CHAPTER 4

Growth and Decay

> Our population and our use of the finite resources of planet Earth are growing exponentially, along with our technical ability to change the environment for good or ill.
>
> Stephen Hawking
> *TED talk (2008)*

Like the late Stephen Hawking, many natural and social scientists are concerned with the dynamic sizes of populations and other quantities over time. In addition to our growing use of natural resources, we may be interested in the size of a plant population being affected by an invasive species, the magnitude of an infection threatening a human population, the quantity of a radioactive material in a storage facility, the penetration of a product in the global marketplace, or the evolving characteristics of a dynamic social network. The possibilities are endless.

To study situations like these, scientists develop a simplified ***model*** that abstracts key characteristics of the actual situation so that it might be more easily understood and explored. In this sense, a model is another example of abstraction. Once we have a model that describes the problem, we can write a ***simulation*** that shows what happens when the model is applied over time. A simulation can provide a framework for past observations or predict future behavior. Scientists often use modeling and simulation in parallel with traditional experiments to compare their observations to a proposed theoretical framework.

These parallel scientific processes are illustrated in Figure 4.1. On the left is the computational process. In this case, we use "model" instead of "algorithm" to acknowledge the possibility that the model is mathematical rather than algorithmic. On the right side is the parallel experimental process, guided by the scientific method. The results of the computational and experimental processes can be compared, possibly leading to model adjustments or new experiments to improve the results.

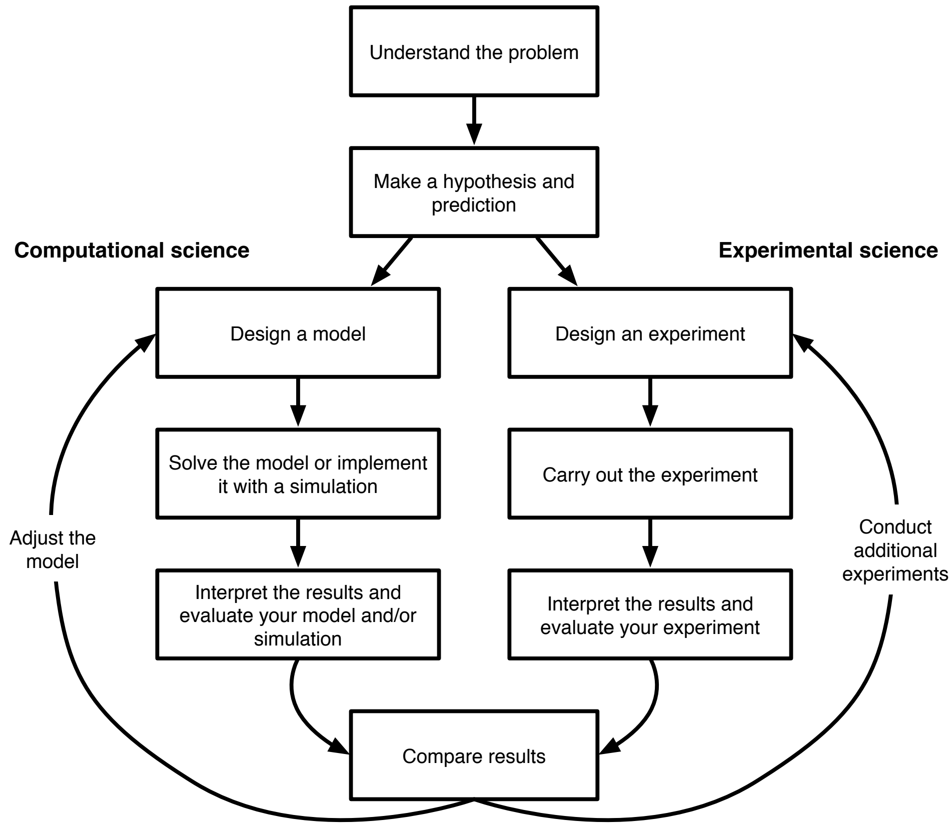


Figure 4.1 Parallel experimental and computational processes.

When we model the dynamic behavior of populations, we will assume that time ticks in discrete steps and, at any particular time step, the current population size is based on the population size at the previous time step. Depending on the problem, a time step may be anywhere from a nanosecond to a century. In general, a new time step may bring population increases, in the form of births and immigration, and population decreases, in the form of deaths and emigration. In this chapter, we will discuss a fundamental algorithmic technique, called an accumulator, that we will use to model dynamic processes like these. Accumulators crop up in all kinds of problems, and lie at the foundation of a variety of different algorithmic techniques. We will continue to see examples throughout the book.

4.1 ACCUMULATORS

Managing a fishing pond

Suppose we manage a fishing pond that contained a population of 12,000 largemouth bass on January 1 of this year. With no fishing, the bass population is expected to grow at a rate of 8% per year, which incorporates both the birth rate and the death rate of the population. The maximum annual fishing harvest allowed is 1,500 bass. Since this is a popular fishing spot, this harvest is attained every year. Is our maximum annual harvest sustainable? If not, how long until the fish population dies out? Should we reduce the maximum harvest? If so, what should it be reduced to?

We can find the projected population size for any given year by starting with the initial population size, and then repeatedly computing the population size in each successive year based on the size in the previous year. In pseudocode, if we remember the current population in a variable named *population*, then we can update the population each year with

$$population \leftarrow population + 0.08 \times population - 1500$$

Or, equivalently,

$$population \leftarrow 1.08 \times population - 1500$$

This is very similar to what we did back on page 14 in our final Sphere Volume algorithm. Remember that an assignment statement evaluates the righthand side first. So the value of *population* on the righthand side of the assignment operator is the value *population* had before this assignment statement was executed. This value is used to compute the new population assigned to the variable on the lefthand side.

If we wanted to know the projected size of the fish population three years from now, we could incorporate this into the following algorithm.

Algorithm Pond Population

```
Input: the initial population
1   population ← initial population
2   repeat three times:
3       population ← 1.08 × population − 1500
Output: the final population
```

Suppose *initial population* is 12000. Then this algorithm performs the following steps:

Trace input: *initial population* = 12000

Step	Line	*population*	Notes
1	1	12000	*population* ← *initial population*
2	2	″	no change; repeat line 3 three times
3	3	11460.0	$population \leftarrow 1.08 \times \underbrace{population}_{12000} - 1500$
4	3	10876.8	$population \leftarrow 1.08 \times \underbrace{population}_{11460.0} - 1500$
5	3	10246.944	$population \leftarrow 1.08 \times \underbrace{population}_{10876.8} - 1500$

Output: *population* = 10246.944

In the first iteration of the loop (step 3 in the trace table), *population* is assigned the previous value of *population* (12,000) times 1.08 minus 1500, which is 11,460. Then, in the second iteration, *population* is updated again after computing the previous value of *population* (now 11,460) times 1.08 minus 1500, which is 10,876.8. In the third iteration, *population* is assigned its final value of 10,246.944. The variable *population* is called an ***accumulator variable*** (or just an *accumulator*) because it accumulates additional value in each iteration of the loop.

So this model projects that the bass population in three years will be 10,246 (ignoring the "fractional fish" represented by the digits to the right of the decimal point).

In Python, we can implement this iterative algorithm with a `for` loop. We used the following `for` loop in Section 2.2 to draw our geometric flower with eight petals:

```
for count in range(8):
    tortoise.forward(200)
    tortoise.left(135)
```

In this case, we need a `for` loop that will iterate three times:

```
population = 12000
for year in range(3):
    population = 1.08 * population - 1500
```

Reflection 4.1 *Type in the* `for` *loop above and add the following statement after the assignment to* `population` *in the body of the* `for` *loop:*

```
print(year + 1, int(population))
```

Run the program. What is printed? Do you see why?

We see in this example that we can use the index variable `year` just like any other variable.

Reflection 4.2 *How would you change this loop to compute the fish population in five years? Ten years?*

Changing the number of years to compute is simple. All we have to do is change the value in the `range` to whatever we want: `range(5)`, `range(10)`, etc. If we put this computation in a function, then we can have the desired number of years passed in as a parameter. The parameter and its use are highlighted in red below.

```
def pond(years):
    """Simulates a fish population in a fishing pond, and
       prints annual population size.  The population
       grows 8% per year with an annual harvest of 1500.

    Parameter:
        years: number of years to simulate

    Return value: the final population size
    """
```

```
 9     population = 12000
10     for year in range(years):
11         population = 1.08 * population - 1500
12         print(year + 1, int(population))

13     return population

14 def main():
15     finalPopulation = pond(10)
16     print('The final population is ' + str(finalPopulation) + '.')

17 main()
```

A trace table to show what happens when we call `pond(10)` is very similar to the one from our pseudocode algorithm, except that we now also want to trace the value of `year`, which is assigned a new value from 0 to 9 in each iteration.

Trace arguments: `years` = 10

Step	Line	`population`	`year`	Notes
1	9	12000	—	`population` ← 12000
2	10	″	0	`year` ← 0
3	11	11460.0	″	`population` ← `1.08 * 12000 - 1500`
4	12	″	″	no changes; prints `1 11460`
5	10	″	1	`year` ← 1
6	11	10876.8	″	`population` ← `1.08 * 11460.0 - 1500`
7	12	″	″	no changes; prints `2 10876`
8	10	″	2	`year` ← 2
9	11	10246.944	″	`population` ← `1.08 * 10876.8 - 1500`
10	12	″	″	no changes; prints `3 10246`
⋮				
29	10	5256.718	9	`year` ← 9
30	11	4177.256	″	`population` ← `1.08 * 5256.718 - 1500`
31	12	″	″	no changes; prints `10 4177`

Return value: 4177.256

Reflection 4.3 *What would happen if* `population = 12000` *was inside the body of the loop instead of before it? What would happen if we omitted the* `population = 12000` *statement altogether?*

The initialization of the accumulator variable before the loop is crucial. If `population` were not initialized before the loop, then an error would occur in the first iteration of the `for` loop because the righthand side of the assignment statement would not make any sense!

Reflection 4.4 *Use the* `pond` *function to answer the original questions: Is this maximum harvest sustainable? If not, how long until the fish population dies out? Should the pond manager reduce the maximum harvest? If so, what should it be reduced to?*

Calling this function with a large enough number of years shows that the fish population drops below zero (which, of course, can't really happen) in year 14:

```
1 11460
2 10876
3 10246
⋮
13 392
14 -1076
⋮
```

This harvesting plan is clearly not sustainable, so the pond manager should reduce it to a sustainable level. In this case, determining the sustainable level is easy: since the population grows at 8% per year and the pond initially contains 12,000 fish, we cannot allow more than $0.08 \cdot 12000 = 960$ fish to be harvested per year without the population declining.

Reflection 4.5 *Generalize the* `pond` *function with two additional parameters: the initial population size and the annual harvest. Using your modified function, compute the number of fish that will be in the pond in 15 years if we change the annual harvest to 800.*

With these modifications, your function might look like this:

```
def pond(years, initialPopulation, harvest):
    """ (docstring omitted) """

    population = initialPopulation
    for year in range(years):
        population = 1.08 * population - harvest
        print(year + 1, int(population))

    return population
```

The value of the `initialPopulation` parameter takes the place of our previous initial population of `12000` and the parameter named `harvest` takes the place of our previous harvest of `1500`. To answer the question above, we can replace the call to the `pond` function from `main` with:

```
finalPopulation = pond(15, 12000, 800)
```

The result that is printed is:

```
1 12160
2 12332
⋮
13 15439
14 15874
15 16344
The final population is 16344.338228396558.
```

Reflection 4.6 *How would you call the new version of the* pond *function to replicate its original behavior, with an annual harvest of 1500?*

Pretty printing

Before moving on, let's look at a helpful Python trick, called a ***format string***, that enables us to format our table of annual populations in a more attractive way. To illustrate the use of a format string, consider the following modified version of the previous function.

```
def pond(years, initialPopulation, harvest):
    """ (docstring omitted) """

    population = initialPopulation
    print('Year | Population')
    print('-----|-----------')
    for year in range(years):
        population = 1.08 * population - harvest
        print('{0:^4} | {1:>9.2f}'.format(year + 1, population))

    return population
```

The first two highlighted lines print a table header to label the columns. Then, in the call to the `print` function inside the `for` loop, we utilize a format string to line up the two values in each row. The syntax of a format string is

```
'<replacement fields>'.format(<values to format>)
```

(The parts in red above are descriptive and not part of the syntax.) The period between the string and `format` indicates that `format` is a method of the string class; we will talk more about the string class in Chapter 6. The parameters of the `format` method are the values to be formatted. The format for each value is specified in a *replacement field* enclosed in curly braces (`{}`) in the format string.

In the example in the `for` loop above, the `{0:^4}` replacement field specifies that the first (really the "zero-th"; computer scientists like to start counting at 0) argument to `format`, in this case `year + 1`, should be centered (`^`) in a field of width 4. The `{1:>9.2f}` replacement field specifies that `population`, as the second argument to `format`, should be right justified (`>`) in a field of width 9 as a floating point number with two places to the right of the decimal point (`.2f`). When formatting floating point numbers (specified by the `f`), the number before the decimal point in the replacement field is the minimum width, including the decimal point. The number after the decimal point in the replacement field is the number of digits to the right of the decimal point in the number. (If we wanted to align to the left, we would use `<`.) Characters in the string that are not in replacement fields (in this case, two spaces with a vertical bar between them) are printed as-is. So, if `year` were assigned the value `11` and `population` were assigned the value `1752.35171`, the above statement would print

$$\underbrace{\texttt{␣12␣␣}}_{\texttt{\{0:\^{}4\}}}\texttt{|}\underbrace{\texttt{␣␣␣1752.35}}_{\texttt{\{1:>9.2f\}}}$$

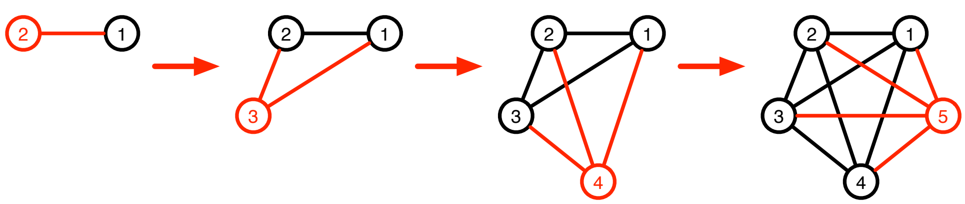
To fill spaces with something other than a space, we can use a *fill character* immediately after the colon. For example, if we replaced the second replacement field with `{1:*>9.2f}`, the previous statement would print the following instead:

$$\underbrace{\texttt{␣12␣␣}}_{\texttt{\{0:\^{}4\}}}\texttt{|}\underbrace{\texttt{␣**1752.35}}_{\texttt{\{1:*>9.2f\}}}$$

Measuring network value

Now let's consider a different problem. Suppose we have created a new online social network (or a new group within an existing social network) that we expect to steadily grow over time. Intuitively, as new members are added, the value of the network to its members grows because new relationships and opportunities become available. The potential value of the network to advertisers also grows as new members are added. But how can we quantify this value?

We will assume that, in our social network, two members can become connected or "linked" by mutual agreement, and that connected members gain access to each other's network profile. The inherent value of the network lies in these connections, or *links*, rather than in the size of its membership. Therefore, we need to figure out how the potential number of links grows as the number of members grows. The picture below visualizes this growth. The circles, called *nodes*, represent members of the social network and lines between nodes represent links between members.



At each step, the red node is added to the network. The red links represent the potential new connections that could result from the addition of the new member.

> **Reflection 4.7** *What is the maximum number of new connections that could arise when each of nodes 2, 3, 4, and 5 are added? In general, what is the maximum number of new connections that could arise from adding node number n?*

Node 2 adds a maximum of 1 new connection, node 3 adds a maximum of 2 new connections, node 4 adds a maximum of 3 new connections, etc. In general, a maximum of $n-1$ new connections arise from the addition of node number n. This pattern is illustrated in the table below.

node number	2	3	4	5	⋯	n
maximum increase in number of links	1	2	3	4	⋯	$n-1$

Therefore, as shown in the last row, the maximum number of links in a network with n nodes is the sum of the numbers in the second row:

$$1 + 2 + 3 + \ldots + n - 1.$$

We will use this sum to represent the potential value of the network.

Let's write a function, similar to the previous one, that lists the maximum number of new links, and the maximum total number of links, as new nodes are added to a network. In this case, we will need an accumulator to count the total number of links. Adapting our `pond` function to this new purpose gives us the following:

```
def countLinks(totalNodes):
    """Prints a table with the maximum total number of links
       in networks with 2 through totalNodes nodes.

    Parameter:
        totalNodes: the total number of nodes in the network

    Return value:
        the maximum number of links in a network with totalNodes nodes
    """

    totalLinks = 0
    for node in range(totalNodes):
        newLinks = ???
        totalLinks = totalLinks + newLinks
        print(node, newLinks, totalLinks)

    return totalLinks
```

In this function, we want our accumulator variable to count the total number of links, so we named it `totalLinks` instead of `population`, and initialized it to zero. Likewise, we named the parameter, which specifies the number of iterations, `totalNodes` instead of `years`, and we named the index variable of the `for` loop `node` instead of `year` because it will now be counting the number of the node that we are adding at each step. In the body of the `for` loop, we add to the accumulator the maximum number of new links added to the network with the current node (we will return to this in a moment) and then print a row containing the node number, the maximum number of new links, and the maximum total number of links in the network at that point. (We leave formatting this row with a format string as an exercise.)

Before we determine what the value of `newLinks` should be, we have to resolve one issue. In the table above, the node numbers range from 2 to the number of nodes in the network, but in our `for` loop, `node` will range from 0 to `totalNodes - 1`. This turns out to be easily fixed because the `range` function can generate a wider variety of number ranges than we have seen thus far. If we give `range` two arguments instead of one, like `range(start, stop)`, the first argument is interpreted as a starting value and the second argument is interpreted as the stopping value, producing a range of values starting at `start` and going up to, *but not including*, `stop`. For example, `range(-5, 10)` produces the integers $-5, -4, -3, \ldots, 8, 9$.

To see this for yourself, type `list(range(-5, 10))` into the Python shell (or `print` it in a program).

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

The `list` function converts a range of numbers into a list that shows all of the numbers in the range.

Reflection 4.8 *What list of numbers does* `range(1, 10)` *produce? What about* `range(10, 1)`*? Can you explain why in each case?*

Reflection 4.9 *Back to our program, what do we want our* `for` *loop to look like?*

For `node` to start at 2 and finish at `totalNodes`, we want our `for` loop to be

```
for node in range(2, totalNodes + 1):
```

Now what should the value of `newLinks` be in our program? The answer is in the table we constructed above; the maximum number of new links added to the network with node number n is $n - 1$. In our loop, the node number is assigned to the name `node`, so we need to add `node - 1` links in each step:

```
newLinks = node - 1
```

With these substitutions, our function looks like this:

```
def countLinks(totalNodes):
    """ (docstring omitted) """

    totalLinks = 0
    for node in range(2, totalNodes + 1):
        newLinks = node - 1
        totalLinks = totalLinks + newLinks
        print(node, newLinks, totalLinks)

    return totalLinks

def main():
    links = countLinks(10)
    print('The total number of links is ' + str(links) + '.')

main()
```

As with our previous `for` loop, you can see more clearly what this loop does by carefully studying the following trace table.

Trace arguments: `totalNodes` = 10					
Step	Line	totalLinks	node	newLinks	Notes
1	3	0	—	—	totalLinks ← 0
2	4	″	2	—	node ← 2
3	5	″	″	1	newLinks ← 2 - 1
4	6	1	″	″	totalLinks ← 0 + 1
5	7	″	″	″	no changes; prints 2 1 1
6	4	″	3	″	node ← 3
7	5	″	″	2	newLinks ← 3 - 1
8	6	3	″	″	totalLinks ← 1 + 2
9	7	″	″	″	no changes; prints 3 2 3
10	4	″	4	″	node ← 4
11	5	″	″	3	newLinks ← 4 - 1
12	6	6	″	″	totalLinks ← 3 + 3
13	7	″	″	″	no changes; prints 4 3 6
⋮					
34	4	36	10	8	node ← 10
35	5	″	″	9	newLinks ← 10 - 1
36	6	45	″	″	totalLinks ← 36 + 9
37	7	″	″	″	no changes; prints 10 9 45
Return value: 45					

When we call `countLinks(10)` from the `main` function above, it prints

```
2 1 1
3 2 3
4 3 6
5 4 10
6 5 15
7 6 21
8 7 28
9 8 36
10 9 45
The total number of links is 45
```

Reflection 4.10 *What does* `countLinks(100)` *return? What does this value represent?*

Organizing a concert

Let's look at one more example. Suppose you are putting on a concert and need to figure out how much to charge per ticket. Your total expenses, for the band and the venue, are \$8,000. The venue can seat at most 2,000 and you have determined through market research that the number of tickets you are likely to sell is related to a ticket's selling price by the following relationship:

```
sales = 2500 - 80 * price
```

According to this relationship, if you give the tickets away for free, you will overfill your venue. On the other hand, if you charge too much, you won't sell any tickets at all. You would like to price the tickets somewhere in between, so as to maximize your profit. Your total income from ticket sales will be `sales * price`, so your profit will be this amount minus $8000.

To determine the most profitable ticket price, we can create a table using a `for` loop similar to that in the previous two problems. In this case, we would like to iterate over a range of ticket prices and print the profit resulting from each choice. In the following function, the `for` loop starts with a ticket price of one dollar and adds one to the price in each iteration until it reaches `maxPrice` dollars.

```
def profitTable(maxPrice):
    """Prints a table of profits from a show based on ticket price.

    Parameters:
        maxPrice: maximum price to consider

    Return value: None
    """

    print('Price    Income     Profit')
    print('------  ---------  ---------')
    for price in range(1, maxPrice + 1):
        sales = 2500 - 80 * price
        income = sales * price
        profit = income - 8000
        formatString = '${0:>5.2f}  ${1:>8.2f}  ${2:8.2f}'
        print(formatString.format(price, income, profit))

def main():
    profitTable(25)

main()
```

The number of expected sales in each iteration is computed from the value of the index variable `price`, according to the relationship above. Then we print the price and the resulting income and profit, formatted nicely with a format string. As we did previously, we can look at what happens in each iteration of the loop with a trace table:

Trace arguments: `maxPrice` = 25						
Step	Line	`price`	`sales`	`income`	`profit`	Notes
1	7	—	—	—	—	prints header
2	8	—	—	—	—	prints underlines
3	9	1	—	—	—	`price` ← 1
4	10	″	2420	—	—	`sales` ← 2500 - 80 * 1
5	11	″	″	2420	—	`income` ← 2420 * 1
6	12	″	″	″	−5580	`profit` ← 2420 - 8000
7	13–14	″	″	″	″	prints price, income, profit
8	9	2	″	″	″	`price` ← 2
9	10	″	2340	″	″	`sales` ← 2500 - 80 * 2
10	11	″	″	4680	″	`income` ← 2340 * 2
11	12	″	″	″	−3320	`profit` ← 4680 - 8000
12	13–14	″	″	″	″	prints price, income, profit
13	9	3	″	″	″	`price` ← 3
⋮						

Reflection 4.11 *Complete a few more iterations in the trace table to make sure you understand how the loop works.*

Reflection 4.12 *Run the program to determine what the most profitable ticket price is.*

The program prints the following table:

```
Price     Income      Profit
------    ---------   ---------
$ 1.00    $ 2420.00   $-5580.00
$ 2.00    $ 4680.00   $-3320.00
$ 3.00    $ 6780.00   $-1220.00
$ 4.00    $ 8720.00   $  720.00
⋮
$15.00    $19500.00   $11500.00
$16.00    $19520.00   $11520.00
$17.00    $19380.00   $11380.00
⋮
$24.00    $13920.00   $ 5920.00
$25.00    $12500.00   $ 4500.00
```

The profit in the third column increases until it reaches $11,520.00 at a ticket price of $16, then it drops off. So the most profitable ticket price seems to be $16.

Reflection 4.13 *Our program only considered whole dollar ticket prices. How can we modify it to increment the ticket price by fifty cents in each iteration instead?*

The `range` function can only create ranges of integers, so we cannot ask the `range` function to increment by 0.5 instead of 1. But we can achieve our goal by doubling the range of numbers that we iterate over, and then set the price in each iteration to be the value of the index variable divided by two.

```
def profitTable(maxPrice):
    """ (docstring omitted) """

    print('Price    Income     Profit')
    print('------  ---------  ---------')
    for price in range(1, 2 * maxPrice + 1):
        realPrice = price / 2
        sales = 2500 - 80 * realPrice
        income = sales * realPrice
        profit = income - 8000
        formatString = '${0:>5.2f}  ${1:>8.2f}  ${2:8.2f}'
        print(formatString.format(realPrice, income, profit))
```

Now when `price` is 1, the "real price" that is used to compute profit is 0.5. When `price` is 2, the "real price" is 1.0, etc.

Reflection 4.14 *Does our new function find a more profitable ticket price than $16?*

Our new function prints the following table.

```
Price    Income     Profit
------  ---------  ---------
$ 0.50  $ 1230.00  $-6770.00
$ 1.00  $ 2420.00  $-5580.00
$ 1.50  $ 3570.00  $-4430.00
$ 2.00  $ 4680.00  $-3320.00
⋮
$15.50  $19530.00  $11530.00
$16.00  $19520.00  $11520.00
$16.50  $19470.00  $11470.00
⋮
$24.50  $13230.00  $ 5230.00
$25.00  $12500.00  $ 4500.00
```

If we look at the ticket prices around $16, we see that $15.50 will actually make $10 more.

Just from looking at the table, the relationship between the ticket price and the profit is not as clear as it would be if we plotted the data instead. For example, does profit rise in a straight line to the maximum and then fall in a straight line? Or is it a more gradual curve? We can answer these questions by drawing a plot with turtle graphics, using the `goto` method to move the turtle from one point to the next.

```
import turtle

def profitPlot(tortoise, maxPrice):
    """ (docstring omitted) """
```

```
    for price in range(1, 2 * maxPrice + 1):
        realPrice = price / 2
        sales = 2500 - 80 * realPrice
        income = sales * realPrice
        profit = income - 8000
        tortoise.goto(realPrice, profit)

def main():
    george = turtle.Turtle()
    screen = george.getscreen()
    screen.setworldcoordinates(0, -15000, 25, 15000)
    profitPlot(george, 25)

main()
```

Our new `main` function sets up a turtle and then uses the `setworldcoordinates` method to change the coordinate system in the drawing window to fit the points that we are likely to plot. In the `for` loop in the `profitPlot` function, since the first value of `realPrice` is `0.5`, the first `goto` is

```
george.goto(0.5, -6770)
```

which draws a line from the origin (0,0) to $(0.5, -6770)$. In the next iteration, the value of `realPrice` is `1.0`, so the loop next executes

```
george.goto(1.0, -5580)
```

which draws a line from the previous position of $(0.5, -6770)$ to $(1.0, -5580)$. The next value of `realPrice` is `1.5`, so the loop then executes

```
george.goto(1.5, -4430)
```

which draws a line from from $(1.0, -5580)$ to $(1.5, -4430)$. And so on, until `realPrice` takes on its final value of 25 and we draw a line from the previous position of $(24.5, 5230)$ to $(25, 4500)$.

Reflection 4.15 *What shape is the plot? Can you see why?*

Reflection 4.16 *When you run this plotting program, you will notice an ugly line from the origin to the first point of the plot. How can you fix this? (We will leave the answer as an exercise.)*

Exercises

Write a function for each of the following problems. Be sure to appropriately document your functions with docstrings and comments. Test each function with both common and boundary case arguments, as described on page 38, and document your test cases. Use a trace table on at least one test case.

4.1.1* Generalize the `pond` function so that it also takes the annual growth rate as a parameter.

4.1.2. Generalize the `pond` function further to allow for the pond to be annually restocked with an additional quantity of fish.

4.1.3. Modify the `countLinks` function so that it prints a table like the following:

```
|       |   Links      |
| Nodes | New | Total |
| ----- | --- | ----- |
|   2   | 1   |     1 |
|   3   | 2   |     3 |
|   4   | 3   |     6 |
|   5   | 4   |    10 |
|   6   | 5   |    15 |
|   7   | 6   |    21 |
|   8   | 7   |    28 |
|   9   | 8   |    36 |
|  10   | 9   |    45 |
```

4.1.4. Modify the `profitTable` function so that it considers all ticket prices that are multiples of a quarter.

4.1.5. In the `profitPlot` function in the text, fix the problem raised by Reflection 4.16.

4.1.6. There are actually three forms of the `range` function:

- 1 parameter: `range(stop)`
- 2 parameters: `range(start, stop)`
- 3 parameters: `range(start, stop, skip)`

With three arguments, `range` produces a range of integers starting at the start value and ending at or before `stop - 1`, adding `skip` each time. For example,

```
range(5, 15, 2)
```

produces the range of numbers `5, 7, 9, 11, 13` and

```
range(-5, -15, -2)
```

produces the range of numbers `-5, -7, -9, -11, -13`. To print these numbers, one per line, we can use a `for` loop:

```
for number in range(-5, -15, -2):
    print(number)
```

(a) Write a `for` loop that prints the integers from 0 to 100.

(b) Write a `for` loop that prints the integers from -50 to 50.

(c) Write a `for` loop that prints the even integers from 2 to 100, using the third form of the `range` function.

(d) Write a `for` loop that prints the odd integers from 1 to 100, using the third form of the `range` function.

(e) Write a `for` loop that prints the integers from 100 to 1 in descending order.

(f) Write a `for` loop that prints the values 7, 11, 15, 19.

(g) Write a `for` loop that prints the values 2, 1, 0, −1, −2.

(h) Write a `for` loop that prints the values −7, −11, −15, −19.

4.1.7* Write a function

```
triangle()
```

that uses a `for` loop to print the following:

```
*
**
***
****
*****
******
*******
********
*********
**********
```

4.1.8. Write a function

```
diamond()
```

that uses `for` loops to print the following:

```
***** *****
****   ****
***     ***
**       **
*         *
*         *
**       **
***     ***
****   ****
***** *****
```

4.1.9. Write a function

```
square(letter, width)
```

that prints a square with the given `width` using the string `letter`. For example, `square('Q', 5)` should print:

```
QQQQQ
QQQQQ
QQQQQ
QQQQQ
QQQQQ
```

4.1.10* Write a `for` loop that uses `range(50)` to print the odd integers from 1 to 100.

4.1.11* Write a function

```
multiples(n)
```

that prints all of the multiples of the parameter `n` between 0 and 100, inclusive. For example, if `n` were `4`, the function should print the values 0, 4, 8, 12, ...

4.1.12. Write a function

```
countdown(n)
```

that prints the integers between 0 and `n` in descending order. For example, if `n` were `5`, the function should print the values 5, 4, 3, 2, 1, 0.

4.1.13. On page 122, we talked about how to simulate the minutes ticking on a digital clock using modular arithmetic. Write a function

```
clock(ticks)
```

that prints `ticks` times starting from midnight, where the clock ticks once each minute. To simplify matters, the midnight hour can be denoted 0 instead of 12. For example, `clock(100)` should print

```
 0:00
 0:01
 0:02
 ⋮
 0:59
 1:00
 1:01
 ⋮
 1:38
 1:39
```

To line up the colons in the times and force the leading zero in the minutes, use a format string like this:

```
print('{0:>2}:{1:0>2}'.format(hours, minutes))
```

4.1.14. Write a function

```
circles(tortoise)
```

that uses turtle graphics and a `for` loop to draw concentric circles with radii $10, 20, 30, \ldots, 100$. (To draw each circle, you may use the turtle graphics `circle` method or the `drawCircle` function you wrote in Exercise 2.3.14.)

4.1.15* Write a function

```
plotSine(tortoise, n)
```

that uses turtle graphics to plot $\sin x$ from $x = 0$ to $x = n$ degrees. Use `setworldcoordinates` to make the x coordinates of the window range from 0 to n and the y coordinates range from -1 to 1.

4.1.16. Python also allows us to pass function names as parameters. So we can generalize the function in Exercise 4.1.15 to plot any function we want. Write a function

```
plot(tortoise, n, f)
```

where `f` is the name of an arbitrary function that takes a single numerical argument and returns a number. Inside the `for` loop in the `plot` function, we can apply the function `f` to the index variable `x` with

```
tortoise.goto(x, f(x))
```

To call the `plot` function, we need to define one or more functions to pass in as arguments. For example, to plot x^2, we can define

```
def square(x):
    return x * x
```

and then call `plot` with

```
plot(george, 20, square)
```

Or, to plot an elongated $\sin x$, we could define

```
def sin(x):
    return 10 * math.sin(x)
```

and then call `plot` with

```
plot(george, 20, sin)
```

After you create your new version of `plot`, also create at least one new function to pass into `plot` for the parameter `f`. Depending on the functions you pass in, you may need to adjust the window coordinate system with `setworldcoordinates`.

4.1.17* Write a function

```
growth1(totalDays)
```

that simulates a population growing by 3 individuals each day. For each day, print the day number and the total population size.

4.1.18. Write a function

```
growth2(totalDays)
```

that simulates a population that grows by 3 individuals each day but also shrinks by, on average, 1 individual every 2 days. For each day, print the day number and the total population size.

4.1.19. Write a function

```
growth3(totalDays)
```

that simulates a population that increases by 110% every day. Assume that the initial population size is 10. For each day, print the day number and the total population size.

4.1.20. Write a function

```
growth4(totalDays)
```

that simulates a population that grows by 2 on the first day, 4 on the second day, 8 on the third day, 16 on the fourth day, etc. Assume that the initial population size is 10. For each day, print the day number and the total population size.

4.1.21* Suppose a bacteria colony grows at a rate of 10% per hour and that there are initially 100 bacteria in the colony. Write a function

```
bacteria(days)
```

that returns the number of bacteria in the colony after the given number of `days`. How many bacteria are in the colony after one week?

4.1.22. Generalize the function that you wrote for the previous exercise so that it also accepts parameters for the initial population size and the growth rate. How many bacteria are in the same colony after one week if it grows at 15% per hour instead?

4.1.23* Write a function

```
sumNumbers(n)
```

that returns the sum of the integers between 1 and `n`, inclusive. For example, `sum(4)` returns $1 + 2 + 3 + 4 = 10$. (Use a `for` loop; if you know a shortcut, don't use it.)

4.1.24. Write a function

```
sumEven(n)
```

that returns the sum of the even integers between 2 and `n`, inclusive. For example, `sumEven(6)` returns $2 + 4 + 6 = 12$. (Use a `for` loop.)

4.1.25. Write a function

```
average(low, high)
```

that returns the average of the integers between `low` and `high`, inclusive. For example, `average(3, 6)` returns $(3 + 4 + 5 + 6)/4 = 4.5$.

4.1.26* Write a function

```
factorial(n)
```

that returns the value of $n! = 1 \times 2 \times 3 \times \cdots \times (n - 1) \times n$. (Be careful; how should the accumulator be initialized?)

4.1.27. Write a function

```
power(base, exponent)
```

that returns the value of `base` raised to the `exponent` power, without using the `**` operator. Assume that `exponent` is a positive integer.

4.1.28. The geometric mean of n numbers is defined to be the nth root of the product of the numbers. (The nth root is the same as the $1/n$ power.) Write a function

```
geoMean(high)
```

that returns the geometric mean of the numbers between 1 and `high`, inclusive.

4.1.29. Write a function

```
sumDigits(number, numDigits)
```

that returns the sum of the individual digits in a parameter `number` that has `numDigits` digits. For example, `sumDigits(1234, 4)` should return the value $1 + 2 + 3 + 4 = 10$. (Hint: use a `for` loop and integer division (`//` and `%`).)

4.1.30. Between the ages of three and thirteen, girls grow an average of about six centimeters per year. Write a function

```
growth(finalAge)
```

that prints a simple height chart based on this information, with one entry for each age, assuming the average girl is 95 centimeters (37 inches) tall at age three.

4.1.31. Consider the following fun game. Pick any positive integer less than 100 and add the squares of its digits. For example, if you choose 25, the sum of the squares of its digits is $2^2 + 5^2 = 29$. Now make the answer your new number, and repeat the process. For example, if we continue this process starting with 25, we get: 25, 29, 85, 89, 145, 42, etc.

Write a function

```
fun(number, iterations)
```

that prints the sequence of numbers generated by this game, starting with the two digit `number`, and continuing for the given number of `iterations`. It will be helpful to know that no number will ever have more than three digits.

Execute your function with every integer between 15 and 25, with `iterations` at least 30. What do you notice? Can you classify each of these integers into one of two groups based on the results?

4.1.32. Create trace tables that show the execution of each of the following functions.

(a)* your `growth1` function from Exercise 4.1.17 when it is called as `growth1(4)`

(b) your `growth3` function from Exercise 4.1.19 when it is called as `growth3(4)`

(c) your `bacteria` function from Exercise 4.1.21 when it is called as `bacteria(5)`

4.1.33* You have $1,000 to invest and need to decide between two savings accounts. The first account pays interest at an annual rate of 1% and is compounded daily, meaning that interest is earned daily at a rate of (1/365)%. The second account pays interest at an annual rate of 1.25% but is compounded monthly. Write a function

```
interest(originalAmount, rate, periods)
```

that computes the interest earned in one year on `originalAmount` dollars in an account that pays the given annual interest rate, compounded over the given number of periods. Assume the interest rate is given as a percentage, not a fraction (i.e., 1.25 vs. 0.0125). Use the function to answer the original question.

4.1.34. Suppose you want to start saving a certain amount each month in an investment account that compounds interest monthly. To determine how much money you expect to have in the future, write a function

```
invest(investment, rate, years)
```

that returns the income earned by investing `investment` dollars every month in an investment account that pays the given rate of return, compounded monthly (`rate` / 12 % each month).

4.1.35. A mortgage loan is charged some rate of interest every month based on the current balance on the loan. If the annual interest rate of the mortgage is r%, then interest equal to $r/12$ % of the current balance is added to the amount owed each month. Also each month, the borrower is expected to make a payment, which reduces the amount owed.

Write a function

```
mortgage(principal, rate, years, payment)
```

that prints a table of mortgage payments and the remaining balance every month of the loan period. The last payment should include any remaining balance. For example, paying $1,000 per month on a $200,000 loan at 4.5% for 30 years should result in the following table:

```
Month   Payment     Balance
1       1000.00   199750.00
2       1000.00   199499.06
3       1000.00   199247.18
⋮
359     1000.00    11111.79
360    11153.46        0.00
```

4.2 DATA VISUALIZATION

Visualizing changes in population size over time will provide more insight into how population models evolve. We could plot population changes with turtle graphics, as we did in Section 4.1, but instead, we will use a dedicated plotting module called `matplotlib`, so-named because it emulates the plotting capabilities of the technical programming language MATLAB[1].

To use `matplotlib`, we first import the module using

```
import matplotlib.pyplot as pyplot
```

`matplotlib.pyplot` is the name of module; "`as pyplot`" allows us to refer to the module in our program with the abbreviation `pyplot` instead of its rather long full name. The basic plotting functions take two arguments: a list of x values and an associated list of y values. As we saw before, a list in Python is represented as a comma-separated sequence of items enclosed in square brackets, such as

```
[4, 7, 2, 3.1, 12, 2.1]
```

We will use lists much more extensively in Chapter 7. For now, we only need to know how to build a list of population sizes in our `for` loop so that we can plot them. Let's look at how to do this in the fishing pond function from page 135, reproduced below.

```
def pond(years, initialPopulation, harvest):
    """ (docstring omitted) """

    population = initialPopulation
    print('Year | Population')
    print('-----|-----------')
    for year in range(years):
        population = 1.08 * population - harvest
        print('{0:^4} | {1:>9.2f}'.format(year + 1, population))

    return population
```

To build this list, we start by creating an empty list before the loop:

```
populationList = [ ]
```

To add an annual population size to the end of the list, we will use the `append` method of the list class. We will first append the initial population size to the end of the empty list with

```
populationList.append(initialPopulation)
```

If we pass in `12000` for the initial population parameter, this will result in `populationList` becoming the single-element list `[12000]`. Inside the loop, we want to append each value of `population` to the end of the growing list with

```
populationList.append(population)
```

[1]MATLAB is a registered trademark of The MathWorks, Inc.

Incorporating this code into our `pond` function, and deleting the calls to `print`, yields:

```
1 def pond(years, initialPopulation, harvest):
2     """Simulates a fish population and plots annual population size.
3        The population grows 8% per year with an annual harvest.

4     Parameters:
5         years:             number of years to simulate
6         initialPopulation: the initial population size
7         harvest:           the size of the annual harvest

8     Return value: the final population size
9     """

10    population = initialPopulation
11    populationList = [ ]
12    populationList.append(initialPopulation)
13    for year in range(1, years + 1):
14        population = 1.08 * population - harvest
15        populationList.append(population)
16    return population
```

We have also changed the `for` loop range to start at 1 to reflect that the first population size computed inside the loop reflects the size at year 1 (and the population before the loop represents "year 0"). The trace table below shows how `populationList` grows with each iteration, assuming an initial population of 12,000.

Trace arguments: `years = 14`, `initialPopulation = 12000`, `harvest = 1500`

Step	Line	year	population	populationList	Notes
1	11	—	12000	—	init `population`
2	12	—	″	`[ ]`	init `populationList`
3	13	—	″	`[12000]`	append `12000`
4	14	1	″	″	`year` ← 1
5	15	″	11460.0	″	update `population`
6	16	″	″	`[12000, 11460.0]`	append `11460.0`
7	14	2	″	″	`year` ← 2
8	15	″	10876.8	″	update `population`
9	16	″	″	`[12000, ..., 10876.8]`	append `10876.8`
⋮					
43	14	14	392.539	`[12000, ..., 392.539]`	`year` ← 14
44	15	″	-1076.056	″	update `population`
45	16	″	″	`[12000, ..., -1076.056]`	append `-1076.056`

Return value: `-1076.056`

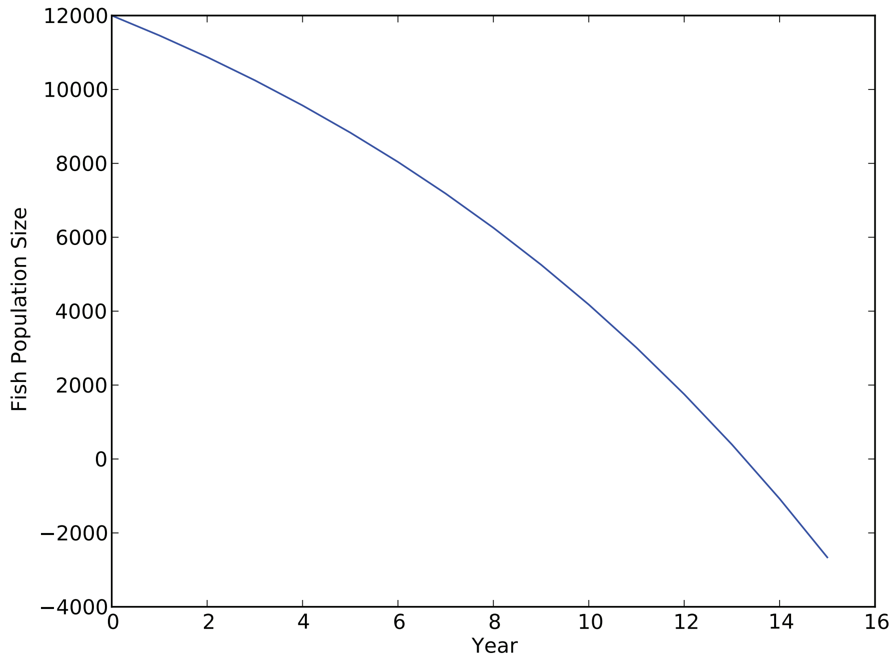


Figure 4.2 Plot of population size in our fishing pond model with `years = 15`.

In each iteration, the current value of `population` is appended to the end of `populationList`. So when the loop is finished, there are `years + 1` population sizes in the list.

Reflection 4.17 *Add a statement to print* `populationList` *at the end of each iteration of the loop so that you can see better how it grows.*

There is a strong similarity between the manner in which we are appending elements to a list and the accumulators that we have been talking about in this chapter. In an accumulator, we accumulate values into a sum by repeatedly adding new values to a running sum. The running sum changes (usually grows) in each iteration of the loop. With the list in the `for` loop above, we are accumulating values in a different way—by repeatedly appending them to the end of a growing list. Therefore, we call this technique a ***list accumulator***.

We now want to use this list of population sizes as the list of y values in a `matplotlib` plot. For the x values, we need a list of the corresponding years, which can be obtained with `range(years + 1)`.

Reflection 4.18 *Why do we need the x values to be* `range(years + 1)` *instead of* `range(1, years + 1)`*? Think about how many population values are in* `populationList`.

Once we have both lists, we can create a plot by calling the `plot` function and then display the plot by calling the `show` function:

```
pyplot.plot(range(years + 1), populationList)
pyplot.show()
```

The first argument to the `plot` function is the list of x values and the second parameter is the list of y values. The `matplotlib.pyplot` module includes many optional ways to customize our plots before we call `show`. Some of the simplest are functions that label the x and y axes:

```
pyplot.xlabel('Year')
pyplot.ylabel('Fish Population Size')
```

Incorporating the plotting code yields the following function, whose output is shown in Figure 4.2.

```
import matplotlib.pyplot as pyplot

def pond(years, initialPopulation, harvest):
    """ (docstring omitted) """

    population = initialPopulation
    populationList = [ ]
    populationList.append(initialPopulation)
    for year in range(1, years + 1):
        population = 1.08 * population - harvest
        populationList.append(population)

    pyplot.plot(range(years + 1), populationList)
    pyplot.xlabel('Year')
    pyplot.ylabel('Fish Population Size')
    pyplot.show()

    return population
```

For more complex plots, we can alter the scales of the axes, change the color and style of the curves, and label multiple curves on the same plot. See Appendix A.4 for a sample of what is available. Some of the options must be specified as *keyword arguments* of the form `name = value`. For example, to color a curve in a plot red and specify a label for the plot legend, you would call something like this:

```
pyplot.plot(x, y, color = 'red', label = 'Bass population')
pyplot.legend()    # creates a legend from labeled lines
```

Exercises

4.2.1* A zombie can convert two people into zombies everyday. Assuming we start with just one zombie, write a function

```
zombieApocalypse(days)
```

that plots the total number of zombies (y axis) roaming the earth over each of the given number of `days` (x axis). Appropriately label your axes. Use your function to create a plot of zombie growth over 14 days.

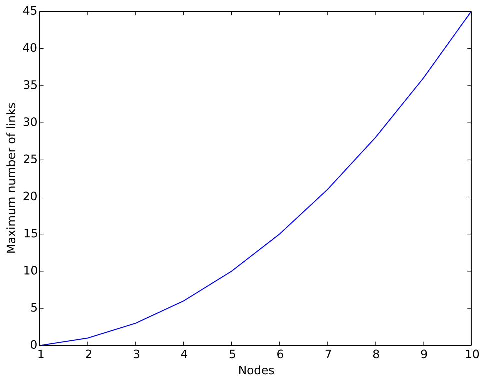


Figure 4.3 Plot for Exercise 4.2.2.

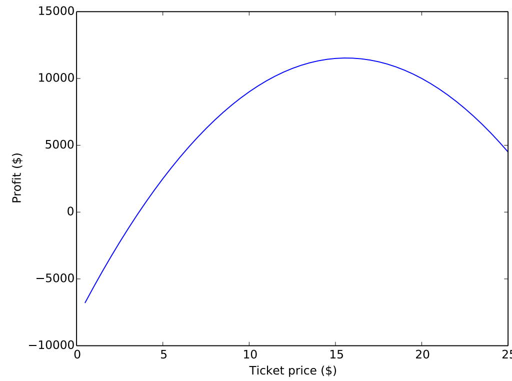


Figure 4.4 Plot for Exercise 4.2.3.

4.2.2. Modify the `countLinks` function on page 138 so that it uses `matplotlib` to plot the number of nodes on the x axis and the maximum number of links on the y axis. Create a plot that shows the maximum number of links for 1 to 10 nodes; it should look like the one in Figure 4.3.

4.2.3* Modify the `profitPlot` function on page 142 so that it uses `matplotlib` to plot ticket price on the x axis and profit on the y axis. (Remove the `tortoise` parameter.) Create a plot that shows the profit for ticket prices up to $25; it should look like the one in Figure 4.4. To get the correct prices (in half dollar increments) on the x axis, you will need to create a second list of x values and append `realPrice` to it in each iteration.

4.2.4. Modify your `growth1` function from Exercise 4.1.17 so that it uses `matplotlib` to plot days on the x axis and the total population on the y axis. Create a plot that shows the growth of the population over 30 days.

4.2.5. Modify your `growth3` function from Exercise 4.1.19 so that it uses `matplotlib` to plot days on the x axis and the total population on the y axis. Create a plot that shows the growth of the population over 30 days.

4.2.6. Modify your `invest` function from Exercise 4.1.34 so that it uses `matplotlib` to plot months on the x axis and your total accumulated investment amount on the y axis. Create a plot that shows the growth of an investment of $50 per month for ten years growing at an annual rate of 8%.

4.2.7* Write a function that compares the growth rates of two bacteria colonies (like in Exercise 4.1.21), one that grows 10% per hour and another that grows 15% per hour. Your function should have one `for` loop that accumulates two population variables and two lists independently. After the loop, use two `pyplot.plot` calls before `pyplot.show()`, each with its own label (as shown above), to plot the populations. Include a legend that shows which curve is which. Create a plot with your function that compares growth over a period of 3 days.

4.2.8. Vampires can each convert v people a day into vampires. However, there is a band of vampire hunters that can kill k vampires per day. Write a function

```
vampireApocalypse(v, k, vampires, people, days)
```

that plots the numbers of vampires and people in a town with initial population `people` over the given number of `days`, assuming the town starts with a coven with `vampires` members. Use your function to create a plot of vampires and people over a period of 7 days. See the previous exercise for how to plot multiple lists.

4.2.9. Write a function that compares the growth in population sizes in Exercises 4.1.17, 4.1.19, and 4.1.20 over a number of days. Create a plot with your function that compares growth over 14 days. Use three calls to `pyplot.plot` before `pyplot.show()` and include a legend. Contrast the three growth rates. What do you notice?

4.3 CONDITIONAL ITERATION

In our fishing pond model, to determine when the population size fell below zero, it was sufficient to simply print the annual population sizes for at least 14 years, and look at the results. However, if it had taken a thousand years for the population size to fall below zero, then looking at the output would be far less convenient. Instead, we would like to have a program tell us the year directly, by ceasing to iterate when `population` drops below zero, and then returning the year it happened. This is a different kind of problem because we no longer know how many iterations are required before the loop starts. In other words, we have no way of knowing what value to pass into `range` in a `for` loop.

Instead, we need a more general kind of loop that will iterate only while some condition is met. Such a loop is generally called a ***while loop***. In Python, a while loop looks like this:

```
while <condition>:
    <body>
```

The `<condition>` is replaced with a Boolean expression that evaluates to `True` or `False`, and the `<body>` is replaced with statements constituting the body of the loop. The loop checks the value of the condition before each iteration. If the condition is true, it executes the body of the loop, and then checks the condition again. If the condition is false, the body of the loop is skipped, and the loop ends.

When will the fish disappear?

To solve this problem, we want to continue to update `population` in a loop while `popluation > 0`. This Boolean expression is true if the value of `population` is positive, and false otherwise. Using this Boolean expression in the `while` loop in the following function, we can find the year that the fish population drops to 0.

```
1  def yearsUntilZero(initialPopulation, harvest):
2      """Computes the number of years until a fish population reaches zero.
3         Population grows 8% per year with an annual harvest.

4      Parameters:
5          initialPopulation: the initial population size
6          harvest:           the size of the annual harvest

7      Return value: year during which the population reaches zero
8      """

9      population = initialPopulation
10     year = 0
11     while population > 0:
12         population = 1.08 * population - harvest
13         year = year + 1
14     return year
```

The following trace table shows how the loop works when `initialPopulation` is `12000` and `harvest` is `1500`, as in our original `pond` function in Section 4.1.

Trace arguments: `initialPopulation = 12000`, `harvest = 1500`				
Step	Line	`population`	`year`	Notes
1	9	`12000`	—	`population` ← `12000`
2	10	″	`0`	`year` ← `0`
3	11	″	″	`population > 0`, so execute the body of the loop
4	12	`11460.0`	″	update `population`
5	13	″	`1`	increment `year`
6	11	″	″	`population > 0`, so execute the body of the loop
7	12	`10876.8`	″	update `population`
8	13	″	`2`	increment `year`
9	11	″	″	`population > 0`, so execute the body of the loop
⋮				
42	11	`392.539`	`13`	`population > 0`, so execute the body of the loop
43	12	`-1076.056`	″	update `population`
44	13	″	`14`	increment `year`
45	11	″	″	`population <= 0`, so exit the loop
46	14	″	″	`return 14`
Return value: `14`				

Before the loop, `population` is `12000` and `year` is `0`. Since `population > 0` is true, the loop body executes in steps 4–5, causing `population` to become `11460` and `year` to become `1`. We then go back to the top of the loop in step 6 to check the condition again. Since `population > 0` is still true, the loop body executes again in steps 7–8, causing `population` to become `10876.8` and `year` to become `2`. Iteration continues until `year` reaches `14`. In this year, `population` becomes `-1076.06`. When the condition is checked at the beginning of the next iteration, we find that `population > 0` is false, so the loop ends and the function returns `14`.

Using `while` loops can be tricky for a few reasons. First, a `while` loop may not iterate at all. For example, if the initial value of `population` were zero, the condition in the `while` loop will be false before the first iteration, and the loop will be over before it starts.

Reflection 4.19 *What will be returned by the function if the initial value of* `population` *were zero?*

A loop that sometimes does not iterate at all is generally not a bad thing, and can even be used to our advantage. In this case, if `population` were initially zero, the function would return zero because the value of `year` would never be incremented in the loop. And this is correct; the population dropped to zero in year zero, before the clock started ticking beyond the initial population size. But it is something that one should always keep in mind when designing algorithms involving `while` loops.

Second, and related to the first point, you need to always make sure that the condition in the `while` loop makes sense before the first iteration. For example, suppose we forgot to give `population` an initial value before the loop. Then the loop condition would not make any sense because `population` was not defined.

Third, a `while` loop may become an ***infinite loop***. For example, suppose `initialPopulation` is `12000` and `harvest` is `800` instead of `1500`. In this case, as we saw on page 134, the population size *increases* every year instead. So the population size will *never* reach zero and the loop condition will *never* be false, so the loop will iterate forever. For this reason, we must always make sure that the body of a `while` loop makes progress toward the loop condition becoming false.

These points can be summarized in two rules to always keep in mind when designing an algorithm with a `while` loop:

1. Initialize the condition before the loop. Always make sure that the condition makes sense and will behave in the intended way the first time it is tested.
2. In each iteration of the loop, work toward the condition eventually becoming false. Not doing so will result in an infinite loop.

When will your nest egg double?

Let's look at one more example. Suppose we have $1000 to invest and we would like to know how long it will take for our money to double in size, growing at 5% per year. To answer this question, let's start with the following incomplete loop that compounds 5% interest each year:

```
amount = 1000
while ???:
    amount = 1.05 * amount
print(amount)
```

Reflection 4.20 *What should be the condition in the* `while` *loop?*

We want the loop to stop iterating when `amount` reaches `2000`. Therefore, we want the loop to continue to iterate *while* `amount < 2000`.

```
amount = 1000
while amount < 2000:
    amount = 1.05 * amount
print(amount)
```

Reflection 4.21 *What is printed by this block of code? What does this result tell us?*

Once the loop is done iterating, the final amount is printed (approximately $2078.93), but this does not answer our question.

Reflection 4.22 *How do we figure out how many years it takes for the $1000 to double?*

To answer our question, we need to count the number of times the `while` loop iterates, which is very similar to what we did in the `yearsUntilZero` function. We can introduce another variable that is incremented in each iteration, and print its value after the loop, along with the final value of `amount`:

```
amount = 1000
while amount < 2000:
    amount = 1.05 * amount
    year = year + 1
print(year, amount)
```

Reflection 4.23 *Make these changes and run the code again. Now what is printed?*

Oops, an error message is printed, telling us that the name `year` is undefined.

Reflection 4.24 *How do we fix the error?*

The problem is that we did not initialize the value of `year` before the loop. Therefore, the first time `year = year + 1` was executed, `year` was undefined on the right side of the assignment statement. Adding one statement before the loop fixes the problem:

```
amount = 1000
year = 0
while amount < 2000:
    amount = 1.05 * amount
    year = year + 1
print(year, amount)
```

Reflection 4.25 *Now what is printed by this block of code? In other words, how many years until the $1000 doubles?*

We will see some more examples of `while` loops later in this chapter, and again in Section 5.6.

Exercises

4.3.1* Suppose you put $1000 into the bank and you get a 3% interest rate compounded annually. How would you use a `while` loop to determine how long it will take for your account to have at least $1200 in it?

4.3.2. Repeat the last question, but this time write a function

```
interest(amount, rate, target)
```

that takes the initial amount, the interest rate, and the target amount as parameters. The function should return the number of years it takes to reach the target amount.

4.3.3. Since `while` loops are more general than `for` loops, we can emulate the behavior of a `for` loop with a `while` loop. For example, we can emulate the behavior of the `for` loop

```
for counter in range(10):
    print(counter)
```

with the `while` loop

```
counter = 0
while counter < 10:
    print(counter)
    counter = counter + 1
```

(a) Create a trace table for each of the loops above to make sure you understand how they are equivalent.

(b) What happens if we omit `counter = 0` before the `while` loop? Why does this happen?

(c) What happens if we omit `counter = counter + 1` from the body of the `while` loop? What does the loop print?

(d) Show how to emulate the following `for` loop with a `while` loop:

```
for counter in range(3, 12):
    print(counter)
```

(e) Show how to emulate the following `for` loop with a `while` loop:

```
for counter in range(12, 3, -1):
    print(counter)
```

4.3.4* In the `profitTable` function on page 142, we used a `for` loop to indirectly consider all ticket prices divisible by a half dollar. Rewrite this function so that it instead uses a `while` loop that increments `price` by $0.50 in each iteration.

4.3.5. A zombie can convert two people into zombies everyday. Starting with just one zombie, how long would it take for the entire world population (7 billion people) to become zombies? Write a function

```
zombieApocalypse()
```

that returns the answer to this question.

4.3.6* Tribbles increase at the rate of 50% per hour (rounding down if there are an odd number of them). How long would it take 10 tribbles to reach a population size of 1 million? Write a function

```
tribbleApocalypse()
```

that returns the answer to this question.

4.3.7. Vampires can each convert v people a day into vampires. However, there is a band of vampire hunters that can kill k vampires per day. If a coven of vampires starts with `vampires` members, how long before a town with a population of `people` becomes a town with no humans left in it? Write a function

```
vampireApocalypse(v, k, vampires, people)
```

that returns the answer to this question.

4.3.8. An amoeba can split itself into two once every h hours. How many hours does it take for a single amoeba to become `target` amoebae? Write a function

```
amoebaGrowth(h, target)
```

that returns the answer to this question.

4.3.9. Write a function

```
virus(rate, target)
```

that returns the number of days until `target` people are infected by a virus, assuming one person is initially infected and the number infected grows by the given `rate` each day.

4.3.10. Suppose each person newly infected by a virus is able to infect `R` additional people. `R` is called the *reproduction number* of the virus. (Think of this as a one-time event; the person does not infect `R` additional people every day.) Write a function

```
virus2(R, target)
```

that returns the number of days until `target` people are infected, assuming only one person is initially infected.

*4.4 CONTINUOUS MODELS

This section is available on the book website.

*4.5 NUMERICAL ANALYSIS

This section is available on the book website.

4.6 SUMMING UP

Although we have solved a variety of different problems in this chapter, almost all of the functions that we have designed have the same basic format:

```
def accumulator(_______):
    total = ____                    # initialize the accumulator
    for index in range(____):       # iterate some number of times
        total = total + _______     # add something to the accumulator
    return total                    # return final accumulator value
```

The functions we designed differ primarily in what is added to the accumulator (the red statement) in each iteration of the loop. Let's look at three of these functions in particular: the `pond` function from page 135, the `countLinks` function from page 138, and the solution to Exercise 4.1.30 from page 148, shown below.

```
def growth(finalAge):
    height = 95
    for age in range(4, finalAge + 1):
        height = height + 6
    return height
```

In the `growth` function, a constant value is added to the accumulator in each iteration:

```
height = height + 6
```

In the `countLinks` function, the value of the index variable, minus one, is added to the accumulator:

```
newLinks = node - 1
totalLinks = totalLinks + newLinks
```

And in the `pond` function, a factor of the accumulator itself is added in each iteration:

```
population = population + 0.08 * population   # ignoring "- 1500"
```

These three types of accumulators grow in three different ways. Adding a constant value to the accumulator in each iteration, as in the `growth` function, results in a final sum that is equal to the number of iterations times the constant value. In other words, if the initial value is a, the constant added value is c, and the number of iterations is n, then the final value of the accumulator is $a + cn$. (In the `growth` function, $a = 95$ and $c = 6$, so the final sum is $95 + 6n$.) As n increases, cn increases by a constant amount. This is called ***linear growth***, and is illustrated by the blue line in Figure 4.5.

Adding the value of the index variable to the accumulator, as in the `countLinks` function, leads to faster growth. In `countLinks`, the final value of the accumulator is

$$1 + 2 + 3 + \cdots + (n - 1)$$

which is equal to

$$\frac{1}{2} \cdot n \cdot (n - 1) = \frac{n^2 - n}{2}.$$

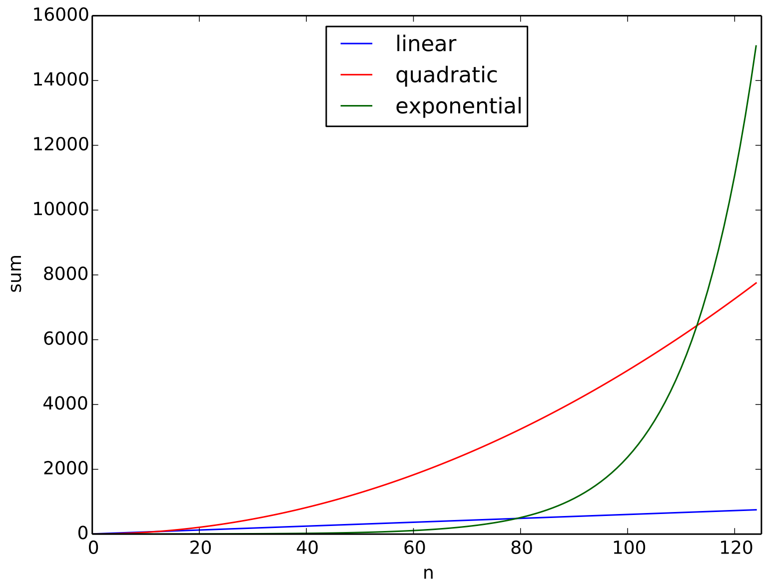


Figure 4.5 An illustration of linear, quadratic, and exponential growth. The curves are generated by accumulators adding 6, the index variable, and 1.08 times the accumulator, respectively, in each iteration.

Tangent 4.1 explains two clever ways to derive this result. Since this sum is proportional to n^2, we say that it exhibits ***quadratic growth***, as shown by the red curve in Figure 4.5. This sum is actually quite handy to know, and it will surface again in Chapter 10.

Finally, adding a factor of the accumulator to itself in each iteration, as in the `pond` function, results in even faster growth. In the `pond` function, if we add `0.08 * population` to `population` in each iteration, the accumulator variable will be equal to the initial value of `population` times 1.08^n at the end of n iterations of the loop. For this reason, we call this ***exponential growth***, which is illustrated by the green curve in Figure 4.5. Notice that, as n gets larger, exponential growth quickly outpaces the other two curves, even when the power of n is small, like 1.08.

So although all accumulator algorithms look more or less alike, the effects of the accumulators can be strikingly different. Understanding the relative rates of these different types of growth is quite important in a variety of fields, not just computer science. For example, mistaking an exponentially growing epidemic for a linearly growing one can be a life or death mistake!

These classes of growth can also be applied to the time complexity of algorithms, as we saw briefly in Section 1.2 and will see more in later chapters. When applied in this way, n represents the size of the algorithm's input and the y-axis represents the

Tangent 4.1: Triangular numbers

There are a few nice tricks to figure out the value of the sum

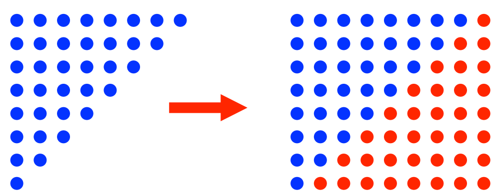
$$1 + 2 + 3 + \cdots + (n-2) + (n-1) + n$$

for any positive integer n. The first technique is to add the numbers in the sum from the outside in. Notice that the sum of the first and last numbers is $n+1$. Then, coming in one position from both the left and right, we find that $(n-1)+2 = n+1$ as well. Next, $(n-2)+3 = n+1$. This pattern will obviously continue, as we are subtracting 1 from the number on the left and adding 1 to the number on the right. In total, there is one instance of $n+1$ for every two terms in the sum, for a total of $n/2$ instances of $n+1$. Therefore, the sum is

$$1 + 2 + 3 + \cdots + (n-2) + (n-1) + n = \frac{n}{2}(n+1) = \frac{n(n+1)}{2}.$$

For example, $1+2+3+\cdots+8 = (8 \cdot 9)/2 = 36$ and $1+2+3+\cdots+1000 = (1000 \cdot 1001)/2 = 500{,}500$.

The second technique to derive this result is more visual. Depict each number in the sum as a column of circles, as shown on the left below with $n = 8$.



The first column has $n = 8$ circles, the second has $n - 1 = 7$, etc. So the total number of circles in this triangle is equal to the value we are seeking. Now make an exact duplicate of this triangle, and place its mirror image to the right of the original triangle, as shown on the right above. The resulting rectangle has n rows and $n+1$ columns, so there are a total of $n(n+1)$ circles. Since the number of circles in this rectangle is exactly twice the number in the original triangle, the number of circles in the original triangle is $n(n+1)/2$. Based on this representation, numbers like 36 and 500,500 that are sums of this form are called *triangular numbers*.

number of elementary steps required by the algorithm to compute the corresponding output. Algorithms that exhibit linear or quadratic time complexity are generally considered to be acceptable algorithms, while those exhibiting exponential time complexity are essentially worthless on all but the smallest inputs.

Exercises

4.6.1. Decide whether each of the following accumulators exhibits linear, quadratic, or exponential growth.

(a)*
```
total = 0
for count in range(n):
    total = total + count * 2
```

(b)
```
total = 10
for count in range(n):
    total = total + count / 2
```

(c)*
```
total = 1
for count in range(n):
    total = total + total
```

(d)
```
total = 0
for count in range(n):
    total = total + 1.2 * total
```

(e)
```
total = 0
for count in range(n):
    total = total + 0.01
```

(f)
```
total = 10
for count in range(n):
    total = 1.2 * total
```

4.6.2. Look at Figure 4.5. For values of n less than about 80, the fast-growing exponential curve is actually below the other two. Explain why.

4.6.3. Write a program to generate Figure 4.5.

4.7 FURTHER DISCOVERY

The epigraph of this chapter is from a TED talk given by Stephen Hawking in 2008. You can watch it yourself at

`www.ted.com/talks/stephen_hawking_asks_big_questions_about_the_universe` .

If you are interested in learning more about population dynamics models, and computational modeling in general, a great source is *Introduction to Computational Science* [61] by Angela Shiflet and George Shiflet.

*4.8 PROJECTS

This section is available on the book website.