Sequences: Lists and Tuples

5

Objectives

In this chapter, you'll:

- Create and initialize lists and tuples.
- Refer to elements of lists, tuples and strings.
- Sort and search lists, and search tuples.
- Pass lists and tuples to functions and methods.
- Use list methods to perform common manipulations, such as searching for items, sorting a list, inserting items and removing items.
- Use additional Python functional-style programming capabilities, including lambdas and the functional-style programming operations filter, map and reduce.
- Use functional-style list comprehensions to create lists quickly and easily, and use generator expressions to generate values on demand.
- Use two-dimensional lists.
- Enhance your analysis and presentation skills with the Seaborn and Matplotlib visualization libraries.

Outline

5.1 Introduction

In the last two chapters, we briefly introduced the list and tuple sequence types for representing ordered collections of items. **Collections** are prepackaged data structures consisting of related data items. Examples of collections include your favorite songs on your smartphone, your contacts list, a library's books, your cards in a card game, your favorite sports team's players, the stocks in an investment portfolio, patients in a cancer study and a shopping list. Python's built-in collections enable you to store and access data conveniently and efficiently. In this chapter, we discuss lists and tuples in more detail.

We'll demonstrate common list and tuple manipulations. You'll see that lists (which are modifiable) and tuples (which are not) have many common capabilities. Each can hold items of the same or different types. Lists can **dynamically resize** as necessary, growing and shrinking at execution time. We discuss one-dimensional and two-dimensional lists.

In the preceding chapter, we demonstrated random-number generation and simulated rolling a six-sided die. We conclude this chapter with our next Intro to Data Science section, which uses the visualization libraries Seaborn and Matplotlib to interactively develop static bar charts containing the die frequencies. In the next chapter's Intro to Data Science section, we'll present an animated visualization in which the bar chart changes *dynamically* as the number of die rolls increases—you'll see the law of large numbers "in action."

5.2 Lists

Here, we discuss lists in more detail and explain how to refer to particular list **elements**. Many of the capabilities shown in this section apply to all sequence types.

Creating a List

Lists typically store **homogeneous data**, that is, values of the *same* data type. Consider the list `c`, which contains five integer elements:

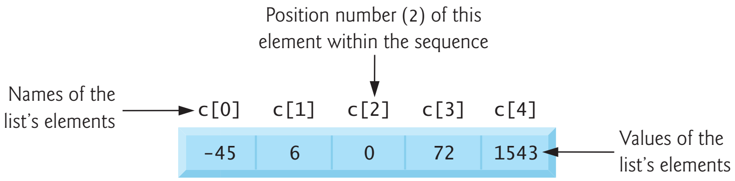
```
In [1]: c = [-45, 6, 0, 72, 1543]

In [2]: c
Out[2]: [-45, 6, 0, 72, 1543]
```

They also may store **heterogeneous data**, that is, data of many different types. For example, the following list contains a student's first name (a string), last name (a string), grade point average (a `float`) and graduation year (an `int`):

```
['Mary', 'Smith', 3.57, 2022]
```

Accessing Elements of a List

You reference a list element by writing the list's name followed by the element's **index** (that is, its **position number**) enclosed in square brackets (`[]`, known as the **subscription operator**). The following diagram shows the list `c` labeled with its element names:


Position number (2) of this element within the sequence
Names of the list's elements
c[0]
c[1]
c[2]
c[3]
c[4]
-45
6
0
72
1543
Values of the list's elements

The first element in a list has the index `0`. So, in the five-element list `c`, the first element is named `c[0]` and the last is `c[4]`:

```
In [3]: c[0]
Out[3]: -45

In [4]: c[4]
Out[4]: 1543
```

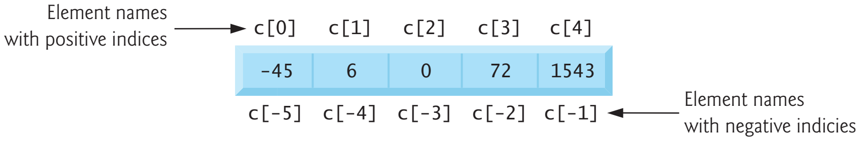
Determining a List's Length

To get a list's length, use the built-in **`len` function**:

```
In [5]: len(c)
Out[5]: 5
```

Accessing Elements from the End of the List with Negative Indices

Lists also can be accessed from the end by using *negative indices*:


Element names with positive indices
c[0]
c[1]
c[2]
c[3]
c[4]
-45
6
0
72
1543
c[-5]
c[-4]
c[-3]
c[-2]
c[-1]
Element names with negative indicies

So, list `c`'s last element (`c[4]`), can be accessed with `c[-1]` and its first element with `c[-5]`:

```
In [6]: c[-1]
Out[6]: 1543

In [7]: c[-5]
Out[7]: -45
```

Indices Must Be Integers or Integer Expressions

An index must be an integer or integer expression (or a *slice*, as we'll soon see):

```
In [8]: a = 1

In [9]: b = 2

In [10]: c[a + b]
Out[10]: 72
```

Using a non-integer index value causes a `TypeError`.

Lists Are Mutable

Lists are mutable—their elements can be modified:

```
In [11]: c[4] = 17

In [12]: c
Out[12]: [-45, 6, 0, 72, 17]
```

You'll soon see that you also can insert and delete elements, changing the list's length.

Some Sequences Are Immutable

Python's string and tuple sequences are immutable—they cannot be modified. You can get the individual characters in a string, but attempting to assign a new value to one of the characters causes a `TypeError`:

```
In [13]: s = 'hello'

In [14]: s[0]
Out[14]: 'h'

In [15]: s[0] = 'H'
-------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-812ef2514689> in <module>()
----> 1 s[0] = 'H'

TypeError: 'str' object does not support item assignment
```

Attempting to Access a Nonexistent Element

Using an out-of-range list, tuple or string index causes an `IndexError`:

```
In [16]: c[100]
-------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-19-9a31ea1e1a13> in <module>()
----> 1 c[100]

IndexError: list index out of range
```

Using List Elements in Expressions

List elements may be used as variables in expressions:

```
In [17]: c[0] + c[1] + c[2]
Out[17]: -39
```

Appending to a List with +=

Let's start with an *empty* list `[]`, then use a `for` statement and `+=` to append the values 1 through 5 to the list—the list grows dynamically to accommodate each item:

```
In [18]: a_list = []

In [19]: for number in range(1, 6):
    ...:     a_list += [number]
    ...:

In [20]: a_list
Out[20]: [1, 2, 3, 4, 5]
```

When the left operand of += is a list, the right operand must be an *iterable*; otherwise, a TypeError occurs. In snippet [19]'s suite, the square brackets around number create a one-element list, which we append to a_list. If the right operand contains multiple elements, += appends them all. The following appends the characters of 'Python' to the list letters:

```
In [21]: letters = []

In [22]: letters += 'Python'

In [23]: letters
Out[23]: ['P', 'y', 't', 'h', 'o', 'n']
```

If the right operand of += is a tuple, its elements also are appended to the list. Later in the chapter, we'll use the list method append to add items to a list.

Concatenating Lists with +

You can **concatenate** two lists, two tuples or two strings using the + operator. The result is a *new* sequence of the same type containing the left operand's elements followed by the right operand's elements. The original sequences are unchanged:

```
In [24]: list1 = [10, 20, 30]

In [25]: list2 = [40, 50]

In [26]: concatenated_list = list1 + list2

In [27]: concatenated_list
Out[27]: [10, 20, 30, 40, 50]
```

A TypeError occurs if the + operator's operands are difference sequence types—for example, concatenating a list and a tuple is an error.

Using for and range to Access List Indices and Values

List elements also can be accessed via their indices and the subscription operator ([]):

```
In [28]: for i in range(len(concatenated_list)):
    ...:     print(f'{i}: {concatenated_list[i]}')
    ...:
0: 10
1: 20
2: 30
3: 40
4: 50
```

The function call range(len(concatenated_list)) produces a sequence of integers representing concatenated_list's indices (in this case, 0 through 4). When looping in this manner, you must ensure that indices remain in range. Soon, we'll show a safer way to access element indices and values using built-in function enumerate.

Comparison Operators

You can compare entire lists element-by-element using comparison operators:

```
In [29]: a = [1, 2, 3]

In [30]: b = [1, 2, 3]

In [31]: c = [1, 2, 3, 4]

In [32]: a == b  # True: corresponding elements in both are equal
Out[32]: True

In [33]: a == c  # False: a and c have different elements and lengths
Out[33]: False

In [34]: a < c  # True: a has fewer elements than c
Out[34]: True

In [35]: c >= b  # True: elements 0-2 are equal but c has more elements
Out[35]: True
```

✓ Self Check

1 *(Fill-In)* Python's string and tuple sequences are ________—they cannot be modified.
Answer: immutable.

2 *(True/False)* The + operator's sequence operands may be of any sequence type.
Answer: False. The + operator's operand sequences must have the *same* type; otherwise, a `TypeError` occurs.

3 *(IPython Session)* Create a function `cube_list` that cubes each element of a list. Call the function with the list `numbers` containing 1 through 10. Show `numbers` after the call.
Answer:

```
In [1]: def cube_list(values):
   ...:     for i in range(len(values)):
   ...:         values[i] **= 3
   ...:

In [2]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [3]: cube_list(numbers)

In [4]: numbers
Out[4]: [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

4 *(IPython Session)* Use an empty list named `characters` and a += augmented assignment statement to convert the string `'Birthday'` into a list of its characters.
Answer:

```
In [5]: characters = []

In [6]: characters += 'Birthday'

In [7]: characters
Out[7]: ['B', 'i', 'r', 't', 'h', 'd', 'a', 'y']
```

5.3 Tuples

As discussed in the preceding chapter, tuples are immutable and typically store heterogeneous data, but the data can be homogeneous. A tuple's length is its number of elements and cannot change during program execution.

Creating Tuples

To create an empty tuple, use empty parentheses:

```
In [1]: student_tuple = ()

In [2]: student_tuple
Out[2]: ()

In [3]: len(student_tuple)
Out[3]: 0
```

Recall that you can pack a tuple by separating its values with commas:

```
In [4]: student_tuple = 'John', 'Green', 3.3

In [5]: student_tuple
Out[5]: ('John', 'Green', 3.3)

In [6]: len(student_tuple)
Out[6]: 3
```

When you output a tuple, Python always displays its contents in parentheses. You may surround a tuple's comma-separated list of values with optional parentheses:

```
In [7]: another_student_tuple = ('Mary', 'Red', 3.3)

In [8]: another_student_tuple
Out[8]: ('Mary', 'Red', 3.3)
```

The following code creates a one-element tuple:

```
In [9]: a_singleton_tuple = ('red',)  # note the comma

In [10]: a_singleton_tuple
Out[10]: ('red',)
```

The comma (,) that follows the string `'red'` identifies `a_singleton_tuple` as a tuple—the parentheses are optional. If the comma were omitted, the parentheses would be redundant, and `a_singleton_tuple` would simply refer to the string `'red'` rather than a tuple.

Accessing Tuple Elements

A tuple's elements, though related, are often of multiple types. Usually, you do not iterate over them. Rather, you access each individually. Like list indices, tuple indices start at 0. The following code creates `time_tuple` representing an hour, minute and second, displays the tuple, then uses its elements to calculate the number of seconds since midnight—note that we perform a *different* operation with each value in the tuple:

```
In [11]: time_tuple = (9, 16, 1)

In [12]: time_tuple
Out[12]: (9, 16, 1)
```

```
In [13]: time_tuple[0] * 3600 + time_tuple[1] * 60 + time_tuple[2]
Out[13]: 33361
```

Assigning a value to a tuple element causes a `TypeError`.

Adding Items to a String or Tuple

As with lists, the += augmented assignment statement can be used with strings and tuples, even though they're *immutable*. In the following code, after the two assignments, `tuple1` and `tuple2` refer to the *same* tuple object:

```
In [14]: tuple1 = (10, 20, 30)

In [15]: tuple2 = tuple1

In [16]: tuple2
Out[16]: (10, 20, 30)
```

Concatenating the tuple `(40, 50)` to `tuple1` creates a *new* tuple, then assigns a reference to it to the variable `tuple1`—`tuple2` still refers to the original tuple:

```
In [17]: tuple1 += (40, 50)

In [18]: tuple1
Out[18]: (10, 20, 30, 40, 50)

In [19]: tuple2
Out[19]: (10, 20, 30)
```

For a string or tuple, the item to the right of += must be a string or tuple, respectively—mixing types causes a `TypeError`.

Appending Tuples to Lists

You can use += to append a tuple to a list:

```
In [20]: numbers = [1, 2, 3, 4, 5]

In [21]: numbers += (6, 7)

In [22]: numbers
Out[22]: [1, 2, 3, 4, 5, 6, 7]
```

Tuples May Contain Mutable Objects

Let's create a `student_tuple` with a first name, last name and list of grades:

```
In [23]: student_tuple = ('Amanda', 'Blue', [98, 75, 87])
```

Even though the tuple is immutable, its list element is mutable:

```
In [24]: student_tuple[2][1] = 85

In [25]: student_tuple
Out[25]: ('Amanda', 'Blue', [98, 85, 87])
```

In the *double-subscripted name* `student_tuple[2][1]`, Python views `student_tuple[2]` as the element of the tuple containing the list `[98, 75, 87]`, then uses `[1]` to access the list element containing `75`. The assignment in snippet `[24]` replaces that grade with 85.

Self Check

1 *(True/False)* A += augmented assignment statement may not be used with strings and tuples, because they're immutable.
Answer: False. A += augmented assignment statement also may be used with strings and tuples, even though they're immutable. The result is a *new* string or tuple, respectively.

2 *(True/False)* Tuples can contain only immutable objects.
Answer: False. Even though a tuple is immutable, its elements can be mutable objects, such as lists.

3 *(IPython Session)* Create a single-element tuple containing 123.45, then display it.
Answer:

```
In [1]: single = (123.45,)

In [2]: single
Out[2]: (123.45,)
```

4 *(IPython Session)* Show what happens when you attempt to concatenate sequences of different types—the list `[1, 2, 3]` and the tuple `(4, 5, 6)`—using the + operator.
Answer:

```
In [3]: [1, 2, 3] + (4, 5, 6)
-------------------------------------------------------------------------
TypeError                               Traceback (most recent call last)
<ipython-input-3-1ac3d3041bfa> in <module>()
----> 1 [1, 2, 3] + (4, 5, 6)

TypeError: can only concatenate list (not "tuple") to list
```

5.4 Unpacking Sequences

The previous chapter introduced tuple unpacking. You can unpack any sequence's elements by assigning the sequence to a comma-separated list of variables. A `ValueError` occurs if the number of variables to the left of the assignment symbol is not identical to the number of elements in the sequence on the right:

```
In [1]: student_tuple = ('Amanda', [98, 85, 87])

In [2]: first_name, grades = student_tuple

In [3]: first_name
Out[3]: 'Amanda'

In [4]: grades
Out[4]: [98, 85, 87]
```

The following code unpacks a string, a list and a sequence produced by `range`:

```
In [5]: first, second = 'hi'

In [6]: print(f'{first}  {second}')
h  i
```

```
In [7]: number1, number2, number3 = [2, 3, 5]

In [8]: print(f'{number1}  {number2}  {number3}')
2  3  5

In [9]: number1, number2, number3 = range(10, 40, 10)

In [10]: print(f'{number1}  {number2}  {number3}')
10  20  30
```

Swapping Values Via Packing and Unpacking

You can swap two variables' values using sequence packing and unpacking:

```
In [11]: number1 = 99

In [12]: number2 = 22

In [13]: number1, number2 = (number2, number1)

In [14]: print(f'number1 = {number1}; number2 = {number2}')
number1 = 22; number2 = 99
```

Accessing Indices and Values Safely with Built-in Function enumerate

Earlier, we called `range` to produce a sequence of index values, then accessed list elements in a `for` loop using the index values and the subscription operator (`[]`). This is error-prone because you could pass the wrong arguments to `range`. If any value produced by `range` is an out-of-bounds index, using it as an index causes an `IndexError`.

The preferred mechanism for accessing an element's index *and* value is the built-in function **`enumerate`**. This function receives an iterable and creates an iterator that, for each element, returns a tuple containing the element's index and value. The following code uses the built-in function **`list`** to create a list containing `enumerate`'s results:

```
In [15]: colors = ['red', 'orange', 'yellow']

In [16]: list(enumerate(colors))
Out[16]: [(0, 'red'), (1, 'orange'), (2, 'yellow')]
```

Similarly the built-in function **`tuple`** creates a tuple from a sequence:

```
In [17]: tuple(enumerate(colors))
Out[17]: ((0, 'red'), (1, 'orange'), (2, 'yellow'))
```

The following `for` loop unpacks each tuple returned by `enumerate` into the variables `index` and `value` and displays them:

```
In [18]: for index, value in enumerate(colors):
    ...:     print(f'{index}: {value}')
    ...:
0: red
1: orange
2: yellow
```

Creating a Primitive Bar Chart

The script in Fig. 5.1 creates a primitive **bar chart** where each bar's length is made of asterisks (*) and is proportional to the list's corresponding element value. We use the function enumerate to access the list's indices and values safely. To run this example, change to this chapter's ch05 examples folder, then enter:

```
ipython fig05_01.py
```

or, if you're in IPython already, use the command:

```
run fig05_01.py
```

```python
# fig05_01.py
"""Displaying a bar chart"""
numbers = [19, 3, 15, 7, 11]

print('\nCreating a bar chart from numbers:')
print(f'Index{"Value":>8}   Bar')

for index, value in enumerate(numbers):
    print(f'{index:>5}{value:>8}   {"*" * value}')
```

```
Creating a bar chart from numbers:
Index   Value   Bar
    0      19   *******************
    1       3   ***
    2      15   ***************
    3       7   *******
    4      11   ***********
```

Fig. 5.1 | Displaying a bar chart.

The for statement uses enumerate to get each element's index and value, then displays a formatted line containing the index, the element value and the corresponding bar of asterisks. The expression

```
"*" * value
```

creates a string consisting of value asterisks. When used with a sequence, the multiplication operator (*) *repeats* the sequence—in this case, the string "*"—value times. Later in this chapter, we'll use the open-source Seaborn and Matplotlib libraries to display a publication-quality bar chart visualization.

Self Check

1 *(Fill-In)* A sequence's elements can be ________ by assigning the sequence to a comma-separated list of variables.
Answer: unpacked.

2 *(True/False)* The following expression causes an error:

```
'-' * 10
```

Answer: False: In this context, the multiplication operator (*) repeats the string ('-') 10 times.

3 *(IPython Session)* Create a tuple `high_low` representing a day of the week (a string) and its high and low temperatures (integers), display its string representation, then perform the following tasks in an interactive IPython session:

a) Use the `[]` operator to access and display the `high_low` tuple's elements.

b) Unpack the `high_low` tuple into the variables `day` and `high`. What happens and why?

Answer: For Part (b) an error occurs because you must unpack *all* the elements of a sequence.

```
In [1]: high_low = ('Monday', 87, 65)

In [2]: high_low
Out[2]: ('Monday', 87, 65)

In [3]: print(f'{high_low[0]}: High={high_low[1]}, Low={high_low[2]}')
Monday: High=87, Low=65

In [4]: day, high = high_low
-------------------------------------------------------------------------
ValueError                              Traceback (most recent call last)
<ipython-input-3-0c3ad5c97284> in <module>()
----> 1 day, high = high_low

ValueError: too many values to unpack (expected 2)
```

4 *(IPython Session)* Create the list `names` containing three name strings. Use a `for` loop and the `enumerate` function to iterate through the elements and display each element's index and value.

Answer:

```
In [4]: names = ['Amanda', 'Sam', 'David']

In [5]: for i, name in enumerate(names):
   ...:     print(f'{i}: {name}')
   ...:
0: Amanda
1: Sam
2: David
```

5.5 Sequence Slicing

You can **slice** sequences to create new sequences of the same type containing *subsets* of the original elements. Slice operations can modify mutable sequences—those that do *not* modify a sequence work identically for lists, tuples and strings.

Specifying a Slice with Starting and Ending Indices

Let's create a slice consisting of the elements at indices 2 through 5 of a list:

```
In [1]: numbers = [2, 3, 5, 7, 11, 13, 17, 19]

In [2]: numbers[2:6]
Out[2]: [5, 7, 11, 13]
```

The slice *copies* elements from the *starting index* to the left of the colon (`2`) up to, but not including, the *ending index* to the right of the colon (`6`). The original list is not modified.

Specifying a Slice with Only an Ending Index

If you omit the starting index, 0 is assumed. So, the slice numbers[:6] is equivalent to the slice numbers[0:6]:

```
In [3]: numbers[:6]
Out[3]: [2, 3, 5, 7, 11, 13]

In [4]: numbers[0:6]
Out[4]: [2, 3, 5, 7, 11, 13]
```

Specifying a Slice with Only a Starting Index

If you omit the ending index, Python assumes the sequence's length (8 here), so snippet [5]'s slice contains the elements of numbers at indices 6 and 7:

```
In [5]: numbers[6:]
Out[5]: [17, 19]

In [6]: numbers[6:len(numbers)]
Out[6]: [17, 19]
```

Specifying a Slice with No Indices

Omitting both the start and end indices copies the entire sequence:

```
In [7]: numbers[:]
Out[7]: [2, 3, 5, 7, 11, 13, 17, 19]
```

Though slices create new objects, slices make *shallow* copies of the elements—that is, they copy the elements' references but not the objects they point to. So, in the snippet above, the new list's elements refer to the *same objects* as the original list's elements, rather than to separate copies. In the "Array-Oriented Programming with NumPy" chapter, we'll explain *deep* copying, which actually copies the referenced objects themselves, and we'll point out when deep copying is preferred.

Slicing with Steps

The following code uses a *step* of 2 to create a slice with every other element of numbers:

```
In [8]: numbers[::2]
Out[8]: [2, 5, 11, 17]
```

We omitted the start and end indices, so 0 and len(numbers) are assumed, respectively.

Slicing with Negative Indices and Steps

You can use a negative step to select slices in *reverse* order. The following code concisely creates a new list in reverse order:

```
In [9]: numbers[::-1]
Out[9]: [19, 17, 13, 11, 7, 5, 3, 2]
```

This is equivalent to:

```
In [10]: numbers[-1:-9:-1]
Out[10]: [19, 17, 13, 11, 7, 5, 3, 2]
```

Modifying Lists Via Slices

You can modify a list by assigning to a slice of it—the rest of the list is unchanged. The following code replaces numbers' first three elements, leaving the rest unchanged:

```
In [11]: numbers[0:3] = ['two', 'three', 'five']

In [12]: numbers
Out[12]: ['two', 'three', 'five', 7, 11, 13, 17, 19]
```

The following deletes only the first three elements of `numbers` by assigning an *empty* list to the three-element slice:

```
In [13]: numbers[0:3] = []

In [14]: numbers
Out[14]: [7, 11, 13, 17, 19]
```

The following assigns a list's elements to a slice of every other element of `numbers`:

```
In [15]: numbers = [2, 3, 5, 7, 11, 13, 17, 19]

In [16]: numbers[::2] = [100, 100, 100, 100]

In [17]: numbers
Out[17]: [100, 3, 100, 7, 100, 13, 100, 19]

In [18]: id(numbers)
Out[18]: 4434456648
```

Let's delete all the elements in `numbers`, leaving the *existing* list empty:

```
In [19]: numbers[:] = []

In [20]: numbers
Out[20]: []

In [21]: id(numbers)
Out[21]: 4434456648
```

Deleting `numbers`' contents (snippet `[19]`) is different from assigning `numbers` a *new* empty list `[]` (snippet `[22]`). To prove this, we display `numbers`' identity after each operation. The identities are different, so they represent separate objects in memory:

```
In [22]: numbers = []

In [23]: numbers
Out[23]: []

In [24]: id(numbers)
Out[24]: 4406030920
```

When you assign a new object to a variable (as in snippet `[21]`), the original object will be garbage collected if no other variables refer to it.

✓ Self Check

1 *(True/False)* Slice operations that modify a sequence work identically for lists, tuples and strings.
Answer: False. Slice operations that *do not* modify a sequence work identically for lists, tuples and strings.

2 *(Fill-In)* Assume you have a list called `names`. The slice expression ________ creates a new list with the elements of `names` in reverse order.
Answer: `names[::-1]`

3 *(IPython Session)* Create a list called numbers containing the values from 1 through 15, then use *slices* to perform the following operations consecutively:

a) Select number's even integers.
b) Replace the elements at indices 5 through 9 with 0s, then show the resulting list.
c) Keep only the first five elements, then show the resulting list.
d) Delete all the remaining elements by assigning to a slice. Show the resulting list.

Answer:

```
In [1]: numbers = list(range(1, 16))

In [2]: numbers
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

In [3]: numbers[1:len(numbers):2]
Out[3]: [2, 4, 6, 8, 10, 12, 14]

In [4]: numbers[5:10] = [0] * len(numbers[5:10])

In [5]: numbers
Out[5]: [1, 2, 3, 4, 5, 0, 0, 0, 0, 0, 11, 12, 13, 14, 15]

In [6]: numbers[5:] = []

In [7]: numbers
Out[7]: [1, 2, 3, 4, 5]

In [8]: numbers[:] = []

In [9]: numbers
Out[9]: []
```

Recall that multiplying a sequence repeats that sequence the specified number of times.

5.6 del Statement

The **del statement** also can be used to remove elements from a list and to delete variables from the interactive session. You can remove the element at any valid index or the element(s) from any valid slice.

Deleting the Element at a Specific List Index

Let's create a list, then use del to remove its last element:

```
In [1]: numbers = list(range(0, 10))

In [2]: numbers
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: del numbers[-1]

In [4]: numbers
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Deleting a Slice from a List

The following deletes the list's first two elements:

```
In [5]: del numbers[0:2]

In [6]: numbers
Out[6]: [2, 3, 4, 5, 6, 7, 8]
```

The following uses a step in the slice to delete every other element from the entire list:

```
In [7]: del numbers[::2]

In [8]: numbers
Out[8]: [3, 5, 7]
```

Deleting a Slice Representing the Entire List

The following code deletes all of the list's elements:

```
In [9]: del numbers[:]

In [10]: numbers
Out[10]: []
```

Deleting a Variable from the Current Session

The `del` statement can delete any variable. Let's delete `numbers` from the interactive session, then attempt to display the variable's value, causing a `NameError`:

```
In [11]: del numbers

In [12]: numbers
-------------------------------------------------------------------------
NameError                               Traceback (most recent call last)
<ipython-input-12-426f8401232b> in <module>()
----> 1 numbers

NameError: name 'numbers' is not defined
```

Self Check

1 *(Fill-In)* Given a list `numbers` containing 1 through 10, `del numbers[-2]` removes the value ________ from the list.
Answer: 9.

2 *(IPython Session)* Create a list called `numbers` containing the values from `1` through `15`, then use the `del` statement to perform the following operations consecutively:

a) Delete a slice containing the first four elements, then show the resulting list.
b) Starting with the first element, use a slice to delete every other element of the list, then show the resulting list.

Answer:

```
In [1]: numbers = list(range(1, 16))

In [2]: numbers
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

In [3]: del numbers[0:4]

In [4]: numbers
Out[4]: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

In [5]: del numbers[::2]

In [6]: numbers
Out[6]: [6, 8, 10, 12, 14]
```

5.7 Passing Lists to Functions

In the last chapter, we mentioned that all objects are passed by reference and demonstrated passing an immutable object as a function argument. Here, we discuss references further by examining what happens when a program passes a mutable list object to a function.

Passing an Entire List to a Function

Consider the function `modify_elements`, which receives a reference to a list and multiplies each of the list's element values by `2`:

```
In [1]: def modify_elements(items):
   ...:     """"Multiplies all element values in items by 2."""
   ...:     for i in range(len(items)):
   ...:         items[i] *= 2
   ...:

In [2]: numbers = [10, 3, 7, 1, 9]

In [3]: modify_elements(numbers)

In [4]: numbers
Out[4]: [20, 6, 14, 2, 18]
```

Function `modify_elements`' `items` parameter receives a reference to the *original* list, so the statement in the loop's suite modifies each element in the original list object.

Passing a Tuple to a Function

When you pass a tuple to a function, attempting to modify the tuple's immutable elements results in a `TypeError`:

```
In [5]: numbers_tuple = (10, 20, 30)

In [6]: numbers_tuple
Out[6]: (10, 20, 30)

In [7]: modify_elements(numbers_tuple)
-------------------------------------------------------------------------
TypeError                               Traceback (most recent call last)
<ipython-input-27-9339741cd595> in <module>()
----> 1 modify_elements(numbers_tuple)

<ipython-input-25-27acb8f8f44c> in modify_elements(items)
      2     """"Multiplies all element values in items by 2."""
      3     for i in range(len(items)):
----> 4         items[i] *= 2
      5
      6

TypeError: 'tuple' object does not support item assignment
```

Recall that tuples may contain mutable objects, such as lists. Those objects still can be modified when a tuple is passed to a function.

A Note Regarding Tracebacks

The previous traceback shows the *two* snippets that led to the `TypeError`. The first is snippet `[7]`'s function call. The second is snippet `[1]`'s function definition. Line numbers precede each snippet's code. We've demonstrated mostly single-line snippets. When an exception occurs in such a snippet, it's always preceded by `---->` 1, indicating that line 1 (the snippet's only line) caused the exception. Multiline snippets like the definition of `modify_elements` show consecutive line numbers starting at 1. The notation `----> 4` above indicates that the exception occurred in line 4 of `modify_elements`. No matter how long the traceback is, the last line of code with `---->` caused the exception.

Self Check

1 *(True/False)* You cannot modify a list's contents when you pass it to a function.
Answer: False. When you pass a list (a mutable object) to a function, the function receives a reference to the original list object and can use that reference to modify the original list's contents.

2 *(True/False)* Tuples can contain lists and other mutable objects. Those mutable objects can be modified when a tuple is passed to a function.
Answer: True.

5.8 Sorting Lists

A common computing task called **sorting** enables you to arrange data either in ascending or descending order. Sorting is an intriguing problem that has attracted intense computer-science research efforts. It's studied in detail in data-structures and algorithms courses. We discuss sorting in more detail in the "Computer Science Thinking: Recursion, Searching, Sorting and Big O" chapter.

Sorting a List in Ascending Order

List method **`sort`** *modifies* a list to arrange its elements in ascending order:

```
In [1]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [2]: numbers.sort()

In [3]: numbers
Out[3]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Sorting a List in Descending Order

To sort a list in descending order, call list method `sort` with the optional keyword argument **`reverse`** set to `True` (`False` is the default):

```
In [4]: numbers.sort(reverse=True)

In [5]: numbers
Out[5]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Built-In Function sorted

Built-in function **`sorted`** *returns a new list* containing the sorted elements of its argument *sequence*—the original sequence is *unmodified*. The following code demonstrates function `sorted` for a list, a string and a tuple:

```
In [6]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [7]: ascending_numbers = sorted(numbers)
In [8]: ascending_numbers
Out[8]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [9]: numbers
Out[9]: [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [10]: letters = 'fadgchjebi'

In [11]: ascending_letters = sorted(letters)

In [12]: ascending_letters
Out[12]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

In [13]: letters
Out[13]: 'fadgchjebi'

In [14]: colors = ('red', 'orange', 'yellow', 'green', 'blue')

In [15]: ascending_colors = sorted(colors)

In [16]: ascending_colors
Out[16]: ['blue', 'green', 'orange', 'red', 'yellow']

In [17]: colors
Out[17]: ('red', 'orange', 'yellow', 'green', 'blue')
```

Use the optional keyword argument `reverse` with the value `True` to sort the elements in descending order.

✓ Self Check

1 *(Fill-In)* To sort a list in descending order, call list method `sort` with the optional keyword argument ________ set to `True`.
Answer: `reverse`.

2 *(True/False)* All sequences provide a `sort` method.
Answer: False. Immutable sequences like tuples and strings do not provide a `sort` method. However, you can sort *any* sequence *without modifying it* by using built-in function `sorted`, which returns a *new* list containing the sorted elements of its argument sequence.

3 *(IPython Session)* Create a `foods` list containing `'Cookies'`, `'pizza'`, `'Grapes'`, `'apples'`, `'steak'` and `'Bacon'`. Use list method `sort` to sort the list in ascending order. Are the strings in alphabetical order?
Answer:

```
In [1]: foods = ['Cookies', 'pizza', 'Grapes',
   ...:          'apples', 'steak', 'Bacon']
   ...:

In [2]: foods.sort()

In [3]: foods
Out[3]: ['Bacon', 'Cookies', 'Grapes', 'apples', 'pizza', 'steak']
```

They're probably not in what you'd consider alphabetical order, but they are in order as defined by the underlying character set—known as **lexicographical order**. As you'll see later in the chapter, strings are compared by their character's numerical values, not their letters, and the values of uppercase letters are *lower* than the values of lowercase letters.

5.9 Searching Sequences

Often, you'll want to determine whether a sequence (such as a list, tuple or string) contains a value that matches a particular **key** value. **Searching** is the process of locating a key.

List Method index

List method **index** takes as an argument a search key—the value to locate in the list—then searches through the list from index 0 and returns the index of the *first* element that matches the search key:

```
In [1]: numbers = [3, 7, 1, 4, 2, 8, 5, 6]

In [2]: numbers.index(5)
Out[2]: 6
```

A ValueError occurs if the value you're searching for is not in the list.

Specifying the Starting Index of a Search

Using method index's optional arguments, you can search a subset of a list's elements. You can use *= to *multiply a sequence*—that is, append a sequence to itself multiple times. After the following snippet, numbers contains two copies of the original list's contents:

```
In [3]: numbers *= 2

In [4]: numbers
Out[4]: [3, 7, 1, 4, 2, 8, 5, 6, 3, 7, 1, 4, 2, 8, 5, 6]
```

The following code searches the updated list for the value 5 starting from index 7 and continuing through the end of the list:

```
In [5]: numbers.index(5, 7)
Out[5]: 14
```

Specifying the Starting and Ending Indices of a Search

Specifying the starting and ending indices causes index to search from the starting index up to but not including the ending index location. The call to index in snippet [5]:

```
numbers.index(5, 7)
```

assumes the length of numbers as its optional third argument and is equivalent to:

```
numbers.index(5, 7, len(numbers))
```

The following looks for the value 7 in the range of elements with indices 0 through 3:

```
In [6]: numbers.index(7, 0, 4)
Out[6]: 1
```

Operators in and not in

Operator in tests whether its right operand's iterable contains the left operand's value:

```
In [7]: 1000 in numbers
Out[7]: False

In [8]: 5 in numbers
Out[8]: True
```

Similarly, operator `not in` tests whether its right operand's iterable does *not* contain the left operand's value:

```
In [9]: 1000 not in numbers
Out[9]: True

In [10]: 5 not in numbers
Out[10]: False
```

Using Operator `in` to Prevent a `ValueError`

You can use the operator `in` to ensure that calls to method `index` do not result in `ValueErrors` for search keys that are not in the corresponding sequence:

```
In [11]: key = 1000

In [12]: if key in numbers:
    ...:     print(f'found {key} at index {numbers.index(search_key)}')
    ...: else:
    ...:     print(f'{key} not found')
    ...:
1000 not found
```

Built-In Functions `any` and `all`

Sometimes you simply need to know whether *any* item in an iterable is `True` or whether *all* the items are `True`. The built-in function **`any`** returns `True` if any item in its iterable argument is `True`. The built-in function **`all`** returns `True` if all items in its iterable argument are `True`. Recall that nonzero values are `True` and 0 is `False`. Non-empty iterable objects also evaluate to `True`, whereas any empty iterable evaluates to `False`. Functions `any` and `all` are additional examples of internal iteration in functional-style programming.

Self Check

1 *(Fill-In)* The ________ operator can be used to extend a list with copies of itself.
Answer: *=.

2 *(Fill-In)* Operators ________ and ________ determine whether a sequence contains or does not contain a value, respectively.
Answer: `in`, `not in`.

3 *(IPython Session)* Create a five-element list containing 67, 12, 46, 43 and 13, then use list method `index` to search for a 43 and 44. Ensure that no `ValueError` occurs when searching for 44.
Answer:

```
In [1]: numbers = [67, 12, 46, 43, 13]

In [2]: numbers.index(43)
Out[2]: 3

In [3]: if 44 in numbers:
   ...:     print(f'Found 44 at index: {numbers.index(44)}')
   ...: else:
   ...:     print('44 not found')
   ...:
44 not found
```

5.10 Other List Methods

Lists also have methods that add and remove elements. Consider the list `color_names`:

```
    In [1]: color_names = ['orange', 'yellow', 'green']
```

Inserting an Element at a Specific List Index

Method **`insert`** adds a new item at a specified index. The following inserts `'red'` at index `0`:

```
    In [2]: color_names.insert(0, 'red')

    In [3]: color_names
    Out[3]: ['red', 'orange', 'yellow', 'green']
```

Adding an Element to the End of a List

You can add a new item to the end of a list with method **`append`**:

```
    In [4]: color_names.append('blue')

    In [5]: color_names
    Out[5]: ['red', 'orange', 'yellow', 'green', 'blue']
```

Adding All the Elements of a Sequence to the End of a List

Use list method **`extend`** to add all the elements of another sequence to the end of a list:

```
    In [6]: color_names.extend(['indigo', 'violet'])

    In [7]: color_names
    Out[7]: ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

This is the equivalent of using +=. The following code adds all the characters of a string then all the elements of a tuple to a list:

```
    In [8]: sample_list = []

    In [9]: s = 'abc'

    In [10]: sample_list.extend(s)

    In [11]: sample_list
    Out[11]: ['a', 'b', 'c']

    In [12]: t = (1, 2, 3)

    In [13]: sample_list.extend(t)

    In [14]: sample_list
    Out[14]: ['a', 'b', 'c', 1, 2, 3]
```

Rather than creating a temporary variable, like `t`, to store a tuple before appending it to a list, you might want to pass a tuple directly to `extend`. In this case, the tuple's parentheses are required, because `extend` expects one iterable argument:

```
    In [15]: sample_list.extend((4, 5, 6))  # note the extra parentheses

    In [16]: sample_list
    Out[16]: ['a', 'b', 'c', 1, 2, 3, 4, 5, 6]
```

A `TypeError` occurs if you omit the required parentheses.

Removing the First Occurrence of an Element in a List

Method **remove** deletes the first element with a specified value—a `ValueError` occurs if `remove`'s argument is not in the list:

```
In [17]: color_names.remove('green')

In [18]: color_names
Out[18]: ['red', 'orange', 'yellow', 'blue', 'indigo', 'violet']
```

Emptying a List

To delete all the elements in a list, call method **clear**:

```
In [19]: color_names.clear()

In [20]: color_names
Out[20]: []
```

This is the equivalent of the previously shown slice assignment

```
color_names[:] = []
```

Counting the Number of Occurrences of an Item

List method **count** searches for its argument and returns the number of times it is found:

```
In [21]: responses = [1, 2, 5, 4, 3, 5, 2, 1, 3, 3,
     ...:              1, 4, 3, 3, 3, 2, 3, 3, 2, 2]
     ...:

In [22]: for i in range(1, 6):
    ...:     print(f'{i} appears {responses.count(i)} times in responses')
    ...:
1 appears 3 times in responses
2 appears 5 times in responses
3 appears 8 times in responses
4 appears 2 times in responses
5 appears 2 times in responses
```

Reversing a List's Elements

List method **reverse** reverses the contents of a list in place, rather than creating a reversed copy, as we did with a slice previously:

```
In [23]: color_names = ['red', 'orange', 'yellow', 'green', 'blue']

In [24]: color_names.reverse()

In [25]: color_names
Out[25]: ['blue', 'green', 'yellow', 'orange', 'red']
```

Copying a List

List method `copy` returns a *new* list containing a *shallow* copy of the original list:

```
In [26]: copied_list = color_names.copy()

In [27]: copied_list
Out[27]: ['blue', 'green', 'yellow', 'orange', 'red']
```

This is equivalent to the previously demonstrated slice operation:

```
copied_list = color_names[:]
```

Self Check

1 *(Fill-In)* To add all the elements of a sequence to the end of a list, use list method ________, which is equivalent to using `+=`.
Answer: `extend`.

2 *(Fill-In)* For a list `numbers`, calling method ________ is equivalent to `numbers[:] = []`.
Answer: `clear`.

3 *(IPython Session)* Create a list called `rainbow` containing `'green'`, `'orange'` and `'violet'`. Perform the following operations consecutively using list methods and show the list's contents after each operation:

a) Determine the index of `'violet'`, then use it to insert `'red'` before `'violet'`.
b) Append `'yellow'` to the end of the list.
c) Reverse the list's elements.
d) Remove the element `'orange'`.

Answer:

```
In [1]: rainbow = ['green', 'orange', 'violet']

In [2]: rainbow.insert(rainbow.index('violet'), 'red')

In [3]: rainbow
Out[3]: ['green', 'orange', 'red', 'violet']

In [4]: rainbow.append('yellow')

In [5]: rainbow
Out[5]: ['green', 'orange', 'red', 'violet', 'yellow']

In [6]: rainbow.reverse()

In [7]: rainbow
Out[7]: ['yellow', 'violet', 'red', 'orange', 'green']

In [8]: rainbow.remove('orange')

In [9]: rainbow
Out[9]: ['yellow', 'violet', 'red', 'green']
```

5.11 Simulating Stacks with Lists

The preceding chapter introduced the function-call stack. Python does not have a built-in stack type, but you can think of a stack as a constrained list. You *push* using list method `append`, which adds a new element to the *end* of the list. You *pop* using list method **`pop`** with no arguments, which removes and returns the item at the *end* of the list.

Let's create an empty list called `stack`, push (`append`) two strings onto it, then `pop` the strings to confirm they're retrieved in last-in, first-out (LIFO) order:

```
In [1]: stack = []

In [2]: stack.append('red')

In [3]: stack
Out[3]: ['red']
```

```
In [4]: stack.append('green')

In [5]: stack
Out[5]: ['red', 'green']

In [6]: stack.pop()
Out[6]: 'green'

In [7]: stack
Out[7]: ['red']

In [8]: stack.pop()
Out[8]: 'red'

In [9]: stack
Out[9]: []

In [10]: stack.pop()
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-10-50ea7ec13fbe> in <module>()
----> 1 stack.pop()

IndexError: pop from empty list
```

For each `pop` snippet, the value that `pop` removes and returns is displayed. Popping from an empty stack causes an `IndexError`, just like accessing a nonexistent list element with `[]`. To prevent an `IndexError`, ensure that `len(stack)` is greater than `0` before calling `pop`. You can run out of memory if you keep pushing items faster than you pop them.

In the exercises, you'll use a list to simulate another popular collection called a **queue** in which you insert at the back and delete from the front. Items are retrieved from queues in **first-in, first-out (FIFO) order**.

✓ Self Check

1 *(Fill-In)* You can simulate a stack with a list, using methods ________ and ________ to add and remove elements, respectively, only at the end of the list.
Answer: `append`, `pop`.

2 *(Fill-In)* To prevent an `IndexError` when calling `pop` on a list, first ensure that ________.
Answer: the list's length is greater than 0.

5.12 List Comprehensions

Here, we continue discussing *functional-style* features with **list comprehensions**—a concise and convenient notation for creating new lists. List comprehensions can replace many `for` statements that iterate over existing sequences and create new lists, such as:

```
In [1]: list1 = []

In [2]: for item in range(1, 6):
   ...:     list1.append(item)
   ...:
```

```
In [3]: list1
Out[3]: [1, 2, 3, 4, 5]
```

Using a List Comprehension to Create a List of Integers

We can accomplish the same task in a single line of code with a list comprehension:

```
In [4]: list2 = [item for item in range(1, 6)]

In [5]: list2
Out[5]: [1, 2, 3, 4, 5]
```

Like snippet `[2]`'s `for` statement, the list comprehension's **`for` clause**

```
    for item in range(1, 6)
```

iterates over the sequence produced by `range(1, 6)`. For each `item`, the list comprehension evaluates the expression to the left of the `for` clause and places the expression's value (in this case, the `item` itself) in the new list. Snippet `[4]`'s particular comprehension could have been expressed more concisely using the function `list`:

```
    list2 = list(range(1, 6))
```

Mapping: Performing Operations in a List Comprehension's Expression

A list comprehension's expression can perform tasks, such as calculations, that **map** elements to new values (possibly of different types). Mapping is a common functional-style programming operation that produces a result with the *same* number of elements as the original data being mapped. The following comprehension maps each value to its cube with the expression `item ** 3`:

```
In [6]: list3 = [item ** 3 for item in range(1, 6)]

In [7]: list3
Out[7]: [1, 8, 27, 64, 125]
```

Filtering: List Comprehensions with `if` Clauses

Another common functional-style programming operation is **filtering** elements to select only those that match a condition. This typically produces a list with *fewer* elements than the data being filtered. To do this in a list comprehension, use the **`if` clause**. The following includes in `list4` only the even values produced by the `for` clause:

```
In [8]: list4 = [item for item in range(1, 11) if item % 2 == 0]

In [9]: list4
Out[9]: [2, 4, 6, 8, 10]
```

List Comprehension That Processes Another List's Elements

The `for` clause can process any iterable. Let's create a list of lowercase strings and use a list comprehension to create a new list containing their uppercase versions:

```
In [10]: colors = ['red', 'orange', 'yellow', 'green', 'blue']

In [11]: colors2 = [item.upper() for item in colors]

In [12]: colors2
Out[12]: ['RED', 'ORANGE', 'YELLOW', 'GREEN', 'BLUE']
```

```
In [13]: colors
Out[13]: ['red', 'orange', 'yellow', 'green', 'blue']
```

✓ Self Check

1 *(Fill-In)* A list comprehension's ________ clause iterates over the specified sequence.
Answer: `for`.

2 *(Fill-In)* A list comprehension's ________ clause filters sequence elements to select only those that match a condition.
Answer: `if`.

3 *(IPython Session)* Use a list comprehension to create a list of tuples containing the numbers 1–5 and their cubes—that is, `[(1, 1), (2, 8), (3, 27), ...]`. To create tuples, place parentheses around the expression to the left of the list comprehension's `for` clause.
Answer:

```
In [1]: cubes = [(x, x ** 3) for x in range(1, 6)]

In [2]: cubes
Out[2]: [(1, 1), (2, 8), (3, 27), (4, 64), (5, 125)]
```

4 *(IPython Session)* Use a list comprehension and the `range` function with a step to create a list of the multiples of 3 that are less than 30.
Answer:

```
In [3]: multiples = [x for x in range(3, 30, 3)]

In [4]: multiples
Out[4]: [3, 6, 9, 12, 15, 18, 21, 24, 27]
```

5.13 Generator Expressions

A **generator expression** is similar to a list comprehension, but creates an iterable **generator object** that produces values *on demand*. This is known as **lazy evaluation**. List comprehensions use **greedy evaluation**—they create lists *immediately* when you execute them. For large numbers of items, creating a list can take substantial memory and time. So generator expressions can reduce your program's memory consumption and improve performance if the whole list is not needed at once.

Generator expressions have the same capabilities as list comprehensions, but you define them in parentheses instead of square brackets. The generator expression in snippet `[2]` squares and returns only the odd values in `numbers`:

```
In [1]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [2]: for value in (x ** 2 for x in numbers if x % 2 != 0):
   ...:     print(value, end='  ')
   ...:
9  49  1  81  25
```

To show that a generator expression does not create a list, let's assign the preceding snippet's generator expression to a variable and evaluate the variable:

```
In [3]: squares_of_odds = (x ** 2 for x in numbers if x % 2 != 0)
In [3]: squares_of_odds
Out[3]: <generator object <genexpr> at 0x1085e84c0>
```

The text "`generator object <genexpr>`" indicates that `square_of_odds` is a generator object that was created from a generator expression (`genexpr`).

Self Check

1 *(Fill-In)* A generator expression is ________—it produces values on demand.
Answer: lazy.

2 *(IPython Session)* Create a generator expression that cubes the even integers in a list containing 10, 3, 7, 1, 9, 4 and 2. Use function `list` to create a list of the results. Note that the function call's parentheses also act as the generator expression's parentheses.
Answer:

```
In [1]: list(x ** 3 for x in [10, 3, 7, 1, 9, 4, 2] if x % 2 == 0)
Out[1]: [1000, 64, 8]
```

5.14 Filter, Map and Reduce

The preceding section introduced several functional-style features—list comprehensions, filtering and mapping. Here we demonstrate the built-in `filter` and `map` functions for filtering and mapping, respectively. We continue discussing reductions in which you process a collection of elements into a *single* value, such as their count, total, product, average, minimum or maximum.

Filtering a Sequence's Values with the Built-In `filter` Function

Let's use built-in function **`filter`** to obtain the odd values in `numbers`:

```
In [1]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [2]: def is_odd(x):
   ...:     """Returns True only if x is odd."""
   ...:     return x % 2 != 0
   ...:

In [3]: list(filter(is_odd, numbers))
Out[3]: [3, 7, 1, 9, 5]
```

Like data, Python functions are objects that you can assign to variables, pass to other functions and return from functions. Functions that receive other functions as arguments are a functional-style capability called **higher-order functions**. For example, `filter`'s first argument must be a function that receives one argument and returns `True` if the value should be included in the result. The function `is_odd` returns `True` if its argument is odd. The `filter` function calls `is_odd` once for each value in its second argument's iterable (`numbers`). Higher-order functions may also return a function as a result.

Function `filter` returns an iterator, so `filter`'s results are not produced until you iterate through them. This is another example of lazy evaluation. In snippet `[3]`, function `list` iterates through the results and creates a list containing them. We can obtain the same results as above by using a list comprehension with an `if` clause:

```
In [4]: [item for item in numbers if is_odd(item)]
Out[4]: [3, 7, 1, 9, 5]
```

Using a `lambda` Rather than a Function

For simple functions like `is_odd` that `return` only a *single expression's value*, you can use a **lambda expression** (or simply a **lambda**) to define the function inline where it's needed—typically as it's passed to another function:

```
In [5]: list(filter(lambda x: x % 2 != 0, numbers))
Out[5]: [3, 7, 1, 9, 5]
```

We pass `filter`'s return value (an iterator) to function `list` here to convert the results to a list and display them.

A lambda expression is an *anonymous function*—that is, a *function without a name*. In the `filter` call

```
filter(lambda x: x % 2 != 0, numbers)
```

the first argument is the lambda

```
lambda x: x % 2 != 0
```

A lambda begins with the **`lambda`** keyword followed by a comma-separated parameter list, a colon (`:`) and an expression. In this case, the parameter list has one parameter named `x`. A `lambda` *implicitly* returns its expression's value. So any simple function of the form

```
def function_name(parameter_list):
    return expression
```

may be expressed as a more concise `lambda` of the form

```
lambda parameter_list: expression
```

Mapping a Sequence's Values to New Values

Let's use built-in function **`map`** with a `lambda` to square each value in `numbers`:

```
In [6]: numbers
Out[6]: [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [7]: list(map(lambda x: x ** 2, numbers))
Out[7]: [100, 9, 49, 1, 81, 16, 4, 64, 25, 36]
```

Function `map`'s first argument is a function that receives one value and returns a new value—in this case, a `lambda` that squares its argument. The second argument is an iterable of values to map. Function `map` uses lazy evaluation. So, we pass to the `list` function the iterator that `map` returns. This enables us to iterate through and create a list of the mapped values. Here's an equivalent list comprehension:

```
In [8]: [item ** 2 for item in numbers]
Out[8]: [100, 9, 49, 1, 81, 16, 4, 64, 25, 36]
```

Combining `filter` and `map`

You can combine the preceding `filter` and `map` operations as follows:

```
In [9]: list(map(lambda x: x ** 2,
    ...:          filter(lambda x: x % 2 != 0, numbers)))
    ...:
Out[9]: [9, 49, 1, 81, 25]
```

There is a lot going on in snippet `[9]`, so let's take a closer look at it. First, `filter` returns an iterable representing only the odd values of `numbers`. Then `map` returns an iterable representing the squares of the filtered values. Finally, `list` uses `map`'s iterable to create the list. You might prefer the following list comprehension to the preceding snippet:

```
In [10]: [x ** 2 for x in numbers if x % 2 != 0]
Out[10]: [9, 49, 1, 81, 25]
```

For each value of `x` in `numbers`, the expression `x ** 2` is performed only if the condition `x % 2 != 0` is `True`.

Reduction: Totaling the Elements of a Sequence with `sum`

As you know reductions process a sequence's elements into a single value. You've performed reductions with the built-in functions `len`, `sum`, `min` and `max`. You also can create custom reductions using the `functools` module's `reduce` function. See `https://docs.python.org/3/library/functools.html` for a code example. When we investigate big data and Hadoop (introduced briefly in Chapter 1), we'll demonstrate MapReduce programming, which is based on the filter, map and reduce operations in functional-style programming.

✓ Self Check

1 *(Fill-In)* ________, ________ and ________ are common operations used in functional-style programming.
Answer: Filter, map, reduce.

2 *(Fill-In)* A(n) ________ processes a sequence's elements into a single value, such as their count, total or average.
Answer: reduction.

3 *(IPython Session)* Create a list called `numbers` containing 1 through 15, then perform the following tasks:

a) Use the built-in function `filter` with a lambda to select only `numbers`' even elements. Create a new list containing the result.

b) Use the built-in function `map` with a lambda to square the values of `numbers`' elements. Create a new list containing the result.

c) Filter `numbers`' even elements, then map them to their squares. Create a new list containing the result.

Answer:

```
In [1]: numbers = list(range(1, 16))

In [2]: numbers
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

In [3]: list(filter(lambda x: x % 2 == 0, numbers))
```

```
Out[3]: [2, 4, 6, 8, 10, 12, 14]

In [4]: list(map(lambda x: x ** 2, numbers))
Out[4]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]

In [5]: list(map(lambda x: x**2, filter(lambda x: x % 2 == 0, numbers)))
Out[5]: [4, 16, 36, 64, 100, 144, 196]
```

4 *(IPython Session)* Map a list of the three Fahrenheit temperatures 41, 32 and 212 to a list of tuples containing the Fahrenheit temperatures and their Celsius equivalents. Convert Fahrenheit temperatures to Celsius with the following formula:

Celsius = (*Fahrenheit* - 32) * (5 / 9)

Answer:

```
In [6]: fahrenheit = [41, 32, 212]

In [7]: list(map(lambda x: (x, (x - 32) * 5 / 9), fahrenheit))
Out[7]: [(41, 5.0), (32, 0.0), (212, 100.0)]
```

The `lambda`'s expression—`(x, (x - 32) * 5 / 9)`—uses parentheses to create a tuple containing the original Fahrenheit temperature (x) and the corresponding Celsius temperature, as calculated by `(x - 32) * 5 / 9`.

5.15 Other Sequence Processing Functions

Python provides other built-in functions for manipulating sequences.

Finding the Minimum and Maximum Values Using a Key Function

We've previously shown the built-in reduction functions `min` and `max` using arguments, such as `int`s or lists of `int`s. Sometimes you'll need to find the minimum and maximum of more complex objects, such as strings. Consider the following comparison:

```
In [1]: 'Red' < 'orange'
Out[1]: True
```

The letter `'R'` "comes after" `'o'` in the alphabet, so you might expect `'Red'` to be less than `'orange'` and the condition above to be `False`. However, strings are compared by their characters' underlying *numerical values*, and lowercase letters have *higher* numerical values than uppercase letters. You can confirm this with built-in function **`ord`**, which returns the numerical value of a character:

```
In [2]: ord('R')
Out[2]: 82

In [3]: ord('o')
Out[3]: 111
```

Consider the list `colors`, which contains strings with uppercase and lowercase letters:

```
In [4]: colors = ['Red', 'orange', 'Yellow', 'green', 'Blue']
```

Let's assume that we'd like to determine the minimum and maximum strings using *alphabetical* order, not *numerical* (lexicographical) order. If we arrange colors alphabetically

```
'Blue', 'green', 'orange', 'Red', 'Yellow'
```

you can see that `'Blue'` is the minimum (that is, closest to the beginning of the alphabet), and `'Yellow'` is the maximum (that is, closest to the end of the alphabet).

Since Python compares strings using numerical values, you must first convert each string to all lowercase or all uppercase letters. Then their numerical values will also represent *alphabetical* ordering. The following snippets enable `min` and `max` to determine the minimum and maximum strings alphabetically:

```
In [5]: min(colors, key=lambda s: s.lower())
Out[5]: 'Blue'

In [6]: max(colors, key=lambda s: s.lower())
Out[6]: 'Yellow'
```

The `key` keyword argument must be a one-parameter function that returns a value. In this case, it's a `lambda` that calls string method **`lower`** to get a string's lowercase version. Functions `min` and `max` call the `key` argument's function for each element and use the results to compare the elements.

Iterating Backward Through a Sequence

Built-in function **`reversed`** returns an iterator that enables you to iterate over a sequence's values backward. The following list comprehension creates a new list containing the squares of `numbers`' values in reverse order:

```
In [7]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [7]: reversed_numbers = [item for item in reversed(numbers)]

In [8]: reversed_numbers
Out[8]: [36, 25, 64, 4, 16, 81, 1, 49, 9, 100]
```

Combining Iterables into Tuples of Corresponding Elements

Built-in function **`zip`** enables you to iterate over *multiple* iterables of data at the *same* time. The function receives as arguments any number of iterables and returns an iterator that produces tuples containing the elements at the same index in each. For example, snippet `[11]`'s call to `zip` produces the tuples `('Bob', 3.5)`, `('Sue', 4.0)` and `('Amanda', 3.75)` consisting of the elements at index 0, 1 and 2 of each list, respectively:

```
In [9]: names = ['Bob', 'Sue', 'Amanda']

In [10]: grade_point_averages = [3.5, 4.0, 3.75]

In [11]: for name, gpa in zip(names, grade_point_averages):
    ...:     print(f'Name={name}; GPA={gpa}')
    ...:
Name=Bob; GPA=3.5
Name=Sue; GPA=4.0
Name=Amanda; GPA=3.75
```

We unpack each tuple into `name` and `gpa` and display them. Function `zip`'s shortest argument determines the number of tuples produced. Here both have the same length.

Self Check

1 *(True/False)* The letter `'V'` "comes after" the letter `'g'` in the alphabet, so the comparison `'Violet' < 'green'` yields `False`.
Answer: False. Strings are compared by their characters' underlying numerical values. Lowercase letters have *higher* numerical values than uppercase. So, the comparison is `True`.

2 *(Fill-In)* Built-in function ________ returns an iterator that enables you to iterate over a sequence's values backward.
Answer: `reversed`.

3 *(IPython Session)* Create the list `foods` containing `'Cookies'`, `'pizza'`, `'Grapes'`, `'apples'`, `'steak'` and `'Bacon'`. Find the smallest string with `min`, then reimplement the `min` call using the `key` function to ignore the strings' case. Do you get the same results? Why or why not?
Answer: The `min` result was different because `'apples'` is the smallest string when you compare them without case sensitivity.

```
In [1]: foods = ['Cookies', 'pizza', 'Grapes',
   ...:          'apples', 'steak', 'Bacon']
   ...:

In [2]: min(foods)
Out[2]: 'Bacon'

In [3]: min(foods, key=lambda s: s.lower())
Out[3]: 'apples'
```

4 *(IPython Session)* Use `zip` with two integer lists to create a new list containing the sum of the elements from corresponding indices in both lists (that is, add the elements at index 0, add the elements at index 1, …).
Answer:

```
In [4]: [(a + b) for a, b in zip([10, 20, 30], [1, 2, 3])]
Out[4]: [11, 22, 33]
```

5.16 Two-Dimensional Lists

Lists can contain other lists as elements. A typical use of such nested (or multidimensional) lists is to represent **tables** of values consisting of information arranged in **rows** and **columns**. To identify a particular table element, we specify *two* indices—by convention, the first identifies the element's row, the second the element's column.

Lists that require two indices to identify an element are called **two-dimensional lists** (or **double-indexed lists** or **double-subscripted lists**). Multidimensional lists can have more than two indices. Here, we introduce two-dimensional lists.

Creating a Two-Dimensional List

Consider a two-dimensional list with three rows and four columns (i.e., a 3-by-4 list) that might represent the grades of three students who each took four exams in a course:

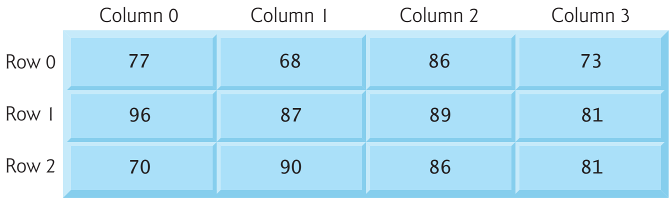
```
In [1]: a = [[77, 68, 86, 73], [96, 87, 89, 81], [70, 90, 86, 81]]
```

Writing the list as follows makes its row and column tabular structure clearer:

```
a = [[77, 68, 86, 73],  # first student's grades
     [96, 87, 89, 81],  # second student's grades
     [70, 90, 86, 81]]  # third student's grades
```

Illustrating a Two-Dimensional List

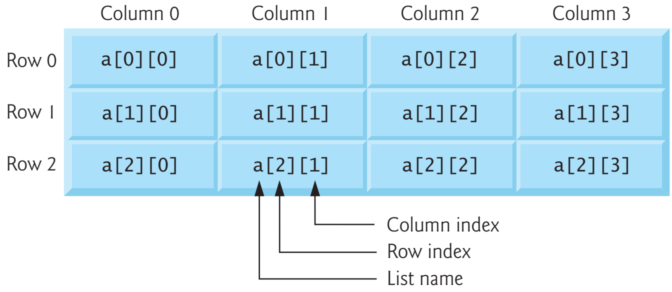
The diagram below shows the list a, with its rows and columns of exam grade values:



	Column 0	Column 1	Column 2	Column 3
Row 0	77	68	86	73
Row 1	96	87	89	81
Row 2	70	90	86	81

Identifying the Elements in a Two-Dimensional List

The following diagram shows the names of list a's elements:



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Every element is identified by a name of the form a[*i*][*j*]—a is the list's name, and *i* and *j* are the indices that uniquely identify each element's row and column, respectively. The element names in row 0 all have 0 as the first index. The element names in column 3 all have 3 as the second index.

In the two-dimensional list a:

- 77, 68, 86 and 73 initialize a[0][0], a[0][1], a[0][2] and a[0][3], respectively,
- 96, 87, 89 and 81 initialize a[1][0], a[1][1], a[1][2] and a[1][3], respectively, and
- 70, 90, 86 and 81 initialize a[2][0], a[2][1], a[2][2] and a[2][3], respectively.

A list with *m* rows and *n* columns is called an **m-by-n list** and has *m* × *n* elements.

The following nested for statement outputs the rows of the preceding two-dimensional list one row at a time:

```
In [2]: for row in a:
   ...:     for item in row:
   ...:         print(item, end=' ')
   ...:     print()
   ...:
77 68 86 73
96 87 89 81
70 90 86 81
```

How the Nested Loops Execute

Let's modify the nested loop to display the list's name and the row and column indices and value of each element:

```
In [3]: for i, row in enumerate(a):
   ...:     for j, item in enumerate(row):
   ...:         print(f'a[{i}][{j}]={item} ', end=' ')
   ...:     print()
   ...:
a[0][0]=77  a[0][1]=68  a[0][2]=86  a[0][3]=73
a[1][0]=96  a[1][1]=87  a[1][2]=89  a[1][3]=81
a[2][0]=70  a[2][1]=90  a[2][2]=86  a[2][3]=81
```

The outer `for` statement iterates over the two-dimensional list's rows one row at a time. During each iteration of the outer `for` statement, the inner `for` statement iterates over *each* column in the current row. So in the first iteration of the outer loop, `row` 0 is

```
[77, 68, 86, 73]
```

and the nested loop iterates through this list's four elements `a[0][0]=77`, `a[0][1]=68`, `a[0][2]=86` and `a[0][3]=73`.

In the second iteration of the outer loop, `row` 1 is

```
[96, 87, 89, 81]
```

and the nested loop iterates through this list's four elements `a[1][0]=96`, `a[1][1]=87`, `a[1][2]=89` and `a[1][3]=81`.

In the third iteration of the outer loop, `row` 2 is

```
[70, 90, 86, 81]
```

and the nested loop iterates through this list's four elements `a[2][0]=70`, `a[2][1]=90`, `a[2][2]=86` and `a[2][3]=81`.

In the "Array-Oriented Programming with NumPy" chapter, we'll cover the NumPy library's `ndarray` collection and the Pandas library's `DataFrame` collection. These enable you to manipulate multidimensional collections more concisely and conveniently than the two-dimensional list manipulations you've seen in this section.

✓ Self Check

1 *(Fill-In)* In a two-dimensional list, the first index by convention identifies the ________ of an element and the second index identifies the ________ of an element.
Answer: row, column.

2 *(Label the Elements)* Label the elements of the two-by-three list `sales` to indicate the order in which they're set to zero by the following program segment:

```
for row in range(len(sales)):
    for col in range(len(sales[row])):
        sales[row][col] = 0
```

Answer: `sales[0][0]`, `sales[0][1]`, `sales[0][2]`,
`sales[1][0]`, `sales[1][1]`, `sales[1][1]`.

3 *(Two-Dimensional Array)* Consider a two-by-three integer list `t`.

a) How many rows does `t` have?

b) How many columns does `t` have?

c) How many elements does `t` have?
d) What are the names of the elements in row 1?
e) What are the names of the elements in column 2?
f) Set the element in row 0 and column 1 to 10.
g) Write a nested `for` statement that sets each element to the sum of its indices.

Answer:

a) 2.
b) 3.
c) 6.
d) `t[1][0]`, `t[1][1]`, `t[1][2]`.
e) `t[0][2]`, `t[1][2]`.
f) `t[0][1] = 10`.
g)
```
for row in range(len(t)):
    for column in range(len(t[row])):
        t[row][column] = row + column
```

4 *(IPython Session)* Given the two-by-three integer list `t`

```
t = [[10, 7, 3], [20, 4, 17]]
```

a) Determine and display the average of `t`'s elements using nested `for` statements to iterate through the elements.
b) Write a `for` statement that determines and displays the average of `t`'s elements using the reductions `sum` and `len` to calculate the sum of each row's elements and the number of elements in each row.

Answer:

```
In [1]: t = [[10, 7, 3], [20, 4, 17]]

In [2]: total = 0

In [3]: items = 0

In [4]: for row in t:
   ...:     for item in row:
   ...:         total += item
   ...:         items += 1
   ...:

In [5]: total / items
Out[5]: 10.166666666666666

In [6]: total = 0

In [7]: items = 0

In [8]: for row in t:
   ...:     total += sum(row)
   ...:     items += len(row)
   ...:

In [9]: total / items
Out[9]: 10.166666666666666
```

5.17 Intro to Data Science: Simulation and Static Visualizations

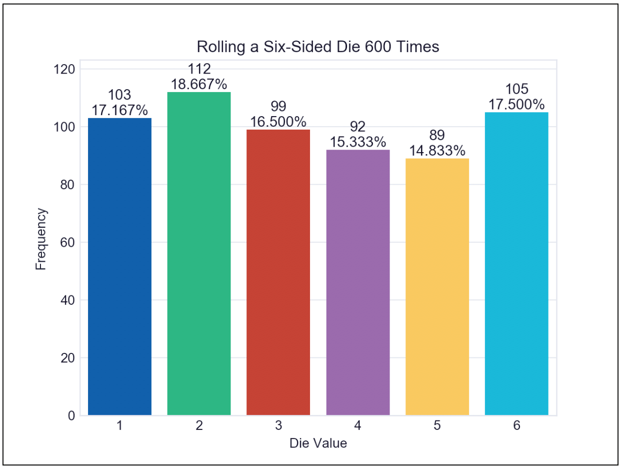
The last few chapters' Intro to Data Science sections discussed basic descriptive statistics. Here, we focus on visualizations, which help you "get to know" your data. Visualizations give you a powerful way to understand data that goes beyond simply looking at raw data.

We use two open-source visualization libraries—Seaborn and Matplotlib—to display *static* bar charts showing the final results of a six-sided-die-rolling simulation. The **Seaborn visualization library** is built over the **Matplotlib visualization library** and simplifies many Matplotlib operations. We'll use aspects of both libraries, because some of the Seaborn operations return objects from the Matplotlib library.

In the next chapter's Intro to Data Science section, we'll make things "come alive" with *dynamic visualizations*. In this chapter's exercises, you'll use simulation techniques and explore the characteristics of some popular card and dice games.

5.17.1 Sample Graphs for 600, 60,000 and 6,000,000 Die Rolls

The screen capture below shows a vertical bar chart that for 600 die rolls summarizes the frequencies with which each of the six faces appear, and their percentages of the total. Seaborn refers to this type of graph as a **bar plot**:



Here we expect about 100 occurrences of each die face. However, with such a small number of rolls, none of the frequencies is exactly 100 (though several are close) and most of the percentages are not close to 16.667% (about 1/6th). As we run the simulation for 60,000 die rolls, the bars will become much closer in size. At 6,000,000 die rolls, they'll appear to be exactly the same size. This is the "law of large numbers" at work. The next chapter will show the lengths of the bars changing dynamically.

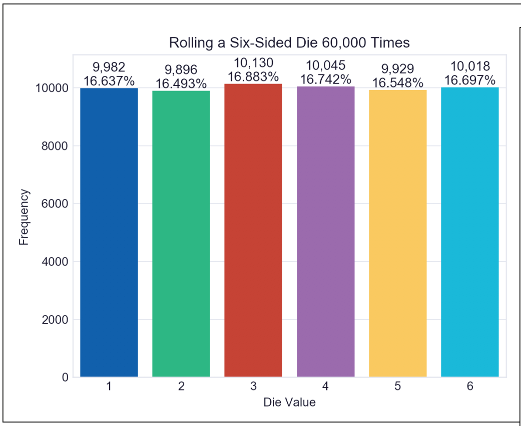
We'll discuss how to control the plot's appearance and contents, including:

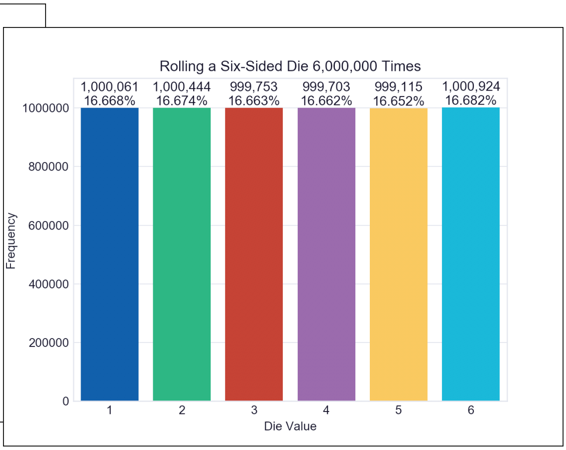
- the graph title inside the window (**Rolling a Six-Sided Die 600 Times**),
- the descriptive labels **Die Value** for the *x*-axis and **Frequency** for the *y*-axis,

- the text displayed above each bar, representing the *frequency* and *percentage* of the total rolls, and
- the bar colors.

We'll use various Seaborn default options. For example, Seaborn determines the text labels along the *x*-axis from the die face values 1–6 and the text labels along the *y*-axis from the actual die frequencies. Behind the scenes, Matplotlib determines the positions and sizes of the bars, based on the window size and the magnitudes of the values the bars represent. It also positions the **Frequency** axis's numeric labels based on the actual die frequencies that the bars represent. There are many more features you can customize. You should tweak these attributes to your personal preferences.

The first screen capture below shows the results for 60,000 die rolls—imagine trying to do this by hand. In this case, we expect about 10,000 of each face. The second screen capture below shows the results for 6,000,000 rolls—surely something you'd never do by hand![1] In this case, we expect about 1,000,000 of each face, and the frequency bars appear to be identical in length (they're close but not exactly the same length). Note that with more die rolls, the frequency percentages are much closer to the expected 16.667%.





Self Check

1 *(Discussion)* If you toss a coin a large odd number of times and associate the value 1 with heads and 2 with tails, what would you expect the mean to be? What would you expect the median and mode to be?

Answer: We'd expect the mean to be 1.5, which seems strange because it's not one of the possible outcomes. As the number of coin tosses increases, the percentages of heads and tails should each approach 50% of the total. However, at any given time, they are not likely to be identical. You're just as likely to have a few more heads than tails as vice versa, and as the number of rolls increases, the face with the larger number of rolls could change repeatedly. There are only two possible outcomes, so the median and mode values will be

[1] When we taught die rolling in our first programming book in the mid-1970s, computers were so much slower that we had to limit our simulations to 6000 rolls. In writing this book's examples, we went to 6,000,000 rolls, which the program completed in a few seconds. We then went to 60,000,000 rolls, which took about a minute.

whichever value there is more of at a given time. So if there are currently more heads than tails, both the median and mode will be heads; otherwise, they'll both be tails. Similar observations apply to die rolling.

5.17.2 Visualizing Die-Roll Frequencies and Percentages

In this section, you'll interactively develop the bar plots shown in the preceding section.

Launching IPython for Interactive Matplotlib Development

IPython has built-in support for interactively developing Matplotlib graphs, which you also need to develop Seaborn graphs. Simply launch IPython with the command:

```
ipython --matplotlib
```

Importing the Libraries

First, let's import the libraries we'll use:

```
In [1]: import matplotlib.pyplot as plt

In [2]: import numpy as np

In [3]: import random

In [4]: import seaborn as sns
```

1. The **`matplotlib.pyplot` module** contains the Matplotlib library's graphing capabilities that we use. This module typically is imported with the name `plt`.
2. The NumPy (Numerical Python) library includes the function `unique` that we'll use to summarize the die rolls. The **`numpy` module** typically is imported as `np`.
3. The `random` module contains Python's random-number generation functions.
4. The **`seaborn` module** contains the Seaborn library's graphing capabilities we use. This module typically is imported with the name `sns`. Search for why this curious abbreviation was chosen.

Rolling the Die and Calculating Die Frequencies

Next, let's use a *list comprehension* to create a list of 600 random die values, then use NumPy's **`unique`** function to determine the unique roll values (most likely all six possible face values) and their frequencies:

```
In [5]: rolls = [random.randrange(1, 7) for i in range(600)]

In [6]: values, frequencies = np.unique(rolls, return_counts=True)
```

The NumPy library provides the high-performance **`ndarray`** collection, which is typically much faster than lists.[2] Though we do not use `ndarray` directly here, the NumPy `unique` function expects an `ndarray` argument and returns an `ndarray`. If you pass a list (like `rolls`), NumPy converts it to an `ndarray` for better performance. The `ndarray` that `unique` returns we'll simply assign to a variable for use by a Seaborn plotting function.

Specifying the keyword argument **`return_counts`**`=True` tells `unique` to count each unique value's number of occurrences. In this case, `unique` returns a tuple of two one-

[2] We'll run a performance comparison in Chapter 7 where we discuss `ndarray` in depth.

dimensional `ndarrays` containing the sorted unique values and the corresponding frequencies, respectively. We unpack the tuple's `ndarrays` into the variables `values` and `frequencies`. If `return_counts` is `False`, only the list of unique values is returned.

Creating the Initial Bar Plot

Let's create the bar plot's title, set its style, then graph the die faces and frequencies:

```
In [7]: title = f'Rolling a Six-Sided Die {len(rolls):,} Times'

In [8]: sns.set_style('whitegrid')

In [9]: axes = sns.barplot(x=values, y=frequencies, palette='bright')
```

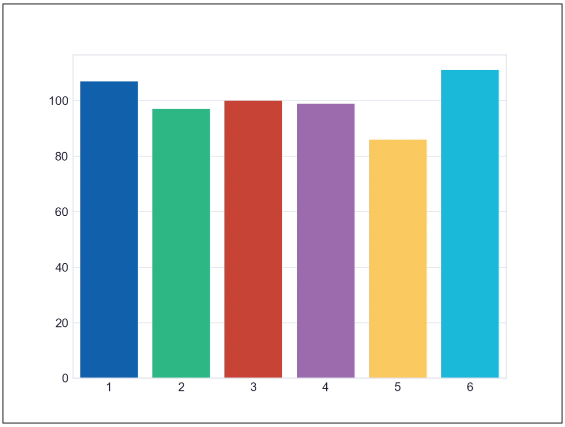
Snippet `[7]`'s f-string includes the number of die rolls in the bar plot's title. The comma (,) format specifier in

```
{len(rolls):,}
```

displays the number with *thousands separators*—so, 60000 would be displayed as 60,000.

By default, Seaborn plots graphs on a plain white background, but it provides several styles to choose from (`'darkgrid'`, `'whitegrid'`, `'dark'`, `'white'` and `'ticks'`). Snippet `[8]` specifies the `'whitegrid'` style, which displays light-gray horizontal lines in the vertical bar plot. These help you see more easily how each bar's height corresponds to the numeric frequency labels at the bar plot's left side.

Snippet `[9]` graphs the die frequencies using Seaborn's **`barplot` function**. When you execute this snippet, the following window appears (because you launched IPython with the `--matplotlib` option):



Seaborn interacts with Matplotlib to display the bars by creating a Matplotlib **`Axes`** object, which manages the content that appears in the window. Behind the scenes, Seaborn uses a Matplotlib **`Figure`** object to manage the window in which the `Axes` will appear. Function `barplot`'s first two arguments are `ndarrays` containing the *x*-axis and *y*-axis values, respectively. We used the optional `palette` keyword argument to choose Seaborn's predefined color palette `'bright'`. You can view the palette options at:

```
https://seaborn.pydata.org/tutorial/color_palettes.html
```

Function `barplot` returns the `Axes` object that it configured. We assign this to the variable `axes` so we can use it to configure other aspects of our final plot. Any changes you make to the bar plot after this point will appear *immediately* when you execute the corresponding snippet.

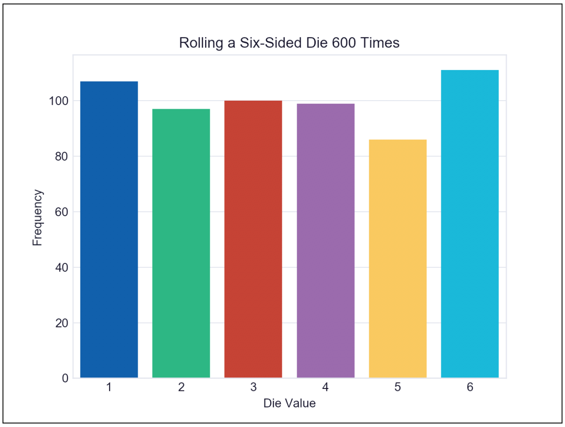
Setting the Window Title and Labeling the *x*- and *y*-Axes

The next two snippets add some descriptive text to the bar plot:

```
In [10]: axes.set_title(title)
Out[10]: Text(0.5,1,'Rolling a Six-Sided Die 600 Times')

In [11]: axes.set(xlabel='Die Value', ylabel='Frequency')
Out[11]: [Text(92.6667,0.5,'Frequency'), Text(0.5,58.7667,'Die Value')]
```

Snippet `[10]` uses the `axes` object's **`set_title`** method to display the `title` string centered above the plot. This method returns a `Text` object containing the title and its *location* in the window, which IPython simply displays as output for confirmation. You can ignore the `Out[]`s in the snippets above.

Snippet `[11]` add labels to each axis. The **`set`** method receives keyword arguments for the `Axes` object's properties to set. The method displays the `xlabel` text along the *x*-axis, and the `ylabel` text along the *y*-axis, and returns a list of `Text` objects containing the labels and their locations. The bar plot now appears as follows:



Finalizing the Bar Plot

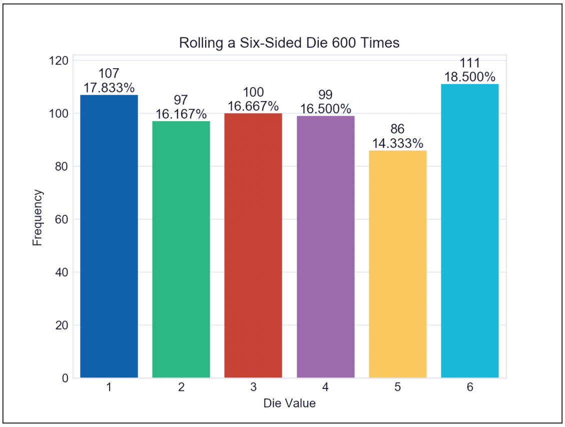
The next two snippets complete the graph by making room for the text above each bar, then displaying it:

```
In [12]: axes.set_ylim(top=max(frequencies) * 1.10)
Out[12]: (0.0, 122.10000000000001)

In [13]: for bar, frequency in zip(axes.patches, frequencies):
    ...:     text_x = bar.get_x() + bar.get_width() / 2.0
    ...:     text_y = bar.get_height()
    ...:     text = f'{frequency:,}\n{frequency / len(rolls):.3%}'
    ...:     axes.text(text_x, text_y, text,
    ...:               fontsize=11, ha='center', va='bottom')
    ...:
```

To make room for the text above the bars, snippet [12] scales the *y*-axis by 10%. We chose this value via experimentation. The Axes object's **set_ylim** method has many optional keyword arguments. Here, we use only top to change the maximum value represented by the *y*-axis. We multiplied the largest frequency by 1.10 to ensure that the *y*-axis is 10% taller than the tallest bar.

Finally, snippet [13] displays each bar's frequency value and percentage of the total rolls. The axes object's patches collection contains two-dimensional colored shapes that represent the plot's bars. The for statement uses zip to iterate through the patches and their corresponding frequency values. Each iteration unpacks into bar and frequency one of the tuples zip returns. The for statement's suite operates as follows:

- The first statement calculates the center *x*-coordinate where the text will appear. We calculate this as the sum of the bar's left-edge *x*-coordinate (bar.get_x()) and half of the bar's width (bar.get_width() / 2.0).
- The second statement gets the *y*-coordinate where the text will appear—bar.get_y() represents the bar's top.
- The third statement creates a two-line string containing that bar's frequency and the corresponding percentage of the total die rolls.
- The last statement calls the Axes object's **text** method to display the text above the bar. This method's first two arguments specify the text's *x–y* position, and the third argument is the text to display. The keyword argument ha specifies the *horizontal alignment*—we centered text horizontally around the *x*-coordinate. The keyword argument va specifies the *vertical alignment*—we aligned the bottom of the text with at the *y*-coordinate. The final bar plot is shown below:



Rolling Again and Updating the Bar Plot—Introducing IPython Magics

Now that you've created a nice bar plot, you probably want to try a different number of die rolls. First, clear the existing graph by calling Matplotlib's **cla** (clear axes) function:

```
In [14]: plt.cla()
```

IPython provides special commands called **magics** for conveniently performing various tasks. Let's use the **%recall magic** to get snippet [5], which created the rolls list, and place the code at the next In [] prompt:

```
In [15]: %recall 5

In [16]: rolls = [random.randrange(1, 7) for i in range(600)]
```

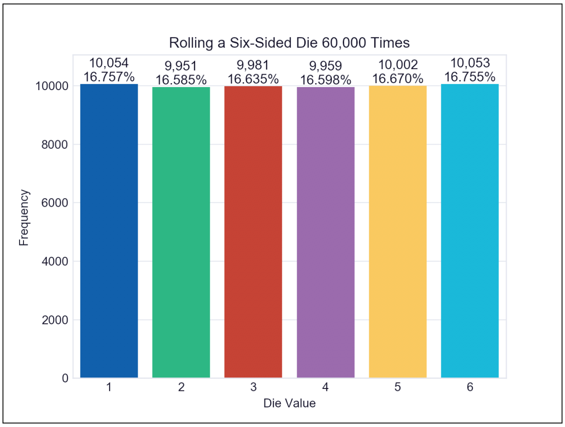
You can now edit the snippet to change the number of rolls to 60000, then press *Enter* to create a new list:

```
In [16]: rolls = [random.randrange(1, 7) for i in range(60000)]
```

Next, recall snippets [6] through [13]. This displays all the snippets in the specified range in the next In [] prompt. Press *Enter* to re-execute these snippets:

```
In [17]: %recall 6-13

In [18]: values, frequencies = np.unique(rolls, return_counts=True)
    ...: title = f'Rolling a Six-Sided Die {len(rolls):,} Times'
    ...: sns.set_style('whitegrid')
    ...: axes = sns.barplot(x=values, y=frequencies, palette='bright')
    ...: axes.set_title(title)
    ...: axes.set(xlabel='Die Value', ylabel='Frequency')
    ...: axes.set_ylim(top=max(frequencies) * 1.10)
    ...: for bar, frequency in zip(axes.patches, frequencies):
    ...:     text_x = bar.get_x() + bar.get_width() / 2.0
    ...:     text_y = bar.get_height()
    ...:     text = f'{frequency:,}\n{frequency / len(rolls):.3%}'
    ...:     axes.text(text_x, text_y, text,
    ...:               fontsize=11, ha='center', va='bottom')
    ...:
```

The updated bar plot is shown below:



Saving Snippets to a File with the %save Magic

Once you've interactively created a plot, you may want to save the code to a file so you can turn it into a script and run it in the future. Let's use the **%save magic** to save snippets 1 through 13 to a file named RollDie.py. IPython indicates the file to which the lines were written, then displays the lines that it saved:

```
In [19]: %save RollDie.py 1-13
The following commands were written to file `RollDie.py`:
import matplotlib.pyplot as plt
import numpy as np
import random
import seaborn as sns
rolls = [random.randrange(1, 7) for i in range(600)]
values, frequencies = np.unique(rolls, return_counts=True)
title = f'Rolling a Six-Sided Die {len(rolls):,} Times'
sns.set_style("whitegrid")
axes = sns.barplot(values, frequencies, palette='bright')
axes.set_title(title)
axes.set(xlabel='Die Value', ylabel='Frequency')
axes.set_ylim(top=max(frequencies) * 1.10)
for bar, frequency in zip(axes.patches, frequencies):
    text_x = bar.get_x() + bar.get_width() / 2.0
    text_y = bar.get_height()
    text = f'{frequency:,}\n{frequency / len(rolls):.3%}'
    axes.text(text_x, text_y, text,
              fontsize=11, ha='center', va='bottom')
```

Command-Line Arguments; Displaying a Plot from a Script

Provided with this chapter's examples is an edited version of the `RollDie.py` file you saved above. We added comments and a two modifications so you can run the script with an argument that specifies the number of die rolls, as in:

```
ipython RollDie.py 600
```

The Python Standard Library's **sys module** enables a script to receive *command-line arguments* that are passed into the program. These include the script's name and any values that appear to the right of it when you execute the script. The `sys` module's **argv** list contains the arguments. In the command above, `argv[0]` is the *string* `'RollDie.py'` and `argv[1]` is the *string* `'600'`. To control the number of die rolls with the command-line argument's value, we modified the statement that creates the `rolls` list as follows:

```
rolls = [random.randrange(1, 7) for i in range(int(sys.argv[1]))]
```

Note that we converted the `argv[1]` string to an `int`.

Matplotlib and Seaborn do not automatically display the plot for you when you create it in a script. So at the end of the script we added the following call to Matplotlib's **show** function, which displays the window containing the graph:

```
plt.show()
```

✓ Self Check

1 *(Fill-In)* The ________ format specifier indicates that a number should be displayed with thousands separators.
Answer: comma (,).

2 *(Fill-In)* A Matplotlib ________ object manages the content that appears in a Matplotlib window.
Answer: `Axes`.

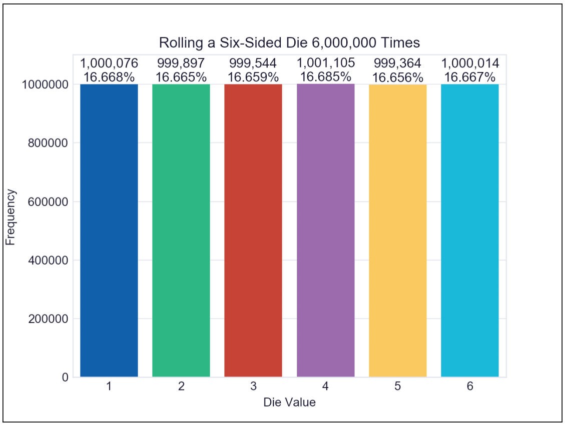
3 *(Fill-In)* The Seaborn function ________ displays data as a bar chart.
Answer: `barplot`.

4 *(Fill-In)* The Matplotlib function ________ displays a plot window from a script.
Answer: `show`.

5 *(IPython Session)* Use the `%recall` magic to repeat the steps in snippets `[14]` through `[18]` to redraw the bar plot for 6,000,000 die rolls. This exercise assumes that you're continuing this section's IPython session. Notice that the heights of the six bars look the same, although each frequency is close to 1,000,000 and each percentage is close to 16.667%.

Answer:

```
In [20]: plt.cla()

In [21]: %recall 5

In [22]: rolls = [random.randrange(1, 7) for i in range(6000000)]

In [23]: %recall 6-13

In [24]: values, frequencies = np.unique(rolls, return_counts=True)
    ...: title = f'Rolling a Six-Sided Die {len(rolls):,} Times'
    ...: sns.set_style('whitegrid')
    ...: axes = sns.barplot(values, frequencies, palette='bright')
    ...: axes.set_title(title)
    ...: axes.set(xlabel='Die Value', ylabel='Frequency')
    ...: axes.set_ylim(top=max(frequencies) * 1.10)
    ...: for bar, frequency in zip(axes.patches, frequencies):
    ...:     text_x = bar.get_x() + bar.get_width() / 2.0
    ...:     text_y = bar.get_height()
    ...:     text = f'{frequency:,}\n{frequency / len(rolls):.3%}'
    ...:     axes.text(text_x, text_y, text,
    ...:               fontsize=11, ha='center', va='bottom')
    ...:
```



5.18 Wrap-Up

This chapter presented more details of the list and tuple sequences. You created lists, accessed their elements and determined their length. You saw that lists are mutable, so you can modify their contents, including growing and shrinking the lists as your programs execute. You saw that accessing a nonexistent element causes an `IndexError`. You used `for` statements to iterate through list elements.

We discussed tuples, which like lists are sequences, but are immutable. You unpacked a tuple's elements into separate variables. You used `enumerate` to create an iterable of tuples, each with a list index and corresponding element value.

You learned that all sequences support slicing, which creates new sequences with subsets of the original elements. You used the `del` statement to remove elements from lists and delete variables from interactive sessions. We passed lists, list elements and slices of lists to functions. You saw how to search and sort lists, and how to search tuples. We used list methods to insert, append and remove elements, and to reverse a list's elements and copy lists.

We showed how to simulate stacks with lists—in an exercise, you'll use the same list methods to simulate a queue with a list. We used the concise list-comprehension notation to create new lists. We used additional built-in methods to sum list elements, iterate backward through a list, find the minimum and maximum values, filter values and map values to new values. We showed how nested lists can represent two-dimensional tables in which data is arranged in rows and columns. You saw how nested `for` loops process two-dimensional lists.

The chapter concluded with an Intro to Data Science section that presented a die-rolling simulation and static visualizations. A detailed code example used the Seaborn and Matplotlib visualization libraries to create a *static* bar plot visualization of the simulation's final results. In the next Intro to Data Science section, we use a die-rolling simulation with a *dynamic* bar plot visualization to make the plot "come alive."

In the next chapter, "Dictionaries and Sets," we'll continue our discussion of Python's built-in collections. We'll use dictionaries to store unordered collections of key–value pairs that map immutable keys to values, just as a conventional dictionary maps words to definitions. We'll use sets to store unordered collections of unique elements.

In the "Array-Oriented Programming with NumPy" chapter, we'll discuss NumPy's `ndarray` collection in more detail. You'll see that while lists are fine for small amounts of data, they are not efficient for the large amounts of data you'll encounter in big data analytics applications. For such cases, the NumPy library's highly optimized `ndarray` collection should be used. `ndarray` (*n*-dimensional array) can be much faster than lists. We'll run Python profiling tests to see just how much faster. As you'll see, NumPy also includes many capabilities for conveniently and efficiently manipulating arrays of *many* dimensions. In big data analytics applications, the processing demands can be humongous, so everything we can do to improve performance significantly matters. In our "Big Data: Hadoop, Spark, NoSQL and IoT" chapter, you'll use one of the most popular big-data databases—MongoDB.[3]

Exercises

Use IPython sessions for each exercise where practical.

5.1 *(What's Wrong with This Code?)* What, if anything, is wrong with each of the following code segments?

a)
```
day, high_temperature = ('Monday', 87, 65)
```

b)
```
numbers = [1, 2, 3, 4, 5]
numbers[10]
```

[3] The database's name is rooted in the word "humongous."

```
c) name = 'amanda'
   name[0] = 'A'
d) numbers = [1, 2, 3, 4, 5]
   numbers[3.4]
e) student_tuple = ('Amanda', 'Blue', [98, 75, 87])
   student_tuple[0] = 'Ariana'
f) ('Monday', 87, 65) + 'Tuesday'
g) 'A' += ('B', 'C')
h) x = 7
   del x
   print(x)
i) numbers = [1, 2, 3, 4, 5]
   numbers.index(10)
j) numbers = [1, 2, 3, 4, 5]
   numbers.extend(6, 7, 8)
k) numbers = [1, 2, 3, 4, 5]
   numbers.remove(10)
l) values = []
   values.pop()
```

5.2 *(What's Does This Code Do?)* What does the following function do, based on the sequence it receives as an argument?

```
def mystery(sequence):
    return sequence == sorted(sequence)
```

5.3 *(Fill in the Missing Code)* Replace the ***s in the following list comprehension and `map` function call, such that given a list of heights in inches, the code maps the list to a list of tuples containing the original height values and their corresponding values in meters. For example, if one element in the original list contains the height 69 inches, the corresponding element in the new list will contain the tuple `(69, 1.7526)`, representing both the height in inches and the height in meters. There are 0.0254 meters per inch.

```
[*** for x in [69, 77, 54]]
list(map(lambda ***, [69, 77, 54]))
```

5.4 *(Iteration Order)* Create a 2-by-3 list, then use a nested loop to:

a) Set each element's value to an integer indicating the order in which it was processed by the nested loop.

b) Display the elements in tabular format. Use the column indices as headings across the top, and the row indices to the left of each row.

5.5 *(IPython Session: Slicing)* Create a string called `alphabet` containing `'abcdefghijklmnopqrstuvwxyz'`, then perform the following separate slice operations to obtain:

a) The first half of the string using starting and ending indices.

b) The first half of the string using only the ending index.

c) The second half of the string using starting and ending indices.

d) The second half of the string using only the starting index.

e) Every second letter in the string starting with `'a'`.

f) The entire string in reverse.

g) Every third letter of the string in reverse starting with `'z'`.

5.6 *(Functions Returning Tuples)* Define a function `rotate` that receives three arguments and returns a tuple in which the first argument is at index `1`, the second argument is at index `2` and the third argument is at index `0`. Define variables `a`, `b` and `c` containing `'Doug'`, `22` and `1984`. Then call the function three times. For each call, unpack its result into `a`, `b` and `c`, then display their values.

5.7 *(Duplicate Elimination)* Create a function that receives a list and returns a (possibly shorter) list containing only the unique values in sorted order. Test your function with a list of numbers and a list of strings.

5.8 *(Sieve of Eratosthenes)* A prime number is an integer greater than 1 that's evenly divisible only by itself and 1. The Sieve of Eratosthenes is an elegant, straightforward method of finding prime numbers. The process for finding all primes less than 1000 is:

a) Create a 1000-element list `primes` with all elements initialized to `True`. List elements with prime indices (like 2, 3, 5, 7, 11, …) will remain `True`. All other list elements will eventually be set to `False`.

b) Starting with index 2, if a given element is `True` iterate through the rest of the list and set to `False` every element in `primes` whose index is a *multiple* of the index for the element we're currently processing. For list index 2, all elements beyond element 2 in the list that have indices which are multiples of 2 (i.e., 4, 6, 8, 10, …, 998) will be set to `False`.

c) Repeat Step (b) for the next `True` element. For list index 3 (which was initialized to `True`), all elements beyond element 3 in the list that have indices which are multiples of 3 (i.e., 6, 9, 12, 15, …, 999) will be set to `False`; and so on. A subtle observation (think about why this is true): The square root of 999 is 31.6, you'll need to test and set to `False` only all multiples of 2, 3, 5, 7, 9, 11, 13, 17, 19, 23, 29 and 31. This will significantly improve the performance of your algorithm, especially if you decide to look for large prime numbers.

When this process completes, the list elements that are still `True` indicate that the index is a prime number. These indices can be displayed. Use a list of 1000 elements to determine and display the prime numbers less than 1000. Ignore list elements 0 and 1. [As you work through the book, you'll discover other Python capabilities that will enable you to cleverly reimplement this exercise.]

5.9 *(Palindrome Tester)* A string that's spelled identically backward and forward, like `'radar'`, is a palindrome. Write a function `is_palindrome` that takes a string and returns `True` if it's a palindrome and `False` otherwise. Use a stack (simulated with a list as we did in Section 5.11) to help determine whether a string is a palindrome. Your function should ignore case sensitivity (that is, `'a'` and `'A'` are the same), spaces and punctuation.

5.10 *(Anagrams)* An anagram of a string is another string formed by rearranging the letters in the first. Write a script that produces all possible anagrams of a given string using only techniques that you've seen to this point in the book. [The `itertools` module provides many functions, including one that produces permutations.]

5.11 *(Summarizing Letters in a String)* Write a function `summarize_letters` that receives a string and returns a list of tuples containing the unique letters and their frequencies in the string. Test your function and display each letter with its frequency. Your function should ignore case sensitivity (that is, `'a'` and `'A'` are the same) and ignore spaces and punctuation. When done, write a statement that says whether the string has all the letters of the alphabet.

5.12 *(Telephone-Number Word Generator)* You should find this exercise to be entertaining. Standard telephone keypads contain the digits zero through nine. The numbers two through nine each have three letters associated with them, as shown in the following table:

Digit	Letters	Digit	Letters	Digit	Letters
2	A B C	5	J K L	8	T U V
3	D E F	6	M N O	9	W X Y
4	G H I	7	P R S		

Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words (or phrases) that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in the preceding table to develop the seven-letter word "NUMBERS." Every seven-letter word or phrase corresponds to exactly one seven-digit telephone number. A budding data science entrepreneur might like to reserve the phone number 244-3282 ("BIGDATA").

Every seven-digit phone number without 0s or 1s corresponds to many different seven-letter words, but most of these words represent unrecognizable gibberish. A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to the letters "PETCARE."

Write a script that, given a seven-digit number, generates every possible seven-letter word combination corresponding to that number. There are 2,187 (3^7) such combinations. Avoid phone numbers with the digits 0 and 1 (to which no letters correspond). See if your phone number corresponds to meaningful words.

5.13 *(Word or Phrase to Phone-Number Generator)* Just as people would enjoy knowing what word or phrase their phone number corresponds to, they might choose a word or phrase appropriate for their business and determine what phone numbers correspond to it. These are sometimes called vanity phone numbers, and various websites sell such phone numbers. Write a script similar to the one in the previous exercise that produces the possible phone number for the given seven-letter string.

5.14 *(Is a Sequence Sorted?)* Create a function `is_ordered` that receives a sequence and returns `True` if the elements are in sorted order. Test your function with sorted and unsorted lists, tuples and strings.

5.15 *(Tuples Representing Invoices)* When you purchase products or services from a company, you typically receive an invoice listing what you purchased and the total amount of money due. Use tuples to represent hardware store invoices that consist of four pieces of data—a part ID string, a part description string, an integer quantity of the item being purchased and, for simplicity, a `float` item price (in general, `Decimal` should be used for monetary amounts). Use the sample hardware data shown in the following table.

Part number	Part description	Quantity	Price
83	Electric sander	7	57.98
24	Power saw	18	99.99
7	Sledge hammer	11	21.50
77	Hammer	76	11.99
39	Jig saw	3	79.50

Perform the following tasks on a list of invoice tuples:

a) Use function sorted with a key argument to sort the tuples by part description, then display the results. To specify the element of the tuple that should be used for sorting, first import the itemgetter function from the operator module as in

```
from operator import itemgetter
```

Then, for sorted's key argument specify itemgetter(*index*) where *index* specifies which element of the tuple should be used for sorting purposes.

b) Use the sorted function with a key argument to sort the tuples by price, then display the results.

c) Map each invoice tuple to a tuple containing the part description and quantity, sort the results by quantity, then display the results.

d) Map each invoice tuple to a tuple containing the part description and the value of the invoice (the product of the quantity and the item price), sort the results by the invoice value, then display the results.

e) Modify Part (d) to filter the results to invoice values in the range $200 to $500.

f) Calculate the total of all the invoices.

5.16 *(Sorting Letters and Removing Duplicates)* Insert 20 random letters in the range 'a' through 'f' into a list. Perform the following tasks and display your results:

a) Sort the list in ascending order.

b) Sort the list in descending order.

c) Get the unique values sort them in ascending order.

5.17 *(Filter/Map Performance)* With regard to the following code:

```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

list(map(lambda x: x ** 2,
    filter(lambda x: x % 2 != 0, numbers)))
```

a) How many times does the filter operation call its lambda argument?

b) How many times does the map operation call its lambda argument?

c) If you reverse the filter and map operations, how many times does the map operation call its lambda argument?

To help you answer the preceding questions, define functions that perform the same tasks as the lambdas. In each function, include a print statement so you can see each time the function is called. Finally, replace the lambdas in the preceding code with the names of your functions.

5.18 *(Summing the Triples of the Even Integers from 2 through 10)* Starting with a list containing 1 through 10, use filter, map and sum to calculate the total of the triples of the even integers from 2 through 10. Reimplement your code with list comprehensions rather than `filter` and `map`.

5.19 *(Finding the People with a Specified Last Name)* Create a list of tuples containing first and last names. Use `filter` to locate the tuples containing the last name `Jones`. Ensure that several tuples in your list have that last name.

5.20 *(Display a Two-Dimensional List in Tabular Format)* Define a function named `display_table` that receives a two-dimensional list and displays its contents in tabular format. List the column indices as headings across the top, and list the row indices at the left of each row.

5.21 *(Computer-Assisted Instruction: Reducing Student Fatigue)* Re-implement Exercise 4.15 to store the computer's responses in lists. Use random-number generation to select responses using random list indices.

5.22 *(Simulating a Queue with a List)* In this chapter, you simulated a stack using a list. You also can simulate a queue collection with a list. **Queues** represent waiting lines similar to a checkout line in a supermarket. The cashier services the person at the *front* of the line *first*. Other customers enter the line only at the end and wait for service.

In a queue, you insert items at the back (known as the **tail**) and delete items from the front (known as the **head**). For this reason, queues are first-in, first-out (FIFO) collections. The insert and remove operations are commonly known as **enqueue** and **dequeue**.

Queues have many uses in computer systems, such as sharing CPUs among a potentially large number of competing applications and the operating system itself. Applications not currently being serviced sit in a queue until a CPU becomes available. The application at the front of the queue is the next to receive service. Each application gradually advances to the front as the applications before it receive service.

Simulate a queue of integers using list methods `append` (to simulate *enqueue*) and **`pop`** with the argument `0` (to simulate *dequeue*). Enqueue the values `3`, `2` and `1`, then dequeue them to show that they're removed in FIFO order.

5.23 *(Functional-Style Programming: Order of `filter` and `map` Calls)* When combining `filter` and `map` operations, the order in which they're performed matters. Consider a list `numbers` containing 10, 3, 7, 1, 9, 4, 2, 8, 5, 6 and the following code:

```
In [1]: numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]

In [2]: list(map(lambda x: x * 2,
   ...:          filter(lambda x: x % 2 == 0, numbers)))
   ...:
Out[3]: [20, 8, 4, 16, 12]
```

Reorder this code to call `map` first and `filter` second. What happens and why?

Exercises 5.24 through 5.26 are reasonably challenging. Once you've done them, you ought to be able to implement many popular card games.

5.24 *(Card Shuffling and Dealing)* In Exercises 5.24 through 5.26, you'll use lists of tuples in scripts that simulate card shuffling and dealing. Each tuple represents one card in the deck and contains a face (e.g., `'Ace'`, `'Deuce'`, `'Three'`, …, `'Jack'`, `'Queen'`, `'King'`) and a suit (e.g., `'Hearts'`, `'Diamonds'`, `'Clubs'`, `'Spades'`). Create an `initialize_deck` function to initialize the deck of card tuples with `'Ace'` through `'King'` of each suit, as in

```
deck = [('Ace', 'Hearts'), ..., ('King', 'Hearts'),
     ('Ace', 'Diamonds'), ..., ('King', 'Diamonds'),
     ('Ace', 'Clubs'), ..., ('King', 'Clubs'),
     ('Ace', 'Spades'), ..., ('King', 'Spades')]
```

Before returning the list, use the `random` module's `shuffle` function to randomly order the list elements. Output the shuffled cards in the following four-column format:

```
Six of Spades      Eight of Spades    Six of Clubs       Nine of Hearts
Queen of Hearts    Seven of Clubs     Nine of Spades     King of Hearts
Three of Diamonds  Deuce of Clubs     Ace of Hearts      Ten of Spades
Four of Spades     Ace of Clubs       Seven of Diamonds  Four of Hearts
Three of Clubs     Deuce of Hearts    Five of Spades     Jack of Diamonds
King of Clubs      Ten of Hearts      Three of Hearts    Six of Diamonds
Queen of Clubs     Eight of Diamonds  Deuce of Diamonds  Ten of Diamonds
Three of Spades    King of Diamonds   Nine of Clubs      Six of Hearts
Ace of Spades      Four of Diamonds   Seven of Hearts    Eight of Clubs
Deuce of Spades    Eight of Hearts    Five of Hearts     Queen of Spades
Jack of Hearts     Seven of Spades    Four of Clubs      Nine of Diamonds
Ace of Diamonds    Queen of Diamonds  Five of Clubs      King of Spades
Five of Diamonds   Ten of Clubs       Jack of Spades     Jack of Clubs
```

5.25 *(Card Playing: Evaluating Poker Hands)* Modify Exercise 5.24 to deal a five-card poker hand as a list of five card tuples. Then create functions (i.e., `is_pair`, `is_two_pair`, `is_three_of_a_kind`, …) that determine whether the hand they receive as an argument contains groups of cards, such as:

a) one pair
b) two pairs
c) three of a kind (e.g., three jacks)
d) a straight (i.e., five cards of consecutive face values)
e) a flush (i.e., all five cards of the same suit)
f) a full house (i.e., two cards of one face value and three cards of another)
g) four of a kind (e.g., four aces)
h) straight flush (i.e., a straight with all five cards of the same suit)
i) … and others.

See `https://en.wikipedia.org/wiki/List_of_poker_hands` for poker-hand types and how they rank with respect to one another. For example, three of a kind beats two pairs.

5.26 *(Card Playing: Determining the Winning Hand)* Use the methods developed in Exercise 5.25 to write a script that deals two five-card poker hands (i.e., two lists of five card tuples each), evaluates each hand and determines which wins. As each card is dealt, it should be removed from the list of tuples representing the deck.

5.27 *(Intro to Data Science: Duplicate Elimination and Counting Frequencies)* Use a list comprehension to create a list of 50 random values in the range 1 through 10. Use NumPy's `unique` function to obtain the unique values and their frequencies. Display the results.

5.28 *(Intro to Data Science: Survey Response Statistics)* Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being "awful" and 5 being "excellent." Place the 20 responses in a list

```
1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 5
```

Determine and display the frequency of each rating. Use the built-in functions, `statistics` module functions and NumPy functions demonstrated in Section 5.17.2 to display the following response statistics: minimum, maximum, range, mean, median, mode, variance and standard deviation.

5.29 *(Intro to Data Science: Visualizing Survey Response Statistics)* Using the list in Exercise 5.28 and the techniques you learned in Section 5.17.2, display a bar chart showing the response frequencies and their percentages of the total responses.

5.30 *(Intro to Data Science: Removing the Text Above the Bars)* Modify the die-rolling simulation in Section 5.17.2 to omit displaying the frequencies and percentages above each bar. Try to minimize the number of lines of code.

5.31 *(Intro to Data Science: Coin Tossing)* Modify the die-rolling simulation in Section 5.17.2 to simulate the flipping a coin. Use randomly generated 1s and 2s to represent heads and tails, respectively. Initially, do not include the frequencies and percentages above the bars. Then modify your code to include the frequencies and percentages. Run simulations for 200, 20,000 and 200,000 coin flips. Do you get approximately 50% heads and 50% tails? Do you see the "law of large numbers" in operation here?

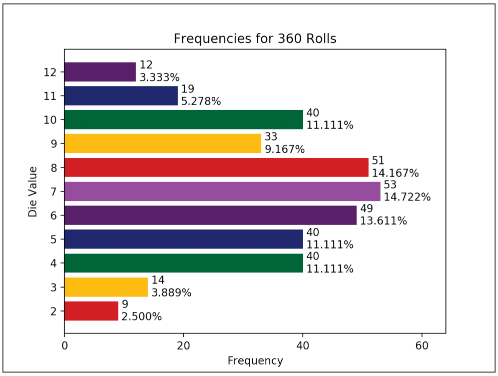
5.32 *(Intro to Data Science: Rolling Two Dice)* Modify the script `RollDie.py` that we provided with this chapter's examples to simulate rolling two dice. Calculate the sum of the two values. Each die has a value from 1 to 6, so the sum of the values will vary from 2 to 12, with 7 being the most frequent sum, and 2 and 12 the least frequent. The following diagram shows the 36 equally likely possible combinations of the two dice and their corresponding sums:

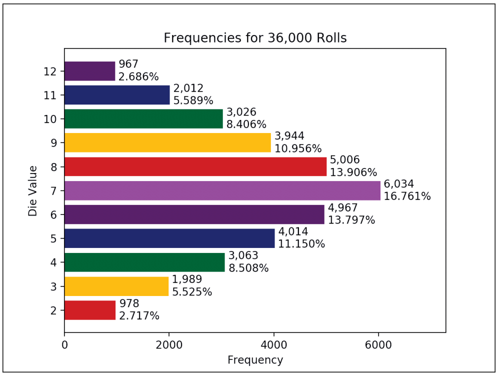
	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

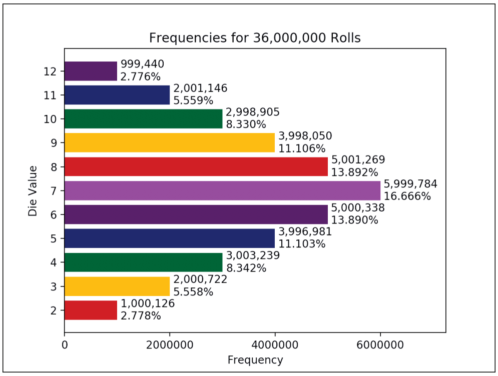
If you roll the dice 36,000 times:

- The values 2 and 12 each occur 1/36th (2.778%) of the time, so you should expect about 1000 of each.
- The values 3 and 11 each occur 2/36ths (5.556%) of the time, so you should expect about 2000 of each, and so on.

Use a command-line argument to obtain the number of rolls. Display a bar plot summarizing the roll frequencies. The following screen captures show the final bar plots for sample executions of 360, 36,000 and 36,000,000 rolls. Use the Seaborn `barplot` function's optional `orient` keyword argument to specify a horizontal bar plot.







5.33 *(Intro to Data Science Challenge: Analyzing the Dice Game Craps)* In this exercise, you'll modify Chapter 4's script that simulates the dice game craps by using the techniques you learned in Section 5.17.2. The script should receive a command-line argument indicating the number of games of craps to execute and use two lists to track the total numbers of games won and lost on the first roll, second roll, third roll, etc. Summarize the results as follows:

a) Display a horizontal bar plot indicating how many games are won and how many are lost on the first roll, second roll, third roll, etc. Since the game could continue indefinitely, you might track wins and losses through the first dozen rolls (of a pair of dice), then maintain two counters that keep track of wins and losses after 12 rolls—no matter how long the game gets. Create separate bars for wins and losses.

b) What are the chances of winning at craps? [*Note:* You should discover that craps is one of the fairest casino games. What do you suppose this means?]

c) What is the mean for the length of a game of craps? The median? The mode?

d) Do the chances of winning improve with the length of the game?