

## Chapter 3

# The Sieve of Eratosthenes

### *An algorithm for finding prime numbers*

Mathematicians who work in the field of number theory are interested in how numbers are related to one another. One of the key ideas in this area is how an integer can be expressed as the product of other integers. If an integer can only be written in product form as the product of 1 and the number itself it is called a **prime number**. Other numbers, known as **composite** numbers, are the product of two or more factors, for example,  $15 = 3 \times 5$  and  $12 = 2 \times 2 \times 3$ .

Ancient mathematicians devoted considerable attention to the subject. Over 2,000 years ago Euclid investigated several relationships among prime numbers, among other things proving there are an infinite number of primes. For most of their history, prime numbers were only of theoretical interest, but today they are at the heart of a variety of important computer applications. The security of messages transmitted using public key cryptography, the most widely used method for transferring sensitive information on the Internet, relies heavily on properties of prime numbers that were discovered thousands of years ago.

One of the fascinating things about prime numbers is the fact that there is no regular pattern to help predict which numbers will be prime. The number 839 is prime, and the next higher prime is 853, a distance of 14 numbers. But the next prime right after that is 857, only 4 numbers away. In some cases they appear in pairs, such as 881 and 883, where the difference between successive primes is only 2.

Because there is no pattern or rule to help us find prime numbers we need some way to systematically search for primes in a given range. A method for finding prime numbers that dates from the time of the ancient Greek mathematicians is known as the **Sieve of Eratosthenes**. The algorithm is very easy to understand, and even simpler to implement. We don't even need to know how to do addition or multiplication: all we have to do is count.

The project for this chapter is to implement the Sieve of Eratosthenes in Python. Our goal is to write a function named `sieve` that will make a list of all the prime numbers up to a specified limit. For example, if we want to know all the prime numbers less than 1,000, we just have to pass that number in a call to `sieve`:

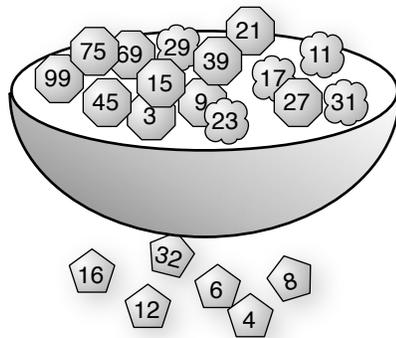
```
>>> sieve(1000)
[2, 3, 5, 7, 11, 13, , ... 983, 991, 997]
```

The main goal for this chapter, as it is in the other chapters in this book, is to understand the algorithm and the computation used to carry out the steps. We will start with an informal description and show how this method was used to make short lists of prime numbers for thousands of years using only paper and pencil. Working through the example raises an interesting question about the algorithm that needs to be resolved before we can get a computer to carry out the steps of the computation. Implementing the Sieve of Eratosthenes in Python and doing some experiments on our “computational workbench” also provides an opportunity to introduce some new programming language constructs and some new features of the PythonLabs software that will be used in later projects.

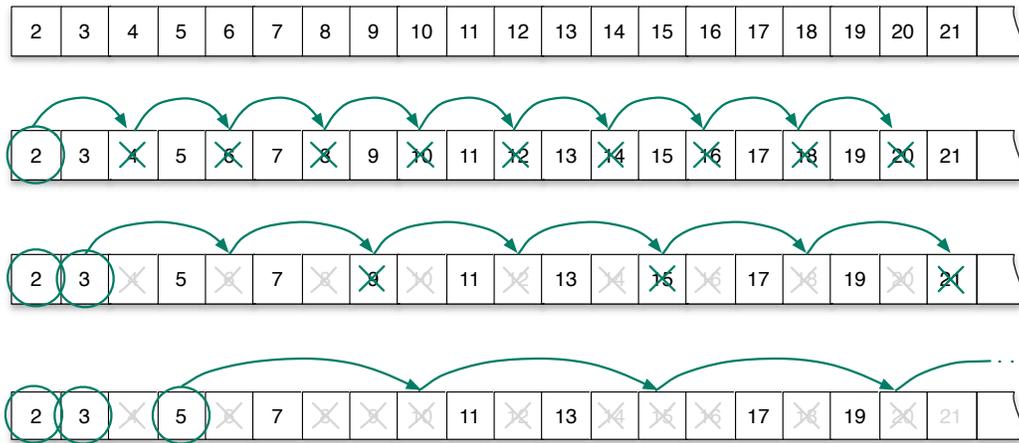
### 3.1 The Algorithm

To see why this algorithm is called a “sieve,” imagine numbers are rocks, where the shape of each rock is determined by its factors. Even numbers, which are multiples of 2, have a certain distinctive shape, multiples of 3 have a slightly different shape, and so on. Now suppose we have a magic bowl with adjustable holes in the bottom. To figure out which numbers are prime, put a bunch of rocks in the bowl and adjust the holes to match the shape for multiples of 2. When we shake the bowl, all the rocks that are multiples of 2 will fall through the holes (Figure 3.1). Next, adjust the holes so they match the shape for multiples of 3, and shake again so the multiples of 3 fall out. If the goal is to have only prime numbers in the bowl we need to keep repeating the adjusting and shaking steps until all the composite numbers have fallen out.

Instead of just starting with a random collection of numbers, and sifting out values in no particular order, the steps of the algorithm tell us how to proceed in a very precise manner.



**Figure 3.1:** The Sieve of Eratosthenes is a “magic bowl” that lets composite numbers fall through holes in the bottom. The first time the bowl is shaken, even numbers (multiples of 2) fall out.



**Figure 3.2:** To find prime numbers, write the integers starting from 2 on a strip of paper, then systematically cross off multiples of 2, multiples of 3, etc. until only primes are left.

As a result of taking a more systematic approach, we are guaranteed that when we're done we will have every prime number within a specified range.

Because we don't really have a magic bowl, we need to use paper and pencil or some other real technology. Figure 3.2 shows how we can carry out the steps using a long strip of paper. Start by listing all the integers, starting with 2 and extending as far as we want. The top row of the figure shows a "paper tape" with numbers from 2 to 21, but the tape can extend arbitrarily far to the right.

To begin the process of removing composites we want to remove multiples of 2. This is easy enough: starting at 2, move to the right two spaces at a time, crossing off every number we land on, as shown on the second row in Figure 3.2. When we are done, the first unmarked number to the right of 2 is 3. We now know 3 is prime, and we step through the list again, this time starting at 3 and moving to the right three spaces at a time.

For the next round the first unmarked number is 5, since 2 and 3 were marked as prime on previous rounds, and 4 was crossed off in the first round. So on the third round we begin at 5 and move to the right in steps of size 5 to remove the multiples of 5.

At the start of each remaining round, we can make the following general statements. Let's call the first unmarked number on the paper  $k$ . Then:

- $k$  is not a multiple of any of the numbers to its left (otherwise it would have been crossed off).
- Because all the numbers were written in order when we first created the list we now know  $k$  is prime.
- On this round we will start at  $k$  and move through the list in steps of size  $k$ .
- The first step takes us to  $k + k = 2k$ , the next lands on  $2k + k = 3k$ , and so on.
- We can cross off every number we land on because it will be a composite number of the form  $i \times k$  for some value of  $i$ .

The method is fairly straightforward, and it should be clear that if we have a large piece of paper and enough patience we can make a list of all primes up to 1,000 or 1,000,000 or any value we want. If we carefully follow the steps exactly as they are specified we are guaranteed to end up with a list of all primes between 2 and the last number on the paper.

The description above leaves out a very important detail: when is the list of primes complete? If you do a few more steps of the example in Figure 3.2, where the last number on the strip of paper is 21, you'll soon notice you aren't crossing out any more numbers. For example, after you "shake the bowl" to cross out multiples of 11, the only remaining numbers will be 13, 17, and 19. You could continue and search for multiples of 13, but you can tell at a glance there aren't any. The smallest composite number that is a multiple of 13 is  $2 \times 13 = 26$ , and this list only goes up to 21, so clearly there aren't any more multiples of 13 in the list. By the same reasoning, there aren't any multiples of 17 or 19 either, and all the numbers remaining on the paper are prime.

Simply saying "repeat until all the numbers left are prime" might be sufficient if we're telling another person how to make a list of prime numbers, but it is not specific enough to include as part of the definition of a Python function. If we want to implement this algorithm in Python we need a more precise specification of when the algorithm is finished.

We'll return to this question later in the chapter, but the first order of business is to figure out how to create lists of numbers in Python and how to scan lists to remove composite numbers.

## Tutorial Project

Take some time to make sure you understand how the Sieve of Eratosthenes works.

- T1. Use the algorithm to make a list of prime numbers less than 50. Write the numbers from 2 to 49 on a piece of paper, then do rounds of "sifting" until you are left with only prime numbers.
- T2. You can check your results with a version of `sieve` defined in the lab module for this chapter:.

```
>>> from PythonLabs.SieveLab import sieve
>>> sieve(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

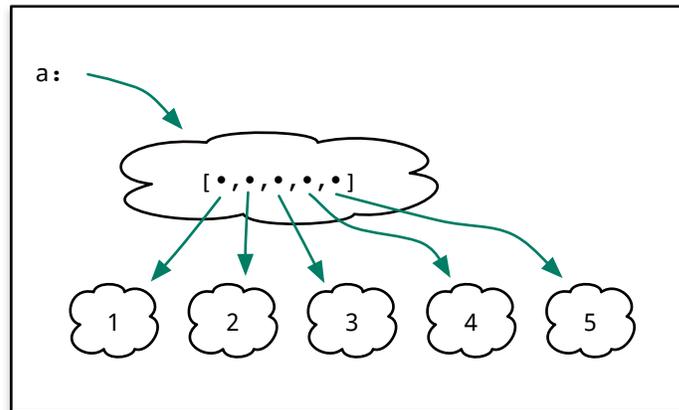
- ◆ How many passes did you make over the full list before you were left with all prime numbers? If we call the last number on the page  $n$ , can you derive a general formula in terms of  $n$  that describes a rule for when you can stop sifting?

## 3.2 Lists

In everyday life a **list** is a collection of items. Some lists are ordered, but many are just random collections of information. Ingredients in recipes are typically ordered because cooks like to have them presented in the order in which they are used, but shopping lists are often just random or semirandom collections of items (even if you're one of those super-organized people who arranges grocery lists by sections, the items within a section can be in any order; there's no reason to put radishes before cucumbers in the produce list).

Mathematicians also deal with collections. The idea of a **set** is one of the fundamental concepts in mathematics. Usually items in a set are not given in any particular order, but if order is important mathematicians say they have an "ordered set" or a "sequence."

**Figure 3.3:** The object store after creating a list of numbers. The variable *a* is a reference to a list object, which is a container that holds references to other objects.



Computer scientists have a wide variety of ways of organizing collections of data, including sets, graphs, trees, and many other structures. For this project, we just need the simple linear collection in the form of a list.

The simplest way to make a list in Python is to write the items in the list between square brackets, separated by commas. To define a variable named *a* that refers to a list of the numbers from 1 to 5 we simply write

```
>>> a = [1, 2, 3, 4, 5]
```

The statement above is an assignment statement. Python handles this assignment statement just like it did the assignments in the previous chapter: it creates an object to represent the expression on the right side, and the name on the left becomes a reference to the object. The only difference here is the value on the right side is a list of numbers.

In programming language terminology, an object that holds a collection of data items is known as a **container**. Figure 3.3 shows what the object store looks like after the assignment statement that defines *a* to be a reference to a list. The new variable is a reference to a container object, and the container holds references to the items that are part of the list.

As we did in the previous chapter, we can verify the result of an assignment by asking Python to print the value of the variable:

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

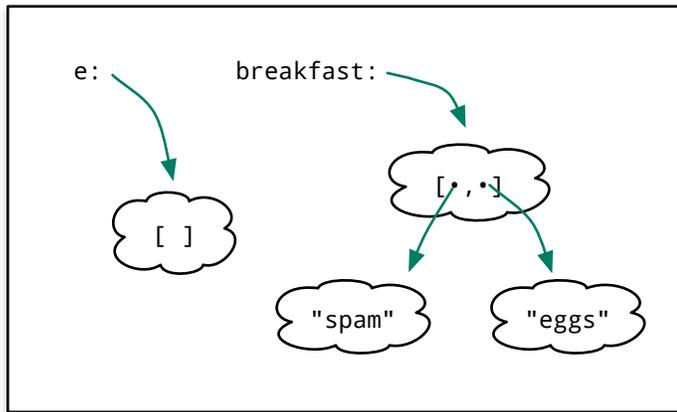
We can also ask Python to tell us what type of object *a* refers to:

```
>>> type(a)
<class 'list'>
```

Recall that “class” is the object-oriented programming terminology for “type.” From this output we can see that the name of this class in Python is `list`.

In the previous chapter we saw Python has a function named `len` that counts the number of characters in a string. The same function tells us how many items are in a list:

```
>>> len(a)
5
```



**Figure 3.4:** In this example the object store has two lists. The list named *e* is empty; it does not contain references to any other objects. The other list, named *breakfast*, holds references to two strings.

Note that it is possible to create a list that has nothing in it:

```
>>> e = []
```

This list is known as an **empty list**. It's analogous to the empty set in math, or, in real-life terms, to a three-ring binder with no paper. The binder is still a binder, even if it's empty. An empty list is an object, just like any other list object; it just doesn't contain any references to other objects. The list on the left in Figure 3.4 shows what an empty list might look like in the object store. Empty lists are very common in a wide variety of algorithms. An algorithm will often create an initially empty list, and add items to it in future steps, or start with a list of items and repeatedly delete items until the list is empty.

Python allows us to put any type of object in a list. This statement creates a list of strings:

```
>>> breakfast = ['spam', 'eggs']
```

Note what happens when we ask Python to compute the length of this list of words:

```
>>> len(breakfast)
2
```

Python did not count the number of letters in the strings. Instead, it counted the number of objects in the list, and because the list was created using two string objects the value returned by the call to `len` is 2 (Figure 3.4).

There are all sorts of interesting things we can do with our new lists. We can add items to the end, insert items in the middle, delete items, invert the order, and carry out dozens of other useful operations. Most of these operations are performed by calling methods defined in the `list` class. For example, to attach a new item to the end of a list, call a method named `append`:

```
>>> breakfast.append('sausage')
```

For the projects in this chapter, however, we only need to know how to create lists and access items in the list, so we'll put off the discussion of other methods until we need them in a project.

## Tutorial Project

T3. Start your IDE and type this statement in the shell window to create a list containing a few even numbers:

```
>>> a = [2, 4, 6, 8]
```

T4. Ask Python to show which class the new object belongs to:

```
>>> type(a)
<class 'list'>
```

T5. Use the len function to find out how many items are in the list:

```
>>> len(a)
4
```

T6. Add a new number to the end of the list:

```
>>> a.append(10)
```

T7. Did the previous expression change the length of the list? Call the len function to check your answer.

T8. Type this expression to make a new empty list named b:

```
>>> b = []
```

T9. Use the len function to find out how many elements are in b. Did you get 0?

T10. Make a list of strings:

```
>>> colors = ['green', 'yellow', 'black']
=> ['green', 'yellow', 'black']
```

T11. Verify there are three objects in this list:

```
>>> len(colors)
3
```

T12. Use the append method to add a new string to the end of colors:

```
>>> colors.append('steel')
```

T13. Use len to figure out how many items are now in colors.

Earlier in this section we learned that len, which we first encountered as a function that returns the number of characters in a string, also works for lists. Do you suppose the same thing is true of the + and \* operators? Can you predict what Python will do if we enter expressions using + or \* and the operands are lists?

T14. Do an experiment to test your prediction. Create two lists named a and b:

```
>>> a = [2,4,6]
>>> b = [1,2,3,4,5]
```

T15. Ask Python to evaluate some expressions with the + and \* operators:

```
>>> a + b
[2, 4, 6, 1, 2, 3, 4, 5]
>>> a * 3
[2, 4, 6, 2, 4, 6, 2, 4, 6]
>>> len((a + b) * 2)
16
```

Make sure you understand what Python did to evaluate each of the expressions above and how it evaluates similar expressions when a and b are strings instead of lists.

In Python strings and lists are special cases of a more general type of object called a **sequence**, which means operators and methods defined for lists often do a similar operation on strings, and vice versa.

### 3.3 Iterating Over Lists: for Statements

After a program has created a list to hold a collection of objects it often needs to carry out some calculation based on all the objects. An application might need to compute the average of a list of numbers, or search for a specific string in a list of words, or any number of similar operations.

The process of working through a list to perform some operation on every object is one form of a type of computation known as **iteration**. The word comes from the Latin word *iter*, which means “road” or “path.” We often talk about iterating over a collection of items, which means we start at the front and step through the collection, one item at a time.

The easiest way to specify this sort of iteration in Python is to use a construct called a **for loop**. The word “loop” has been part of computer programming jargon since the earliest programming languages. It refers to the idea that when the computer has finished executing a set of statements it should cycle back to the beginning and repeat the same operations again. When we are iterating over a list, we want the system to execute the statements for one item, then go back to the start of the loop and execute the same steps again for the next item in the list.

In Python, a for loop consists of a **for statement**, which controls how often the loop is executed, followed by one or more statements that make up the body of the loop. Two examples are shown in Figure 3.5. The `print_list` function uses a for loop to print every item in a list, and `total` computes the sum of a list of numbers.

The statement “for `x` in `a`” says we want to do something with every item in the list `a`. Python will assign `x` to the first item in the list and execute the statements in the body. It then goes back to the top of the loop, assigns `x` to the next item from the list, and executes the body again. This process repeats as long as there are still items in the list. The end result is that the statement in the body is executed once for each item, in order from the front of the list to the end.

The `total` function in Figure 3.5 uses a for loop to compute the sum of a list of numbers. The body of the loop uses a type of assignment statement we haven’t seen yet. The `+=` operator is a combination of addition and assignment. An expression of the form `n += m` means “add `m` to `n`.” The statement on line 4 is telling Python to add one of the numbers in the list to `sum`, a local variable that is keeping a running total of the values seen so far. The function starts by initializing `sum` to 0 and then uses the for loop to add each item in the list to `sum`. After the loop terminates the `sum` is returned as the value of the function call.

```
1 def print_list(a):
2     for x in a:
3         print(x)
```

```
1 def total(a):
2     sum = 0
3     for x in a:
4         sum += x
5     return sum
```

**Figure 3.5:** Examples of for loops. **(left)** A function that prints every item in a list. **(right)** A function that computes the sum of a list of numbers. Note that when there are two or more statements in the body of a loop they must all be indented the same number of spaces.

### Displaying Output in the Shell Window

Python has a built-in function named `print`. Whenever we call `print` we can pass it one or more objects as arguments and Python will display the objects in the shell window.

Here is an example that prints three integers:

```
>>> print(1,2,3)
1 2 3
```

The arguments passed to `print` can be any type of object. In this example `breakfast` refers to a list of strings:

```
>>> print(breakfast)
['spam', 'eggs', 'sausage']
```

This function isn't often used in interactive sessions since we can get the value of a variable by just typing its name, but printing is going to come in very handy when we start experimenting with more complicated algorithms. At various points in an algorithm we are going to want to call `print` to display the values of selected variables so we can keep track of how the algorithm is progressing.

## Tutorial Project

Use your IDE to create a new file named `lists.py` to use for experimenting with operations on lists.

T16. Add the definition of the `print_list` function of Figure 3.5 to `lists.py`, save the file, and then load it into an interactive shell session.

T17. Create a list to test your `print_list` function:

```
>>> colors = ['blue', 'yellow', 'black', 'green', 'red']
```

T18. Call your function, passing it the new list:

```
>>> print_list(colors)
blue
yellow
...
```

Did your function print every string in the test list? If not, compare your definition line by line with the function shown in Figure 3.5, paying particular attention to how the lines are indented.

The next set of experiments shows how the increment operator works.

T19. Initialize a variable and then use an assignment statement to update it to a new value:

```
>>> n = 4
>>> n = n + 1
>>> n
5
```

T20. Do another update, but this time use the increment operator:

```
>>> n += 1
>>> n
6
```

### Updating a Variable

The function that computes the mean of a list of numbers illustrates a common situation in computer programming. On each iteration of the loop we want to add another item from the list to the running total. One way to write this might be

```
sum = sum + x
```

This statement always appears strange to new programmers because it looks so much like an algebraic equation that does not make any sense. However, remember that the equal sign in Python is not a comparison operator, it is the assignment operator. This statement is processed just like any other assignment: Python looks for the current value of `sum` in its object store, adds `x` to that value, and overwrites the old value of `sum` with the new one.

Experienced Python programmers write the statement using an “augmented assignment” which does exactly the same thing:

```
sum += x
```

T21. When updating an integer, the operand on the right side of the operator can be any integer value:

```
>>> n += 3
>>> n
9
```

T22. Add the definition of the `total` function (Figure 3.5) to your `lists.py` file. Save the file and load the function into your shell session.

T23. Create a short list of numbers to test the function:

```
>>> a = [3,4,3,2,1]
```

T24. Python already has a built-in function named `sum` that computes the sum of a list of numbers. Use this function to show you the result you should expect when you test your `total` function:

```
>>> sum(a)
13
```

T25. Call `total` to compute the sum of numbers in your test list:

```
>>> total(a)
13
```

If you did not get 13 as the result, go back and check every line to make sure it is exactly as shown in Figure 3.5).

T26. What do you think your function will do if you pass it an empty list? Pass an empty list to `total` to see if your prediction is correct:

```
>>> total( [] )
```

## 3.4 List Indexes

Often when we write programs that use a list we need to do more than simply access every item in order. Some algorithms use only part of a list, while others need to access items out of order. These algorithms need a way to specify the locations of the items they use, allowing a programmer to write something that tells Python “fetch the second item in the list” or “set a variable  $m$  to the item in the middle of the list.”

Python and other programming languages use the idea of a **list index** to refer to a position within a list. An index for a list with  $n$  items is a number between 0 and  $n - 1$ . Note that the index of the first item is 0, not 1. This is a convention widely used in programming languages, but a common source of errors for new programmers who expect to use 1 as the first location.

To access an item in a list, simply write the name of the list followed by the location of the item, where the index is enclosed in “square brackets.” Here is an example, using a list of five strings:

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels[0]
'a'
>>> vowels[3]
'o'
```

### List Indexes

Computer scientists are famous for beginning at 0 instead of 1 when they start assigning labels to things. List indexes are a good example. If a list has 10 items, the locations in the list are labeled from 0 to 9.

If you type an expression with a list index operator and you see a result that doesn't look right, the first thing to check is to make sure you count starting at 0. If you type `a[1]` expecting to see the first string in the list shown below you'll get the wrong value. You need to remember to type `a[0]`.

```
>>> a = ['H', 'He', 'Li', 'Be', 'B', 'C']
>>> a[1]
'He'
>>> a[0]
'H'
>>> len(a)
6
>>> a[6]
IndexError: list index out of range
>>> a[5]
'C'
```

H	He	Li	Be	B	C
0	1	2	3	4	5

When we are reading an expression like `a[i]` out loud we say “a sub i.” The convention is derived from mathematical notation, where a sequence  $x$  is written  $x_1, x_2, \dots, x_n$ . The subscript notation from math and the square bracket notation used in computer programming are two different ways of expressing the idea that we are working with an ordered collection of items. Just remember that the first object in a list in Python is at location 0 instead of location 1.

If we know an item is in a list and we want Python to tell us where it is located we can call a method named `index`. If a list has  $n$  items, the value returned by a call to `index` will be a number between 0 and  $n - 1$ :

```
>>> vowels
['a', 'e', 'i', 'o', 'u']
>>> vowels.index('e')
1
>>> vowels.index('u')
4
```

If the specified item is not in the list, `index` generates an error:

```
>>> vowels.index('x')
ValueError: 'x' is not in list
```

In Chapter 2 we saw that the word `in` can be used as a Boolean operator to see if a string is contained inside another one. We can do the same thing here, to see if an item is contained in a list:

```
>>> 'e' in vowels
True
>>> 'x' in vowels
False
```

Note that both the `index` method and the `in` operator will search through a list to find an item. The difference is that `in` will return a Boolean value, depending on whether or not the item was found, and `index` returns an integer that tells us *where* an item was found, and it generates an error if the item is not in the list.

Another example of a function that iterates over a list is `partial_total`, shown in Figure 3.6. Like the `total` function used in a previous example, this function computes the sum of elements in a list, but in this case it uses only the values at the front of the list. The number of values to sum is specified by the first argument, so a call of the form `partial_total(n, a)` means “compute the sum of the first  $n$  items in  $a$ .”

```
>>> a
[3, 4, 3, 2, 1]
>>> partial_total(3, a)
10
```

Because the `partial_total` function is not iterating over the entire list we need to specify the part of the list we want to use. The technique shown in Figure 3.6 is to use a function named `range` to specify a set of index values. The expression `range(i, j)` means “the set of integers from  $i$  through  $j - 1$ .”

```
1 def partial_total(n,a):
2     """
3     Compute the total of the first
4     n items in list a.
5     """
6     sum = 0
7     for i in range(0,n):
8         sum += a[i]
9     return sum
```

Download: [sieve/partialtotal.py](#)

**Figure 3.6:** This function computes the sum of the values at the front of a list. The number of items to include in the sum is the first argument passed to the function. **Note:** There is a potential bug in this code (see Exercise T39 at the end of this section).

If we want to access the first  $n$  items in a list the for statement is

```
for i in range(0, n):
    # ... do something with a[i] ...
```

Notice how the assignment statement on line 4 of `partial_total` differs from the corresponding statement in `total` (Figure 3.5). If the loop is written as “for  $x$  in  $a$ ” then Python has already fetched a value from the list and saved it in  $x$ . But if the for statement uses a range expression to define values for an index variable we need to access the item in the list using an index expression, for example as  $a[i]$ .

We don’t need a list in order to use `range`; any situation where we need a sequence of integers is a good occasion to use a for loop with a range expression. Here is a loop that simply prints each number from 1 to 10 along with its square root:

```
for i in range(1,11):
    print(i, sqrt(i))
```

Notice that the last value of  $i$  will be 10 and not 11. This may seem awkward at first—if the function is going to make values between 1 and 10 why not just write `range(1, 10)`? As the example with list indexes showed, location numbers start with 0 and end with  $n - 1$ , and it is convenient to write `range(0, n)` when the values are going to be used to index into a list.

When we implement the Sieve of Eratosthenes we are going to want to iterate over a list of numbers using different step sizes. On the first round we will cross off every second number, on the next round every third number, and so on. For situations like this, Python allows us to pass a third argument to `range` that specifies the distance between successive values. Here is the previous for loop, but this time it only prints the square roots of odd numbers:

```
for i in range(1,11,2):
    print(i, sqrt(i))
```

Because this loop has a step size of 2, the variable  $i$  will be assigned 1, 3, 5, etc.

## Tutorial Project

T27. Make a small list of strings to use for testing index expressions:

```
>>> notes = ['do', 're', 'mi', 'fa', 'sol', 'la', 'ti']
```

T28. Use an index expression to get the first item in the list:

```
>>> notes[0]
'do'
```

T29. Because there are seven strings in this list we can find the last one at location 6:

```
>>> notes[6]
'ti'
```

T30. Here is another way to access the last item:

```
>>> notes[len(notes)-1]
'ti'
```

T31. Try asking Python for values at other locations, using any index between 0 and 6. Is the result what you expected?

T32. Repeat the experiment, but ask for an index that is past the end of the list; for example

```
>>> notes[12]
IndexError: list index out of range
```

T33. Next try some expressions using `in`, which should evaluate to `True` or `False`, depending on whether the list contains the specified item:

```
>>> 're' in notes
True
>>> 'bzzt' in notes
False
```

T34. Use Python's `index` method to find out where the items are located:

```
>>> notes.index('re')
1
>>> notes.index('bzzt')
ValueError: 'bzzt' is not in list
```

Do you see why the first expression above returned a value of 1?

The next set of exercises asks you to type a `for` loop in your shell session. After you type the first line, your prompt should change to an ellipsis (three dots) to show you that Python is expecting you to continue something you started on an earlier line. Don't forget to type spaces before each line in the body of the loop. After the last line simply hit the return key to end the loop.

T35. Use `range` to iterate over the `notes` list and print something about each note:

```
>>> for i in range(0, len(notes)):
...     print(notes[i], 'has', len(notes[i]), 'letters')
...
do has 2 letters
re has 2 letters
...

```

Make sure you understand exactly why Python prints what it does.

T36. Enter the for loop that prints the square roots of numbers from 1 to 10 (don't forget to import the `sqrt` function if you haven't done it already):

```
>>> from math import sqrt
>>> for i in range(1,11):
...     print(i, sqrt(i))
...
1 1.0
2 1.4142135623730951
...
```

T37. Repeat the previous exercise, but this time use `range(1, 11, 2)`. Did Python execute the body of the loop only for odd values of `i`? Do you see why?

T38. Either download the definition of `partial_total` (Figure 3.6) or add the definition to your `lists.py` file. Load the definition into your interactive session and call it a few times (using the same list you used to test `total`) to verify it works.

T39. What happens if you pass a number to `partial_total` that is larger than the number of items in the list? For example, what does Python do if `a` has five items and you call `partial_total(7, a)`?

- ◆ Add statements to your definition of `partial_total` so the function returns a correct value even if `n` is too large. If `n` is greater than the number of items in the list, simply return the sum of all the items.

## 3.5 The Animated Sieve

Now that we know how to create lists of integers in Python we are ready to start writing a function that uses the Sieve of Eratosthenes to create a list of prime numbers. As a first step we are going to take a closer look at how the sieve works, using a function included as part of `PythonLabs`.

The projects in this section use a technique known as **algorithm animation** to help visualize the steps of a computation. Animated algorithms draw 2D pictures in a graphics window called a **canvas** and then update the drawing as the algorithm proceeds. In future chapters we will use animations to illustrate how objects are moved around by different sort algorithms, show the motion of planets during a solar system simulation, and visualize many other projects throughout the book.

To use the animated version of the sieve we need to import the functions with this statement:

```
>>> from PythonLabs.SieveLab import *
```

The asterisk here means “everything,” so this statement imports all the functions defined in `SieveLab`.

Because the lists we will experiment with are too long to fit on a single line they will be displayed on the canvas as a set of rows and columns, in a format a person might use when writing the numbers on a piece of paper, and we will refer to the list of numbers as the “worksheet.” But regardless of whether the numbers are written on a single line or in a grid, the basic operation is the same: when crossing off multiples of some number  $k$  we want to iterate over the set of numbers using steps of size  $k$ .

### Constructors

In Python we can use class names as functions. In the previous chapter we learned that Python's string class is named `str`, and this chapter introduced lists, which belong to a class named `list`. If we use these names as functions Python will create an object of that type. For `str` and `list`, Python creates an empty string or an empty list:

```
>>> str()
''
>>> list()
[]
```

We can also pass arguments to these functions if we want the new object to be initialized. For example, to make a list of numbers, pass a range expression to `list`:

```
>>> list(range(0,10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In object-oriented programming terminology these functions that create objects of a specified class are known as *constructors*. We will learn more about constructors in Chapter 7, when we learn how to define our own classes.

Recall that the first step in the algorithm is to create a list of integers, starting with 2. Fortunately, Python has several features that will help us make the list without having to type a long list of numbers. One way is to use the name `list` as a function, passing it a range expression to tell it to make a list containing all the values in that range. For example, this is how we can make a list of numbers up to but not including 10:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is an example of a more general technique where we use a class name as a function in order to create an instance of that class (for more see the sidebar on “Constructors”).

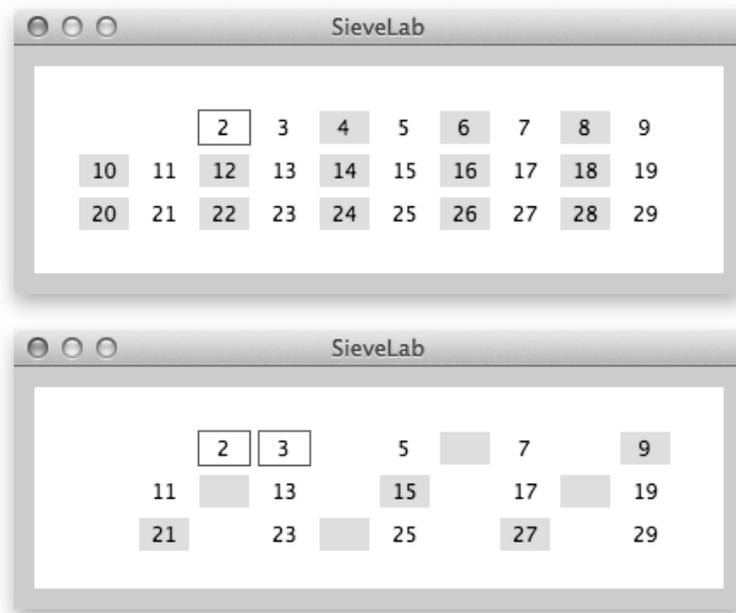
If we want to find prime numbers less than 100, a statement that creates a list with numbers between 2 and 99 is

```
>>> list(range(2,99))
[2, 3, 4, 5, ... 97, 98]
```

But notice that because 2 is the first item it is at location 0. The number 3 is at location 1, and in general, any number  $i$  is at location  $i - 2$ .

We can avoid some potential confusion if we add two “placeholder” values at the front of the list. What we want is a list that looks like this:

```
[?, ?, 2, 3, 4, ... ]
```



**Figure 3.7:** *Top:* A snapshot of the worksheet during the first iteration of the Sieve of Eratosthenes, after 2 has been identified as a prime number and its multiples have been marked. *Bottom:* After multiples of 2 are erased, the second iteration marks multiples of 3.

If we can figure out what to put in place of the question marks we will have what we want: the number 2 is in location 2, 3 is in location 3, and so on.

A natural candidate for the two placeholder values is a special Python object named `None`. This object is often used to introduce a variable before we know what value we want to assign it, or in situations where we need space for something.

We saw earlier that the `+` operator means “concatenate” when we apply it to two lists. That means we can initialize the worksheet by creating a list with two `None` objects and then “adding” the list of numbers beginning with 2. Putting all these pieces together, here is the statement that initializes our worksheet with numbers from 2 to 99:

```
>>> worksheet = [None, None] + list(range(2,100))
```

If we print the value of this variable we can see it’s exactly what we want:

```
>>> worksheet
[None, None, 2, 3, 4, 5, ... 97, 98, 99]
```

To start the animation, pass the initial worksheet in a call to `view_sieve`, one of the functions imported from `SieveLab`:

```
>>> view_sieve(worksheet)
```

If the canvas is not already on the screen, a new window will open up and the system will draw the numbers in the list in a rectangular grid.

For this interactive session, where we're using visualization to explore the algorithm, the operations performed on each round are split into two separate functions. When we write the Python code later in this chapter all the steps will be in a single function, but in SieveLab the steps have been separated into two different functions for better control of the interactive visualizations.

A function named `mark_multiples` will highlight a number and all of its multiples. On the first round we need to cross off multiples of 2, so this is what we type:

```
>>> mark_multiples(2, worksheet)
```

After Python returns from this function the canvas should look something like the snapshot in the top half of Figure 3.7 (to save space only the top three rows of the worksheet are shown). Notice how the box for the number 2 is highlighted and the boxes for all the multiples of 2 have a gray background to show they will be deleted.

To continue, call a second function that removes the marked numbers from the worksheet:

```
>>> erase_multiples(2, worksheet)
```

All of the grayed-out numbers will be erased from the canvas (as seen in the snapshot in the lower half of Figure 3.7).

We can repeat the calls to these two functions as many times as we'd like. Before each call to `mark_multiples`, simply look for the next number following the most recently identified prime. After removing multiples of 2 we can see the next prime is 3. Note that the next prime after 3 will be 5, because 4 will be filtered out as a multiple of 2.

Let's return now to the question of how we will be able to tell when only prime numbers are left on the worksheet. The process described in the introduction said "repeat sifting until it's clear there are no composites left." The example given at the time was that we don't have to sift out multiples of 13 if the list only goes up to 21 because the lowest possible multiple of 13 is  $2 \times 13$ , or 26. Now that we want to automate the process we need to be more precise about the terminating condition so we can implement the algorithm in Python.

It turns out we can stop the marking and erasing process as soon as the first unmarked number is greater than  $\sqrt{n}$ . Every composite number less than  $n$  will have at least one prime factor  $p$  less than  $\sqrt{n}$ , which means the number will have been crossed off when we filter out the multiples of  $p$  (the sidebar on "Prime Factors" on page 73 explains why).

In the example used in the tutorial project for this section, where  $n = 100$ , we can stop filtering after we remove the multiples of 7. After removing multiples of 7 the next unmarked number will be 11, and because it's greater than  $\sqrt{100} = 10$  all the numbers left on the worksheet will be prime.

### ◆ Prime Factors

Every composite number can be written as the product of two smaller numbers. If either of those numbers is itself composite it can also be written as a product. We can continue this process (which is known as **factoring**) until each smaller number is prime.

At least one of the prime factors of a number  $k = i \times j$  must be less than or equal to  $\sqrt{k}$ . It's easy to see why: if both  $i$  and  $j$  are greater than  $\sqrt{k}$  then their product would be larger than  $k$  because  $i \times j > \sqrt{k} \times \sqrt{k} = k$ .

When looking for primes less than  $n$ , every composite number between 2 and  $n$  always has one prime factor  $p < \sqrt{n}$  so the number will be removed when we sift multiples of  $p$ . That means the Sieve of Eratosthenes can stop sifting when the smallest unmarked number on the worksheet is greater than  $\sqrt{n}$ .

As an example, suppose we are looking for primes less than 1000. The stopping point is  $\sqrt{1000} \approx 31.62$ . After we filter multiples of 31, the next number on the worksheet is 37, and we are done.

$$65 = 5 \times 13$$

$$147 = 3 \times 7 \times 7$$

$$308 = 2 \times 2 \times 7 \times 11$$

$$793 = 13 \times 61$$

$$861 = 3 \times 7 \times 41$$

$$901 = 17 \times 53$$

$$961 = 31 \times 31$$

*When  $n = 1000$  every composite number has at least one prime factor smaller than 31.62*

## Tutorial Project

T40. Import the functions defined in SieveLab:

```
>>> from PythonLabs.SieveLab import *
```

T41. Pass a range to the list constructor to see how it can be used to create a list of numbers:

```
>>> list(range(0,10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

T42. Note how the last number in the list is one less than the upper limit passed to range. Try making a few more lists with different starting and ending values to make sure you understand how the constructor works.

T43. Make a list of odd integers:

```
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
```

Do you see why this list contains every other number starting with 1?

T44. Do some experiments with the + operator to verify it concatenates two lists:

```
>>> a = ['do', 're', 'mi']
>>> b = ['fa', 'sol', 'la']
>>> a + b
['do', 're', 'mi', 'fa', 'sol', 'la']
```

T45. Initialize the worksheet with integers less than 100:

```
>>> worksheet = [None, None] + list(range(2,100))
```

T46. Display the worksheet on the PythonLabs canvas. You should see a 10 x 10 grid:

```
>>> view_sieve(worksheet)
```

T47. Call mark\_multiples to highlight the multiples of 2:

```
>>> mark_multiples(2, worksheet)
```

Is the number 2 circled? Are all the multiples of 2 grayed out?

T48. Erase the multiples of 2. All the grayed-out numbers should disappear:

```
>>> erase_multiples(2, worksheet)
```

T49. Mark the multiples of 3:

```
>>> mark_multiples(3, worksheet)
```

If your IDE supports “command line editing” this is a good place to use it. You can save yourself some typing if you hit the up-arrow key twice, change the 2 to a 3, and hit the return key.

T50. Notice how every third square is gray, including locations that contained numbers that were erased in the previous step.

T51. Erase the multiples of 3:

```
>>> erase_multiples(3, worksheet)
```

T52. Do you see how 5 is next number to sift out? It’s the lowest unmarked number because 4, which is a multiple of 2, was removed in the first round.

T53. Keep repeating the marking and erasing steps until all the numbers left on the worksheet are prime numbers. Note that one way to tell there are no composite numbers left is when all the cells grayed out by a call to `mark_multiples` have already been erased on a previous round.

T54. How many rounds of marking and erasing did you do before you decided there were no more composite numbers left?

T55. Call `sieve(100)` to get the list of primes and compare this list with the worksheet on your canvas. Do they agree?

The SieveLab implementation of `sieve` will draw the worksheet and show the steps of the algorithm on the canvas if we pass it some additional arguments.

T56. The following command shows how to call `sieve` and have it display the worksheet as a  $25 \times 40$  grid:

```
>>> sieve(1000, vis=True, nrows=25, ncols=40, delay=1.0)
```

The last argument determines the speed of the animation. If you want the program to pause for 2 seconds after each marking and erasing step use `delay=2.0`.

T57. Is the final display of the list of primes less than 1000 what you expected? Can you see why the last number circled on the canvas is 31?

## 3.6 A Helper Function

The animations in the previous section show what we need to do when we write our own Python program to implement the Sieve of Eratosthenes. We’ve already seen how to initialize the worksheet; now we need to figure out how to remove composite numbers.

We’re going to take an approach commonly used by programmers and break the problem into two smaller sub-problems. In this section we will write a function named `sift` that will implement a single round of finding and removing multiples. For example, if we call `sift(5, worksheet)` we will expect it to scan through the list of numbers to remove all the multiples of 5. When we are satisfied this function is working correctly, we will turn our attention to the main part of the algorithm, which will have a loop that repeatedly calls `sift` to systematically remove all the composite numbers.

The `sift` function has a very specific purpose, and it is unlikely we would ever want to use it for anything other than as part of a project that implements the Sieve of Eratosthenes.

Programmers write these sorts of specialized functions as a way of focusing on a single aspect of a complex problem, allowing them to design and test the code for each piece separately. A function like `sift` that is intended only to be called for one special purpose is called a **helper function**. After we implement and test our helper function we will use it in the next section as we implement our own version of `sieve`.

The Python code for the function will start with a `def` statement that looks like this:

```
def sift(k, a):
```

When the function is called, the value of a newly discovered prime number will be in `k`, and `a` will be a reference to the main worksheet list.

The first question we need to consider is what it means to “cross off” a number. When we used paper and pencil it was sufficient to draw a line through a number or erase it. Now that we are using a Python list to hold the numbers on the worksheet we need to figure out how to record the fact that a number is composite.

It’s tempting to use a statement that removes numbers from the list. Python has a statement named `del` (short for “delete”) that does just that. Here is an example that shows a list and what it looks like after using `del` to remove the middle item:

```
>>> greeks = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> del greeks[2]
>>> greeks
['alpha', 'beta', 'delta', 'epsilon']
```

The problem with this approach is that `del` shortens the list. In the example above we can see `greeks` is a list of four strings after the `del` statement is executed. This poses a big problem for the `sift` function because the algorithm depends on removing every  $k^{\text{th}}$  item when it is working its way through the list to remove multiples of  $k$ . If we start removing items from the worksheet successive multiples will no longer be  $k$  numbers apart.

So instead of actually removing a number we should figure out a way to leave it there but mark it as composite. A straightforward way of doing this is to simply overwrite the number with a placeholder. We’ve already decided to use two `None` objects at the front of the list as placeholders, so a natural choice for crossing off a composite number is to simply store a `None` in the corresponding location in the list.

From the original presentation of the algorithm in Section 3.1 and the animations in the previous section it should be clear that to sift out the multiples of some number  $k$  we want to store a `None` object at regular intervals in the worksheet. The first multiple is  $2 \times k$ . In Python terms, we want to store a `None` in `a[2*k]` and then skip ahead  $k$  places to get to the next multiple. This suggests we should use a `range` expression that has an initial value of  $2*k$ , an upper limit that is the length of the list, and a step size of  $k$ :

```
range(2*k, len(a), k)
```

A Python implementation of `sift` that uses this strategy is shown in Figure 3.8. As an example of how it works, consider what the `for` loop does when  $k$  is 5. The locations set to `None` are `a[10]`, `a[15]`, etc., which are all the locations in the worksheet that contain multiples of 5.

This version of the `sift` function is an accurate implementation of the steps illustrated in the previous section, but it actually is doing more work than necessary. An optional programming exercise at the end of this chapter gives some hints about how one might write a more efficient `for` loop that requires fewer iterations.

```

1 def sift(k, a):
2     "Remove multiples of k from list a"
3     for i in range(2*k, len(a), k):
4         a[i] = None

```

**Figure 3.8:** The *sift* helper function iterates through the worksheet in steps of size *k*, replacing numbers with placeholders (*None* objects).

Download: [sieve/sift.py](#)

## Tutorial Project

The goal for this section is to create an initial version of a file named `sieve.py` that will eventually contain the complete definition of `sieve` and its helper functions. This first version will contain only the definition of `sift`; the remaining code will be added in the next section.

To get started, you can either download the function shown in Figure 3.8 and rename the file to `sieve.py` or use your IDE to create new file named `sieve.py` and enter the code shown in the figure.

T58. Load your `sift` function into an interactive shell session in your IDE.

T59. Make a short worksheet to test `sift`:

```
>>> worksheet = [None, None] + list(range(2,30))
```

T60. Call `sift` to remove multiples of 2:

```
>>> sift(2,worksheet)
```

T61. Take a look at the worksheet:

```
>>> worksheet
[None, None, 2, 3, None, 5, None, 7, None, ... 27, None, 29]
```

Do you see how all the even numbers have been replaced by `None`?

T62. Call `sift(3, worksheet)` and verify the multiples of 3 are removed.

T63. What do you suppose would happen if you try to remove multiples of 4 at this point? Check your answer by calling `sift(4, worksheet)`.

T64. Call `sift` one more time to remove multiples of 5, then ask Python to display the worksheet:

```
>>> sift(5,worksheet)
>>> worksheet
[None, None, 2, 3, None, 5, None, 7, None, ... None, None, 29]
```

T65. Are there any more composite numbers in the worksheet?

## 3.7 The sieve Function

With our new helper function implemented and tested we are ready to start working on the sieve function itself. In programming jargon, sieve is the **top-level** function. It is the function that can be called, either by a user in an interactive session or from another program, whenever a list of prime numbers is needed.

As we saw in earlier examples, `sieve` will have a single argument, and it should return a list of prime numbers less than that value, for example

```
>>> sieve(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Because we are going to be passing `sieve` a parameter that specifies the upper limit, the `def` statement at the beginning of the definition will specify a single argument:

```
def sieve(n):
```

The first thing the function needs to do is create the worksheet. All we need to do is modify the expression we used earlier so the upper limit is `n`, the argument passed to `sieve`:

```
    worksheet = [None, None] + list(range(2,n))
```

Note that if we call `sieve(100)` the initial list will go up to 99.

The heart of the implementation will be a loop that calls the `sift` helper to remove composites. We clearly want to use some sort of `for` loop to generate integers to pass to `sift`, and it's also clear that the range of values should start with 2. But what should we use for the upper limit? What is the last number we need to pass to `sift` before we know all the remaining values on the worksheet are prime?

Recall from Section 3.5 that whenever we are looking for primes up to some number  $n$  we can stop sifting as soon as we reach a value greater than  $\sqrt{n}$ . Unfortunately, if we try to specify  $\sqrt{n}$  as the upper limit of a range expression we will get an error message. The square root function in Python's `math` library returns floating point values, but `range` expects us to pass it integers.

Another function in Python's `math` module will help us convert the upper limit to an integer to use in the range expression. The function is called `ceil`, which is short for "ceiling." Calling `ceil(x)` returns the smallest integer greater than  $x$ , that is, it "rounds up" to the next whole number.

Putting all these pieces together, here is a `for` loop that will call `sift` using all the integers from 2 up to the desired limit:

```
    for k in range(2, ceil(sqrt(n))):
        sift(k, worksheet)
```

This loop would work, but it does too much work. It will call `sift` to remove multiples of 2, then 3, and then 4. Exercise T63 showed there is no harm in sifting out multiples of 4: the helper function will just store `None` in the worksheet at locations 8, 12, 16, etc. But those locations are already set to `None` because they are also multiples of 2, so we would be wasting time if we called `sift` with `k` set to 4.

An improvement is to test if the current value has already been crossed off. We can add an `if` statement to the body of the loop so that we call `sift` only if `k` has not already been crossed off. When we are checking to see if something is `None` the idiom in Python is to use a Boolean operator named `is`:

```
    for k in range(2, ceil(sqrt(n))):
        if worksheet[k] is not None:
            sift(k, worksheet)
```

The complete code for our Python implementation of the sieve function is shown in Figure 3.9. It includes the definitions of the two helper functions, the one described earlier that sifts out multiples and a new helper that makes the final list of numbers. The new helper, named `non_nulls`, condenses the worksheet by removing all the `None` objects to create the list that will be returned by the call to `sieve`. The logic used by `non_nulls` is straightforward. It creates an empty list to hold the final result and then passes through the worksheet to collect all the numbers, appending them to the result list.

```
1  from math import sqrt, ceil
2
3  def sieve(n):
4      "Return a list of all prime numbers less than n"
5
6      worksheet = [None, None] + list(range(2,n))
7
8      for k in range(2, ceil(sqrt(n))):
9          if worksheet[k] is not None:
10             sift(k, worksheet)
11
12         return non_nulls(worksheet)
13
14     def sift(k, a):
15         "Remove multiples of k from list a"
16         for i in range(2*k, len(a), k):
17             a[i] = None
18
19     def non_nulls(a):
20         "Return a copy of list a with None objects removed"
21         res = []
22         for x in a:
23             if x is not None:
24                 res.append(x)
25         return res
```

Download: [sieve/sieve.py](#)

**Figure 3.9:** A Python implementation of the Sieve of Eratosthenes.

## Tutorial Project

The tutorial for this section will complete the implementation of the `sieve` function, using the iterative and incremental development style described in the “Art of Programming” sidebar on page 79. If you completed the tutorial exercise from the previous section you have a file named `sieve.py` with the definition of the `sift` helper function. The exercises in this section will add the top-level `sieve` function and the second helper, `non_nulls`.

T66. Start a new shell session with your IDE. Type this `import` statement in your shell window so you can try some experiments with the `ceil` and `sqrt` functions:

```
>>> from math import sqrt, ceil
```

T67. Call `ceil` a few times to make sure you understand what it does:

```
>>> ceil(10.7)
11
>>> ceil(10.2)
11
>>> ceil(10)
10
```

## The Art of Programming (Part I)

At times programming is as much a craft as it is a profession. This sidebar describes some “tricks of the trade” programmers use when developing new programs.

### Iterative and Incremental Development

Many Python programmers have adopted a process from software engineering known as *iterative and incremental development*. The idea is to make very small changes to a program and to continually test the program, performing a set of tests after each change.

This approach to programming also encourages programmers to reuse and adapt code that is already known to work. Python is very well suited to this style of programming. We can type statements in an interactive shell window, for example to see how a function `s` is called, and then copy and paste from the shell into the text of our program.

Here are some suggested milestones if you want to try this style of programming with your `sieve` function. Once each version is working to your satisfaction move on to the next one:

- The first version of `sieve` should do nothing more than create the worksheet and return the entire list.
- Add a `for` loop that prints "filter out" followed by the value of `k`.
- Comment out the print statement and add a call to `sift`, passing it `k` and the worksheet.
- Type in and test the `non_nulls` helper.
- The final version of `sieve` will call `non_nulls` and return the resulting list.

An important benefit of this style of programming is that we find mistakes as soon as we make them. There are dozens of things that can go wrong with even the simplest program: forget a colon at the end of the `def` statement? Leave off the closing parenthesis at the end of the list of parameters? Use the wrong number of spaces in front of statements in the body of a function? By testing a program each time we make a small change we will find these errors as soon as they are made, and when part of a program passes its tests we can move on with confidence to the next phase of the project.

### Print Statements as Scaffolding

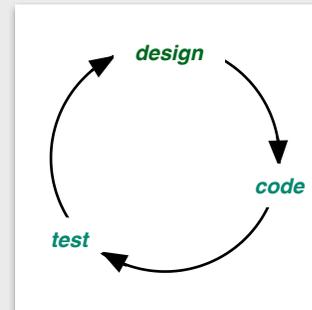
A straightforward technique for debugging a program is to add `print` statements so you can follow the flow of execution. An example is putting a `print` statement at the beginning of a `for` loop so you can see the value of variables at the start of each iteration.

```
for k in range(2, ceil(sqrt(n))):
    # print('sifting', k)
    if worksheet[k]:
        sift(k, worksheet)
```

*After verifying this loop works as expected "comment out" the print statement but leave it in the file.*

When the program is working, rather than delete the print statements, programmers often just insert a comment symbol at the start of the line. If it turns out the program needs further debugging it's easy to remove the comment symbol to reactivate the statement.

These sorts of statements are known as “scaffolding” — they are created and used when the program is under construction, but not intended to be active in the final version.



T68. Compute the upper limit of the range of values to sift for different values of  $n$ :

```
>>> ceil(sqrt(30))
6
>>> ceil(sqrt(100))
10
>>> ceil(sqrt(1000))
32
```

T69. Add the `import` statement to your `sieve.py` file so `ceil` and `sqrt` will be available for your `sieve` function.

T70. Add the definition of the `non_nulls` helper function to your file and load it into your shell window.

T71. Make a small list to test `non_nulls`:

```
>>> colors = ['red', None, 'green', None, 'blue']
```

T72. Test the function:

```
>>> non_nulls(colors)
['red', 'green', 'blue']
```

T73. Make up a few other lists and pass them to `non_nulls`:

```
>>> non_nulls([None, 5, 7, 9, None])
[5, 7, 9]
```

Is it doing what you expect?

You're now ready to add the code for the top-level function. The incremental development philosophy described in the sidebar suggests you do this in two steps. First write and test the code that creates the worksheet, and when that is working, add the `for` loop that calls the `sift` helper.

T74. Type in a version of the `sieve` function that simply creates the worksheet and returns it as the value of the function:

```
def sieve(n):
    worksheet = [None, None] + list(range(2,n))
    return worksheet
```

T75. Save the file and load the outline into your interactive session. Test the function:

```
>>> sieve(30)
[None, None, 2, 3, 4, 5, 6, 7, ... 27, 28, 29]
```

Try calling the function with different values of  $n$ . Do you always get back a list that starts with two `None` objects followed by a list of integers from 2 to  $n - 1$ ?

T76. Edit the return statement so the worksheet is passed to `non_nulls`:

```
return non_nulls(worksheet)
```

T77. Save the file and reload the function. Repeat the tests you made earlier.

```
>>> sieve(30)
[2, 3, 4, 5, 6, 7, ... 27, 28, 29]
```

This time are you getting back lists that have the `None` objects removed?

T78. Add the `for` loop that iterates over the worksheet:

```
for k in range(2, ceil(sqrt(n))):
    if worksheet[k] is not None:
        sift(k, worksheet)
```

At this point your `sieve` function should look like the one in Figure 3.9. You should be able to load it into an interactive session and create a complete list of prime numbers up to any specified upper limit.

## 3.8 ♦ Running a Program from the Command Line

Once we have written a set of functions and tested them with the IDE we are ready to start using them to solve problems. This is where a command line interface will be useful. If we want to see a list of prime numbers, it would be convenient if we could just type a single command in a terminal emulator window instead of starting the IDE, loading the code into an interactive session, and calling the function.

The simplest way to execute a Python program from the command line is to type the program's name as part of the shell command that starts Python. As explained in the Lab Manual, if we run Python without specifying any file names the system starts an interactive session in the terminal emulator window. In this example, the % symbol is the system prompt and the three greater than symbols are Python's interactive prompt:<sup>1</sup>

```
% python3
>>>
```

If we also type the name of a file containing Python source code Python will load that file and execute all the commands in the file. This example shows what we would type to use our `sieve` function to generate a list of prime numbers up to 50 and what will be printed in the terminal window as a result:

```
% python3 sieve.py 50
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

However if we type the command shown above using the version of `sieve.py` in Figure 3.9 it will look like nothing happens. The problem is that the file will be loaded, and the function will be defined, but Python doesn't know what we want to do with it. We need to tell Python to get the upper limit from the command line and pass it in a call to `sieve`.

In order to introduce the technique for creating a command line application we will use the `celsius` function from Chapter 2. The program shown in Figure 3.10 reads an argument from the command line and passes it to the `celsius` function. This command shows how to run the program to convert 77°F to Celsius:

```
% python3 celsius.py 77
25.0
```

The first thing to notice is the program uses a standard library named `sys`. This library includes a variety of items used to connect a program to the operating system. In this case, the statement on line 1 imports `argv`, which is a list containing the collection of arguments the user typed on the command line. The name stands for **argument vector** (“vector” is another term for a list, and the name `argv` has been a standard part of Unix since the 1970s).

The `if` statement on line 7 is the key part of this program. Think of it as a “magic incantation” that effectively says “if this file is being run as a command line program execute the following statements.” We'll dig into the parts of the Boolean expression later in the book when we look at modules and the set of variables Python automatically creates when it loads a module. For now, however, it's sufficient to know that any time you want to turn

---

<sup>1</sup>On Microsoft Windows the command is simply `python`, without the “3”.

```
1  from sys import argv
2
3  def celsius(f):
4      "Convert temperature f to Celsius."
5      return (f - 32) * 5 / 9
6
7  if __name__ == "__main__" :
8      # print(argv)
9      t = int(argv[1])
10     print(celsius(t))
```

Download: [CLI/celsius.py](#)

**Figure 3.10:** This version of `celsius.py` will read a string from the command line, convert it to an integer, pass that number to the `celsius` function, and print the result.

a collection of functions into a command line application you should add an `if` statement that looks exactly like the one on line 7 of this file.

An important detail about `argv` is that it is a list of strings. Even though we type a number on the command line, it is passed to Python as a string of digits. If we uncomment the `print` statement on line 8, so the first thing the program does is print the contents of `argv`, this is what we would see when we run the `celsius` program:

```
% python3 celsius.py 77
['celsius.py', '77']
25.0
```

Notice how the first item in `argv` is the name of the program and the temperature string is in `argv[1]`. The statement on line 9 converts the argument into an integer and saves it in a variable named `t`.

Finally, line 10 has the statement that passes the input temperature to our `celsius` function and prints the result returned by the function. With all these new statements added, this is what we will see when we run the program in a terminal emulator window:

```
% python3 celsius.py 80
26.666666666666668
```

The reason we went to all the trouble of adding the `if` statement on line 7 is that it gives us a choice of running interactively or from the command line. When want to load the file into an interactive session we will type this statement in the shell window:

```
>>> from celsius import *
```

In this case the program is not being run from the command line so the Boolean expression in line 7 is `False`. Python will just load the function but it won't call it yet. To convert temperatures we just call the function interactively, like we did previously:

```
>>> celsius(80)
26.666666666666668
```

## ♦ Tutorial Project

To work the exercises in this section you can use the editor built in to your IDE to add statements to your program but you won't be using the interactive shell to test the changes. Instead, start a separate terminal emulator window where you can type the commands that will launch your program.

- ♦ Add the statement on line 1 of Figure 3.10 to the front of your `celsius.py` file so it imports `argv`.
- ♦ Add the “magic incantation” if statement from line 7 of Figure 3.10 to the end of your file. But for the first version of your command line program put these two statements in the body of the if statement:

```
print(argv[1])
print(type(argv[1]))
```

- ♦ Save the file, start a terminal window, and use the `cd` command to navigate to the directory where you saved `celsius.py`.<sup>2</sup>
- ♦ Using the terminal window, run this first version of your program (the percent sign here is the system prompt, which you don't type):

```
% python3 celsius.py hello
hello
<class 'str'>
```

From this little experiment it should be clear that `argv[1]` is a string.

- ♦ Repeat the experiment but this time type a number for the argument:

```
% python3 celsius.py 100
100
<class 'str'>
```

Python is still interpreting the argument as a string, in this case the three-character string with the digits “1”, “0”, and “0”.

- ♦ If you want to see how the `int` function converts strings to integers, start an interactive shell in your IDE and try some tests:

```
>>> s = "42"
>>> type(s)
<class 'str'>
>>> n = int(s)
>>> type(n)
<class 'int'>
>>> n / 7
6.0
```

- ♦ Replace the two `print` statements in your program with the statements shown on lines 9 and 10 of Figure 3.10.
- ♦ Your program should now convert the string in `argv[1]` to an integer, pass that number to `celsius`, and print the result. Test your program with an argument of 100:

```
% python3 celsius.py 100
37.77777777777778
```

- ♦ Add the `import` statement and “magic incantation” lines to your `sieve.py` file to turn it into a command line application. Can you run the program from the command line to make a list of primes up to 100? Can you still load the file into an interactive session and call `sieve` interactively?

<sup>2</sup>See your Lab Manual if you need information about how to use the `cd` command.

### 3.9 Summary

This chapter explored the Sieve of Eratosthenes, an algorithm that has been used for thousands of years to make lists of prime numbers. The algorithm finds all primes up to some maximum value  $n$  by making a list of all numbers from 2 to  $n$  and then systematically “sifting out” the composite numbers.

We saw how to implement the algorithm in Python in the form of a function named `sieve`. After we load the function into an interactive session, we can call it to create primes up to a specified value:

```
>>> sieve(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

The project introduced an important type of object called a container, which is an object that holds references to other objects. One of the most widely used kinds of containers in Python is a list. We saw how to create lists of numbers and lists of strings, and how to write statements that carry out some important operations on lists:

- We can access an item in a list using an index expression
- Several built-in functions operate on lists, for example we can use `len` to count the number of items in a list
- Methods defined in the list class perform other common operations; for example, if `a` is a reference to a list object, a call to `a.append(x)` will add `x` to the end of the list

Another important idea in computing introduced in this chapter is iteration, which generally means “repetition.” We saw how to use a “for loop” to iterate over a list:

```
for x in a:
    ...
```

This statement tells Python to set the variable `x` to the first element in `a` and then execute the statements in the body of the loop. After the last statement has been executed, Python gets the next item from the list, stores that in `x`, and again executes all the statements in the body. This process repeats until every item in the list has been processed.

The implementation of the `sieve` function was also our first introduction to a very important programming skill: the ability to look at a complex algorithm and break it into smaller pieces that can be worked on independently. If we can implement the smaller pieces as helper functions, we can separately test and debug the operations they perform. When all the pieces are working properly we can assemble them into the final program. This approach to designing solutions to complex problems will be used often throughout the book.

Prime numbers play an important role in modern cryptography. Algorithms that encrypt messages, for example credit card numbers you submit to a secure website when you order something online, use “encryption keys” that are created by multiplying together two large prime numbers. Encrypted messages are safe as long as an intruder cannot break the key into its prime factors. The prime numbers used to make keys are huge. To make it very difficult for an intruder, keys should be around 600 digits long.

The programs at secure websites that choose prime numbers to make keys can’t simply use the Sieve of Eratosthenes to generate their primes because they would have to make an initial worksheet with every number from 1 to  $10^{600}$  (to put this in perspective, there are

---

### Concepts and Terminology Introduced in This Chapter

---

<b>prime number</b>	An integer that is not evenly divisible by any numbers except 1 and itself
<b>composite number</b>	An integer that can be expressed as the product to two or more other integers
<b>list</b>	An ordered collection of objects
<b>index</b>	A number that specifies a location in a list; if a list has $n$ items the indexes range from 0 to $n - 1$
<b>iteration for loop</b>	A technique for solving a problem by repeating a set of steps A programming construct often used to apply an operation to each item in a collection, e.g., to iterate over a list
<b>top-level function</b>	A function that implements a complete solution to a problem
<b>helper function</b>	A function designed to solve a small part of a larger problem
<b>range</b>	A built-in function in Python used to generate a sequence of values in a specified range
<b>ceil</b>	A function from Python's math library that "rounds up" to the nearest integer
<b>None</b>	A special name that means "no object," used as a placeholder in lists or for variables that will be given a value later

---

only  $10^{80}$  atoms in the entire universe). But the sieve is still used by these websites. One of the steps in the algorithm that makes keys relies on a list of small primes, less than 10,000, and Eratosthenes' sieve is still used today to make these shorter lists.

### Exercises

- 3.1. Using a paper and pencil, follow the steps prescribed by the Sieve of Eratosthenes to make a list of prime numbers up to 30. How many rounds of sifting did you have to do?
- 3.2. The last number on the tape in Figure 3.2 is 21. Is the last round of sifting, which removes multiples of 5, necessary if the tape ends at 21? Explain why or why not.
- 3.3. In the example shown in Figure 3.2 the number 12 is marked twice, once on the first round and then again on the second round. Explain the situations where a number is crossed off in more than one round and why it doesn't affect the final result.
- 3.4. What is the value of  $k$  when the for statement on line 5 of Figure 3.9 terminates when  $n = 500$ ? When  $n = 5,000$ ?

3.5. Suppose a list is defined with this assignment statement:

```
>>> names = ['pete', 'john', 'paul', 'george']
```

Explain what Python will print if we ask it to evaluate each of these expressions:

- a) `names[0]`
- b) `names[2]`
- c) `names[4]`
- d) `names[2] == 'george'`

3.6. Using the same list of names, explain what Python would do if we ask it to evaluate each of the following expressions.

- a) `len(names)`
- b) `len(names[0])`
- c) `len(names[0] + names[1])`
- d) `len(names[0]) + len(names[1])`

3.7. Do you think Python will let you change an item in a list using an assignment statement? What do you think will happen if you type this expression (assuming `names` has been defined as above)?

```
>>> names[0] = 'ringo'
```

Use a shell session in your IDE to check your answer.

3.8. Suppose a list is defined with this statement:

```
>>> gas = ['He', 'Ne', 'Ar', 'Kr', 'Xe', 'Rn']
```

What are the values of the following Python expressions?

- a) `'Ne' in gas`
- b) `'Fe' in gas`
- c) `gas.index('Ne')`
- d) `gas.index('Xe')`
- e) `gas.index('Rb')`

3.9. Suppose we define two lists of numbers as follows:

```
>>> a = [1, 1, 2, 3, 5, 8]
```

```
>>> b = [13, 21, 34]
```

Explain what Python will do if we ask it to evaluate each of these expressions:

```
a[0] + b[0]
```

```
a + b
```

3.10. Using the definitions of `a` and `b` from the previous problem, does the following statement make any sense?

```
>>> a += b
```

Explain what Python would do if `a` and `b` were both integers. Does it do something analogous if they are both lists of integers?

3.11. Briefly summarize, in one or two sentences, how Python would execute the following for loop:

```
for i in range(0,10):  
    print(i, i * i)
```

3.12. What would Python print if you entered the following call to `total` (the example function shown on page 62)?

```
>>> total([4, 2, 9, 6, 5, 3, 6])
```

## Programming Projects

- 3.13. Define a function named `prime_before` that returns the largest prime number less than a specific number:

```
>>> prime_before(100)
97
>>> prime_before(1000)
997
```

Hint: Use `sieve` as a helper function.

- 3.14. Write a function named `pi` that prints a table that shows how many primes are less than  $n$  for various values of  $n$ . The argument to your function should be a list of numbers to use for  $n$ , and the output should be a set of lines that shows the value of  $n$  and the number of primes less than  $n$ :

```
>>> pi( [10, 100, 1000, 10000, 100000] )
10 4
100 25
1000 168
10000 1229
100000 9592
```

- 3.15. A straightforward solution to the previous problem is to call `sieve` using each value of  $n$  and print the length of each list. Can you modify your program so it only calls `sieve` once?
- 3.16. A **Mersenne prime** is a prime number that is one less than a power of 2. For example, 131,071 is Mersenne prime because  $131,071 + 1 = 2^{17}$ . Write a function named `mersennes` that returns a list of all Mersenne primes less than a specified value. This is how you would call the function to make a list of Mersenne primes less than 1,000:

```
>>> mersennes(1000)
[3, 7, 31, 127]
```

- 3.17. A **prime pair** is a set of two numbers that are both prime and that differ by 2. For example, 599 and 601 are a prime pair. Write a function named `prime_pairs` that prints all the prime pairs less than a specified value:

```
>>> prime_pairs(50)
3 5
5 7
11 13
17 19
29 31
41 43
```

- 3.18. ♦ The specification for the `sift` function says it should remove multiples of  $k$  from the worksheet by removing  $2 \times k$ ,  $3 \times k$ , and so on. But  $2 \times k$  has already been removed because it is a multiple of 2. Similarly,  $3 \times k$  should have been removed by an earlier call to `sift` out multiples of 3. Can you think of a more effective starting point for removing multiples of  $k$ ? Derive a formula for the location of the first place in the worksheet that contains a multiple of  $k$  and edit your definition of `sift` so it starts at this location.

- 3.19. Write a function named `mean` that will return the mean (arithmetic average) of a list of numbers:

```
>>> mean([4,6,3,8,1])
4.4
```

Hint: Copy the definition of `total`, use the same loop to compute the sum of the numbers, and return the sum divided by the length of the list.

- 3.20. Write a function named `palindrome` that returns `True` if a string reads the same backward and forward:

```
>>> palindrome('madam')
True
>>> palindrome('madman')
False
>>> palindrome('maam')
True
```

Hint: Write a `for` loop with an index variable `i` that goes only half-way through the string. In the body of the loop, compute an index `j` that is `i` characters from the end, and compare the characters and locations `i` and `j`.

- 3.21. Add a function named `gpa` to the file named `gpa.py` you created in the previous chapter (this is the file with the definition `gp`, a function that translates a single letter grade into a number). Your function can use a `for` loop to iterate over a list of grades passed as an argument and return the grade point average:

```
>>> gpa(['A', 'B', 'B'])
3.3333333333333335
```

If you implemented the checks for `+` and `-` grades:

```
>>> gpa(['A', 'A+', 'B+'])
3.8666666666666667
```

- 3.22. ♦ What does your `gpa` function do if one of the strings is not a valid letter grade? Here is the result from a program where `gp` returns 0 if the grade is not valid, and `gpa` computes the average by summing points and dividing by the length of the list:

```
>>> gpa(['A', 'Q', 'B'])
2.3333333333333335
```

Note how the letter `'Q'` was treated as an `'F'`. A better design might be to print a message when a string is not a valid letter grade and to ignore it in the calculation:

```
>>> gpa(['A', 'Q', 'B'])
Unknown grade: Q
3.5
```

Modify your `gpa` function so it prints a warning message and computes the average using only valid letter grades.

- 3.23. A `for` loop can also be used to iterate over a string. For example, this loop prints every character in a string:

```
for ch in s:
    print(ch)
```

Write a function named `vcount` that will count the number of vowels in a string `s`:

```
>>> vcount("banana")
3
>>> vcount("uno, dos, tres")
4
```

Hint: Instead of writing a complicated `if` statement like

```
if ch == 'a' or ch == 'e' or ...
```

define a string named `vowels` before the loop:

```
vowels = 'aeiou'
```

Then, inside the loop, check to see if a character is in the string:

```
if ch in vowels:
```

### Lists of Lists

In this chapter we saw lists of numbers and lists of strings. We can also make a list of lists:

```
>>> a = [ [8,10], [13,12], [6,8] ]
```

If we ask Python how long this list is it should tell us how many lists are contained in a:

```
>>> len(a)
3
```

If we ask for the first item in a we should get back the first list in a:

```
>>> a[0]
[8, 10]
```

3.24. Each item in this list is a pair of numbers that represents the dimensions of rooms in a house:

```
>>> h = [ [18,12], [14,11], [8,10], [8,10] ]
```

Write a function named `area` that computes the total area of all rooms:

```
>>> area(h)
530
```

3.25. Write a function named `avg_bmi` that computes the average body mass index of a group of people, using your `bmi` function from the previous chapter (Project 2.10) as a helper function. The input to `avg_bmi` will be a list of pairs of numbers, where each pair represents the height (in inches) and weight (in pounds) of one person:

```
>>> pop = [ [76, 200], [70, 180], [66, 150] ]
```

If that list is passed to `avg_bmi` this should be the result:

```
>>> avg_bmi(pop)
24.791528013264173
```

3.26. Write a new version of the function `prime_pairs` (see Problem 3.17 above) that returns a list of prime pairs instead of printing the pairs in the terminal window:

```
>>> prime_pairs(50)
[[3, 5], [5, 7], [11, 13], [17, 19], [29, 31], [41, 43]]
```

3.27. ♦ Add statements to your BMI program (Problem 2.10) so it can be used as a command line application:

```
% python3 bmi.py 65 140
23.294674556213018
```

3.28. ♦ Add statements to your loan rate calculator program (Problem 2.11) so it can be used as a command line application:

```
% python3 pmt.py 150000 4.5 30
760.0279647388287
```

Note that the second argument is a floating point number, so you will have to figure out how to convert `argv[2]` to a `float` instead of an `int`.