❶     print(f"{magician.title()}, that was a great trick!")
```

The only difference in this code is at ❶ where we compose a message to each magician, starting with that magician's name. The first time through the loop the value of `magician` is `'alice'`, so Python starts the first message with the name `'Alice'`. The second time through the message will begin with `'David'`, and the third time through the message will begin with `'Carolina'`.

The output shows a personalized message for each magician in the list:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
```

You can also write as many lines of code as you like in the `for` loop. Every indented line following the line `for magician in magicians` is considered *inside the loop*, and each indented line is executed once for each value in the list. Therefore, you can do as much work as you like with each value in the list.

Let's add a second line to our message, telling each magician that we're looking forward to their next trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
❶    print(f"I can't wait to see your next trick, {magician.title()}. \n")
```

Because we have indented both calls to `print()`, each line will be executed once for every magician in the list. The newline (`"\n"`) in the second `print()` call ❶ inserts a blank line after each pass through the loop. This creates a set of messages that are neatly grouped for each person in the list:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

You can use as many lines as you like in your `for` loops. In practice you'll often find it useful to do a number of different operations with each item in a list when you use a `for` loop.

Doing Something after a for Loop

What happens once a `for` loop has finished executing? Usually, you'll want to summarize a block of output or move on to other work that your program must accomplish.

Any lines of code after the `for` loop that are not indented are executed once without repetition. Let's write a thank you to the group of magicians as a whole, thanking them for putting on an excellent show. To display this group message after all of the individual messages have been printed, we place the thank you message after the `for` loop without indentation:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

❶ print("Thank you, everyone. That was a great magic show!")
```

The first two calls to `print()` are repeated once for each magician in the list, as you saw earlier. However, because the line at ❶ is not indented, it's printed only once:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

When you're processing data using a `for` loop, you'll find that this is a good way to summarize an operation that was performed on an entire data set. For example, you might use a `for` loop to initialize a game by running through a list of characters and displaying each character on the screen. You might then write some additional code after this loop that displays a *Play Now* button after all the characters have been drawn to the screen.

Avoiding Indentation Errors

Python uses indentation to determine how a line, or group of lines, is related to the rest of the program. In the previous examples, the lines that printed messages to individual magicians were part of the `for` loop because they were indented. Python's use of indentation makes code very easy to read. Basically, it uses whitespace to force you to write neatly formatted code with a clear visual structure. In longer Python programs, you'll notice blocks of code indented at a few different levels. These indentation levels help you gain a general sense of the overall program's organization.

As you begin to write code that relies on proper indentation, you'll need to watch for a few common *indentation errors*. For example, people sometimes indent lines of code that don't need to be indented or forget to indent lines that need to be indented. Seeing examples of these errors now will help you avoid them in the future and correct them when they do appear in your own programs.

Let's examine some of the more common indentation errors.

Forgetting to Indent

Always indent the line after the `for` statement in a loop. If you forget, Python will remind you:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
❶ print(magician)
```

The call to `print()` ❶ should be indented, but it's not. When Python expects an indented block and doesn't find one, it lets you know which line it had a problem with.

```
File "magicians.py", line 3
    print(magician)
    ^
IndentationError: expected an indented block
```

You can usually resolve this kind of indentation error by indenting the line or lines immediately after the `for` statement.

Forgetting to Indent Additional Lines

Sometimes your loop will run without any errors but won't produce the expected result. This can happen when you're trying to do several tasks in a loop and you forget to indent some of its lines.

For example, this is what happens when we forget to indent the second line in the loop that tells each magician we're looking forward to their next trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
❶ print(f"I can't wait to see your next trick, {magician.title()}.")
```

The call to `print()` ❶ is supposed to be indented, but because Python finds at least one indented line after the `for` statement, it doesn't report an error. As a result, the first `print()` call is executed once for each name in the list because it is indented. The second `print()` call is not indented, so it is executed only once after the loop has finished running. Because the final value associated with `magician` is `'carolina'`, she is the only one who receives the "looking forward to the next trick" message:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

This is a *logical error*. The syntax is valid Python code, but the code does not produce the desired result because a problem occurs in its logic. If you expect to see a certain action repeated once for each item in a list and it's executed only once, determine whether you need to simply indent a line or a group of lines.

Indenting Unnecessarily

If you accidentally indent a line that doesn't need to be indented, Python informs you about the unexpected indent:

hello_world.py

```
message = "Hello Python world!"
❶ print(message)
```

We don't need to indent the `print()` call ❶, because it isn't part of a loop; hence, Python reports that error:

```
File "hello_world.py", line 2
    print(message)
    ^
```

```
IndentationError: unexpected indent
```

You can avoid unexpected indentation errors by indenting only when you have a specific reason to do so. In the programs you're writing at this point, the only lines you should indent are the actions you want to repeat for each item in a `for` loop.

Indenting Unnecessarily after the Loop

If you accidentally indent code that should run after a loop has finished, that code will be repeated once for each item in the list. Sometimes this prompts Python to report an error, but often this will result in a logical error.

For example, let's see what happens when we accidentally indent the line that thanked the magicians as a group for putting on a good show:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.")
    print("Thank you everyone, that was a great magic show!")
```

Because the line at ❶ is indented, it's printed once for each person in the list, as shown here:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

Thank you everyone, that was a great magic show!
David, that was a great trick!
I can't wait to see your next trick, David.

Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you everyone, that was a great magic show!
```

This is another logical error, similar to the one in "[Forgetting to Indent Additional Lines](#)" on [page 54](#). Because Python doesn't know what you're trying to accomplish with your code, it will run all code that is written in valid syntax. If an action is repeated many times when it should be executed only once, you probably need to unindent the code for that action.

Forgetting the Colon

The colon at the end of a `for` statement tells Python to interpret the next line as the start of a loop.

```
magicians = ['alice', 'david', 'carolina']
❶ for magician in magicians
    print(magician)
```

If you accidentally forget the colon, as shown at ❶, you'll get a syntax error because Python doesn't know what you're trying to do. Although this is an easy error to fix, it's not always an easy error to find. You'd be surprised by the amount of time programmers spend hunting down single-character errors like this. Such errors are difficult to find because we often just see what we expect to see.

TRY IT YOURSELF

4-1. Pizzas: Think of at least three kinds of your favorite pizza. Store these pizza names in a list, and then use a `for` loop to print the name of each pizza.

- Modify your `for` loop to print a sentence using the name of the pizza instead of printing just the name of the pizza. For each pizza you should have one line of output containing a simple statement like *I like pepperoni pizza*.

- Add a line at the end of your program, outside the `for` loop, that states how much you like pizza. The output should consist of three or more lines about the kinds of pizza you like and then an additional sentence, such as *I really love pizza!*

4-2. Animals: Think of at least three different animals that have a common characteristic. Store the names of these animals in a list, and then use a `for` loop to print out the name of each animal.

- Modify your program to print a statement about each animal, such as *A dog would make a great pet.*
- Add a line at the end of your program stating what these animals have in common. You could print a sentence such as *Any of these animals would make a great pet!*

Making Numerical Lists

Many reasons exist to store a set of numbers. For example, you'll need to keep track of the positions of each character in a game, and you might want to keep track of a player's high scores as well. In data visualizations, you'll almost always work with sets of numbers, such as temperatures, distances, population sizes, or latitude and longitude values, among other types of numerical sets.

Lists are ideal for storing sets of numbers, and Python provides a variety of tools to help you work efficiently with lists of numbers. Once you understand how to use these tools effectively, your code will work well even when your lists contain millions of items.

Using the `range()` Function

Python's `range()` function makes it easy to generate a series of numbers. For example, you can use the `range()` function to print a series of numbers like this:

first_numbers.py

```
for value in range(1, 5):
    print(value)
```

Although this code looks like it should print the numbers from 1 to 5, it doesn't print the number 5:

```
1
2
3
4
```

In this example, `range()` prints only the numbers 1 through 4. This is another result of the off-by-one behavior you'll see often in programming languages. The `range()` function causes Python to start counting at the first value you give it, and it stops when it reaches the second value you provide. Because it stops at that second value, the output never contains the end value, which would have been 5 in this case.

To print the numbers from 1 to 5, you would use `range(1, 6)`:

```
for value in range(1, 6):
    print(value)
```

This time the output starts at 1 and ends at 5:

```
1
2
3
4
5
```

If your output is different than what you expect when you're using `range()`, try adjusting your end value by 1.

You can also pass `range()` only one argument, and it will start the sequence of numbers at 0. For example, `range(6)` would return the numbers from 0 through 5.

Using `range()` to Make a List of Numbers

If you want to make a list of numbers, you can convert the results of `range()` directly into a list using the `list()` function. When you wrap `list()` around a call to the `range()` function, the output will be a list of numbers.

In the example in the previous section, we simply printed out a series of numbers. We can use `list()` to convert that same set of numbers into a list:

```
numbers = list(range(1, 6))
print(numbers)
```

And this is the result:

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

We can also use the `range()` function to tell Python to skip numbers in a given range. If you pass a third argument to `range()`, Python uses that value as a step size when generating numbers.

For example, here's how to list the even numbers between 1 and 10:

even_numbers.py

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

In this example, the `range()` function starts with the value 2 and then adds 2 to that value. It adds 2 repeatedly until it reaches or passes the end value, 11, and produces this result:

```
[2, 4, 6, 8, 10]
```

You can create almost any set of numbers you want to using the `range()` function. For example, consider how you might make a list of the first 10 square numbers (that is, the square of each integer from 1 through 10). In Python, two asterisks (`**`) represent exponents. Here's how you might put the first 10 square numbers into a list:

squares.py

```
❶ squares = []
❷ for value in range(1, 11):
❸     square = value ** 2
❹     squares.append(square)
❺ print(squares)
```

We start with an empty list called `squares` ❶. At ❷, we tell Python to loop through each value from 1 to 10 using the `range()` function. Inside the loop, the current value is raised to the second power and assigned to the variable `square` ❸. At ❹, each new value of `square` is appended to the list `squares`. Finally, when the loop has finished running, the list of squares is printed ❺:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

To write this code more concisely, omit the temporary variable `square` and append each new value directly to the list:

```
squares = []
for value in range(1,11):
❶     squares.append(value**2)
```

```
print(squares)
```

The code at ❶ does the same work as the lines at ❸ and ❹ in *squares.py*. Each value in the loop is raised to the second power and then immediately appended to the list of squares.

You can use either of these two approaches when you're making more complex lists. Sometimes using a temporary variable makes your code easier to read; other times it makes the code unnecessarily long. Focus first on writing code that you understand clearly, which does what you want it to do. Then look for more efficient approaches as you review your code.

Simple Statistics with a List of Numbers

A few Python functions are helpful when working with lists of numbers. For example, you can easily find the minimum, maximum, and sum of a list of numbers:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

Note The examples in this section use short lists of numbers in order to fit easily on the page. They would work just as well if your list contained a million or more numbers.

List Comprehensions

The approach described earlier for generating the list `squares` consisted of using three or four lines of code. A *list comprehension* allows you to generate this same list in just one line of code. A list comprehension combines the `for` loop and the creation of new elements into one line, and automatically appends each new element. List comprehensions are not always presented to beginners, but I have included them here because you'll most likely see them as soon as you start looking at other people's code.

The following example builds the same list of square numbers you saw earlier but uses a list comprehension:

squares.py

```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

To use this syntax, begin with a descriptive name for the list, such as `squares`. Next, open a set of square brackets and define the expression for the values you want to store in the new list. In this example the expression is `value**2`, which raises the value to the second power. Then, write a `for` loop to generate the numbers you want to feed into the expression, and close the square brackets. The `for` loop in this example is `for value in range(1, 11)`, which feeds the values 1 through 10 into the expression `value**2`. Notice that no colon is used at the end of the `for` statement.

The result is the same list of square numbers you saw earlier:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

It takes practice to write your own list comprehensions, but you'll find them worthwhile once you become comfortable creating ordinary lists. When you're writing three or four lines of code to generate lists and it begins to feel repetitive, consider writing your own list comprehensions.

TRY IT YOURSELF

4-3. Counting to Twenty: Use a `for` loop to print the numbers from 1 to 20, inclusive.

4-4. One Million: Make a list of the numbers from one to one million, and then use a `for` loop to print the numbers. (If the output is taking too long, stop it by pressing CTRL-C or by closing the output window.)

4-5. Summing a Million: Make a list of the numbers from one to one million, and then use `min()` and `max()` to make sure your list actually starts at one and ends at one million. Also, use the `sum()` function to see how quickly Python can add a million numbers.

4-6. Odd Numbers: Use the third argument of the `range()` function to make a list of the odd numbers from 1 to 20. Use a `for` loop to print each number.

4-7. Threes: Make a list of the multiples of 3 from 3 to 30. Use a `for` loop to print the numbers in your list.

4-8. Cubes: A number raised to the third power is called a *cube*. For example, the cube of 2 is written as `2**3` in Python. Make a list of the first 10 cubes (that is, the cube of each integer from 1 through 10), and use a `for` loop to print out the value of each cube.

4-9. Cube Comprehension: Use a list comprehension to generate a list of the first 10 cubes.

Working with Part of a List

In Chapter 3 you learned how to access single elements in a list, and in this chapter you've been learning how to work through all the elements in a list. You can also work with a specific group of items in a list, which Python calls a *slice*.

Slicing a List

To make a slice, you specify the index of the first and last elements you want to work with. As with the `range()` function, Python stops one item before the second index you specify. To output the first three elements in a list, you would request indices 0 through 3, which would return elements 0, 1, and 2.

The following example involves a list of players on a team:

players.py

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
❶ print(players[0:3])
```

The code at ❶ prints a slice of this list, which includes just the first three players. The output retains the structure of the list and includes the first three players in the list:

```
['charles', 'martina', 'michael']
```

You can generate any subset of a list. For example, if you want the second, third, and fourth items in a list, you would start the slice at index 1 and end at index 4:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

This time the slice starts with 'martina' and ends with 'florence':

```
['martina', 'michael', 'florence']
```

If you omit the first index in a slice, Python automatically starts your slice at the beginning of the list:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

Without a starting index, Python starts at the beginning of the list:

```
['charles', 'martina', 'michael', 'florence']
```

A similar syntax works if you want a slice that includes the end of a list. For example, if you want all items from the third item through the last item, you can start with index 2 and omit the second index:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

Python returns all items from the third item through the end of the list:

```
['michael', 'florence', 'eli']
```

This syntax allows you to output all of the elements from any point in your list to the end regardless of the length of the list. Recall that a negative index returns an element a certain distance from the end of a list; therefore, you can output any slice from the end of a list. For example, if we want to output the last three players on the roster, we can use the slice `players[-3:]`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

This prints the names of the last three players and would continue to work as the list of players changes in size.

Note You can include a third value in the brackets indicating a slice. If a third value is included, this tells Python how many items to skip between items in the specified range.

Looping Through a Slice

You can use a slice in a `for` loop if you want to loop through a subset of the elements in a list. In the next example we loop through the first three players and print their names as part of a simple roster:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
❶ for player in players[:3]:
    print(player.title())
```

Instead of looping through the entire list of players at ❶, Python loops through only the first three names:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Slices are very useful in a number of situations. For instance, when you're creating a game, you could add a player's final score to a list every time that player finishes playing. You could then get a player's top three scores by sorting the list in decreasing order and taking a slice that includes just the first three scores. When you're working with data, you can use slices to process your data in chunks of a specific size. Or, when you're building a web application, you could use slices to display information in a series of pages with an appropriate amount of information on each page.

Copying a List

Often, you'll want to start with an existing list and make an entirely new list based on the first one. Let's explore how copying a list works and examine one situation in which copying a list is useful.

To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index (`[:]`). This tells Python to make a slice that starts at the first item and ends with the last item, producing a copy of the entire list.

For example, imagine we have a list of our favorite foods and want to make a separate list of foods that a friend likes. This friend likes everything in our list so far, so we can create their list by copying ours:

foods.py

```

❶ my_foods = ['pizza', 'falafel', 'carrot cake']
❷ friend_foods = my_foods[:]

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)

```

At ❶ we make a list of the foods we like called `my_foods`. At ❷ we make a new list called `friend_foods`. We make a copy of `my_foods` by asking for a slice of `my_foods` without specifying any indices and store the copy in `friend_foods`. When we print each list, we see that they both contain the same foods:

```

My favorite foods are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']

```

To prove that we actually have two separate lists, we'll add a new food to each list and show that each list keeps track of the appropriate person's favorite foods:

```

my_foods = ['pizza', 'falafel', 'carrot cake']
❶ friend_foods = my_foods[:]

❷ my_foods.append('cannoli')
❸ friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)

```

At ❶ we copy the original items in `my_foods` to the new list `friend_foods`, as we did in the previous example. Next, we add a new food to each list: at ❷ we add `'cannoli'` to `my_foods`, and at ❸ we add `'ice cream'` to `friend_foods`. We then print the two lists to see whether each of these foods is in the appropriate list.

```

My favorite foods are:
❹ ['pizza', 'falafel', 'carrot cake', 'cannoli']

My friend's favorite foods are:
❺ ['pizza', 'falafel', 'carrot cake', 'ice cream']

```

The output at ❹ shows that `'cannoli'` now appears in our list of favorite foods but `'ice cream'` doesn't. At ❺ we can see that `'ice cream'` now appears in our friend's list but `'cannoli'` doesn't. If we had simply set `friend_foods` equal to `my_foods`, we would not produce two separate lists. For example, here's what happens when you try to copy a list without using a slice:

```

my_foods = ['pizza', 'falafel', 'carrot cake']

# This doesn't work:
❶ friend_foods = my_foods

my_foods.append('cannoli')
friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)

```

Instead of storing a copy of `my_foods` in `friend_foods` at ❶, we set `friend_foods` equal to `my_foods`. This syntax actually tells

Python to associate the new variable `friend_foods` with the list that is already associated with `my_foods`, so now both variables point to the same list. As a result, when we add `'cannoli'` to `my_foods`, it will also appear in `friend_foods`. Likewise `'ice cream'` will appear in both lists, even though it appears to be added only to `friend_foods`.

The output shows that both lists are the same now, which is not what we wanted:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

Note Don't worry about the details in this example for now. Basically, if you're trying to work with a copy of a list and you see unexpected behavior, make sure you are copying the list using a slice, as we did in the first example.

TRY IT YOURSELF

4-10. Slices: Using one of the programs you wrote in this chapter, add several lines to the end of the program that do the following:

- Print the message *The first three items in the list are:*. Then use a slice to print the first three items from that program's list.
- Print the message *Three items from the middle of the list are:*. Use a slice to print three items from the middle of the list.
- Print the message *The last three items in the list are:*. Use a slice to print the last three items in the list.

4-11. My Pizzas, Your Pizzas: Start with your program from [Exercise 4-1 \(page 56\)](#). Make a copy of the list of pizzas, and call it `friend_pizzas`. Then, do the following:

- Add a new pizza to the original list.
- Add a different pizza to the list `friend_pizzas`.
- Prove that you have two separate lists. Print the message *My favorite pizzas are:*, and then use a `for` loop to print the first list. Print the message *My friend's favorite pizzas are:*, and then use a `for` loop to print the second list. Make sure each new pizza is stored in the appropriate list.

4-12. More Loops: All versions of `foods.py` in this section have avoided using `for` loops when printing to save space. Choose a version of `foods.py`, and write two `for` loops to print each list of foods.

Tuples

Lists work well for storing collections of items that can change throughout the life of a program. The ability to modify lists is particularly important when you're working with a list of users on a website or a list of characters in a game. However, sometimes you'll want to create a list of items that cannot change. Tuples allow you to do just that. Python refers to values that cannot change as *immutable*, and an immutable list is called a *tuple*.

Defining a Tuple

A tuple looks just like a list except you use parentheses instead of square brackets. Once you define a tuple, you can access individual elements by using each item's index, just as you would for a list.

For example, if we have a rectangle that should always be a certain size, we can ensure that its size doesn't change by putting the dimensions into a tuple:

dimensions.py

```
❶ dimensions = (200, 50)
❷ print(dimensions[0])
   print(dimensions[1])
```

We define the tuple `dimensions` at ❶, using parentheses instead of square brackets. At ❷ we print each element in the tuple individually, using the same syntax we've been using to access elements in a list:

```
200
50
```

Let's see what happens if we try to change one of the items in the tuple `dimensions`:

```
dimensions = (200, 50)
❶ dimensions[0] = 250
```

The code at ❶ tries to change the value of the first dimension, but Python returns a type error. Basically, because we're trying to alter a tuple, which can't be done to that type of object, Python tells us we can't assign a new value to an item in a tuple:

```
Traceback (most recent call last):
  File "dimensions.py", line 2, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

This is beneficial because we want Python to raise an error when a line of code tries to change the dimensions of the rectangle.

Note Tuples are technically defined by the presence of a comma; the parentheses make them look neater and more readable. If you want to define a tuple with one element, you need to include a trailing comma:

```
my_t = (3,)
```

It doesn't often make sense to build a tuple with one element, but this can happen when tuples are generated automatically.

Looping Through All Values in a Tuple

You can loop over all the values in a tuple using a `for` loop, just as you did with a list:

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

Python returns all the elements in the tuple, just as it would for a list:

```
200
50
```

Writing over a Tuple

Although you can't modify a tuple, you can assign a new value to a variable that represents a tuple. So if we wanted to change our dimensions, we could redefine the entire tuple:

```
❶ dimensions = (200, 50)
   print("Original dimensions:")
   for dimension in dimensions:
       print(dimension)

❷ dimensions = (400, 100)
❸ print("\nModified dimensions:")
   for dimension in dimensions:
       print(dimension)
```

The lines starting at ❶ define the original tuple and print the initial dimensions. At ❷, we associate a new tuple with the

variable `dimensions`. We then print the new dimensions at ❸. Python doesn't raise any errors this time, because reassigning a variable is valid:

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

When compared with lists, tuples are simple data structures. Use them when you want to store a set of values that should not be changed throughout the life of a program.

TRY IT YOURSELF

4-13. Buffet: A buffet-style restaurant offers only five basic foods. Think of five simple foods, and store them in a tuple.

- Use a `for` loop to print each food the restaurant offers.
- Try to modify one of the items, and make sure that Python rejects the change.
- The restaurant changes its menu, replacing two of the items with different foods. Add a line that rewrites the tuple, and then use a `for` loop to print each of the items on the revised menu.

Styling Your Code

Now that you're writing longer programs, ideas about how to style your code are worthwhile to know. Take the time to make your code as easy as possible to read. Writing easy-to-read code helps you keep track of what your programs are doing and helps others understand your code as well.

Python programmers have agreed on a number of styling conventions to ensure that everyone's code is structured in roughly the same way. Once you've learned to write clean Python code, you should be able to understand the overall structure of anyone else's Python code, as long as they follow the same guidelines. If you're hoping to become a professional programmer at some point, you should begin following these guidelines as soon as possible to develop good habits.

The Style Guide

When someone wants to make a change to the Python language, they write a *Python Enhancement Proposal (PEP)*. One of the oldest PEPs is *PEP 8*, which instructs Python programmers on how to style their code. PEP 8 is fairly lengthy, but much of it relates to more complex coding structures than what you've seen so far.

The Python style guide was written with the understanding that code is read more often than it is written. You'll write your code once and then start reading it as you begin debugging. When you add features to a program, you'll spend more time reading your code. When you share your code with other programmers, they'll read your code as well.

Given the choice between writing code that's easier to write or code that's easier to read, Python programmers will almost always encourage you to write code that's easier to read. The following guidelines will help you write clear code from the start.

Indentation

PEP 8 recommends that you use four spaces per indentation level. Using four spaces improves readability while leaving room for multiple levels of indentation on each line.

In a word processing document, people often use tabs rather than spaces to indent. This works well for word processing documents, but the Python interpreter gets confused when tabs are mixed with spaces. Every text editor provides a setting that lets you use the TAB key but then converts each tab to a set number of spaces. You should definitely use your TAB key, but also make sure your editor is set to insert spaces rather than tabs into your document.

Mixing tabs and spaces in your file can cause problems that are very difficult to diagnose. If you think you have a mix of tabs and spaces, you can convert all tabs in a file to spaces in most editors.

Line Length

Many Python programmers recommend that each line should be less than 80 characters. Historically, this guideline developed because most computers could fit only 79 characters on a single line in a terminal window. Currently, people can fit much longer lines on their screens, but other reasons exist to adhere to the 79-character standard line length. Professional programmers often have several files open on the same screen, and using the standard line length allows them to see entire lines in two or three files that are open side by side onscreen. PEP 8 also recommends that you limit all of your comments to 72 characters per line, because some of the tools that generate automatic documentation for larger projects add formatting characters at the beginning of each commented line.

The PEP 8 guidelines for line length are not set in stone, and some teams prefer a 99-character limit. Don't worry too much about line length in your code as you're learning, but be aware that people who are working collaboratively almost always follow the PEP 8 guidelines. Most editors allow you to set up a visual cue, usually a vertical line on your screen, that shows you where these limits are.

Note Appendix B shows you how to configure your text editor so it always inserts four spaces each time you press the TAB key and shows a vertical guideline to help you follow the 79-character limit.

Blank Lines

To group parts of your program visually, use blank lines. You should use blank lines to organize your files, but don't do so excessively. By following the examples provided in this book, you should strike the right balance. For example, if you have five lines of code that build a list, and then another three lines that do something with that list, it's appropriate to place a blank line between the two sections. However, you should not place three or four blank lines between the two sections.

Blank lines won't affect how your code runs, but they will affect the readability of your code. The Python interpreter uses horizontal indentation to interpret the meaning of your code, but it disregards vertical spacing.

Other Style Guidelines

PEP 8 has many additional styling recommendations, but most of the guidelines refer to more complex programs than what you're writing at this point. As you learn more complex Python structures, I'll share the relevant parts of the PEP 8 guidelines.

TRY IT YOURSELF

4-14. PEP 8: Look through the original PEP 8 style guide at <https://python.org/dev/peps/pep-0008/>. You won't use much of it now, but it might be interesting to skim through it.

4-15. Code Review: Choose three of the programs you've written in this chapter and modify each one to comply with PEP 8:

- Use four spaces for each indentation level. Set your text editor to insert four spaces every time you press TAB, if you haven't already done so (see Appendix B for instructions on how to do this).
- Use less than 80 characters on each line, and set your editor to show a vertical guideline at the 80th character position.
- Don't use blank lines excessively in your program files.

Summary

In this chapter you learned how to work efficiently with the elements in a list. You learned how to work through a list using a `for` loop, how Python uses indentation to structure a program, and how to avoid some common indentation errors. You learned to make simple numerical lists, as well as a few operations you can perform on numerical lists. You learned how to slice a list to work with a subset of items and how to copy lists properly using a slice. You also learned about tuples, which provide a degree of protection to a set of values that shouldn't change, and how to style your increasingly complex code to make it easy to read.

In Chapter 5, you'll learn to respond appropriately to different conditions by using `if` statements. You'll learn to string together relatively complex sets of conditional tests to respond appropriately to exactly the kind of situation or information you're looking for. You'll also learn to use `if` statements while looping through a list to take specific actions with selected elements from a list.