13 DYNAMIC PROGRAMMING

Dynamic programming was invented by Richard Bellman in the 1950s. Don't try to infer anything about the technique from its name. As Bellman described it, the name "dynamic programming" was chosen to hide from governmental sponsors "the fact that I was really doing mathematics… [the phrase dynamic programming] was something not even a Congressman could object to."[80]

Dynamic programming is a method for efficiently solving problems that exhibit the characteristics of overlapping subproblems and optimal substructure. Fortunately, many optimization problems exhibit these characteristics.

A problem has **optimal substructure** if a globally optimal solution can be found by combining optimal solutions to local subproblems. We've already looked at a number of such problems. Merge sort, for example, exploits the fact that a list can be sorted by first sorting sublists and then merging the solutions.

A problem has **overlapping subproblems** if an optimal solution involves solving the same problem multiple times. Merge sort does not exhibit this property. Even though we are performing a merge many times, we are merging different lists each time.

It's not immediately obvious, but the 0/1 knapsack problem exhibits both of these properties. First, however, we digress to look at a problem where the optimal substructure and overlapping subproblems are more obvious.

13.1 Fibonacci Sequences, Revisited

In Chapter 4, we looked at a straightforward recursive implementation of the Fibonacci function:

```
def fib(n):
    """Assumes n is an int >= 0
       Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

While this implementation of the recurrence is obviously correct, it is terribly inefficient. Try, for example, running `fib(120)`, but don't wait for it to complete. The complexity of the implementation

is a bit hard to derive, but it is roughly O(fib(n)). That is, its growth is proportional to the growth in the value of the result, and the growth rate of the Fibonacci sequence is substantial. For example, `fib(120)` is 8,670,007,398,507,948,658,051,921. If each recursive call took a nanosecond, `fib(120)` would take about 250,000 years to finish.

Let's try and figure out why this implementation takes so long. Given the tiny amount of code in the body of `fib`, it's clear that the problem must be the number of times that `fib` calls itself. As an example, look at the tree of calls associated with the invocation `fib(6)`.

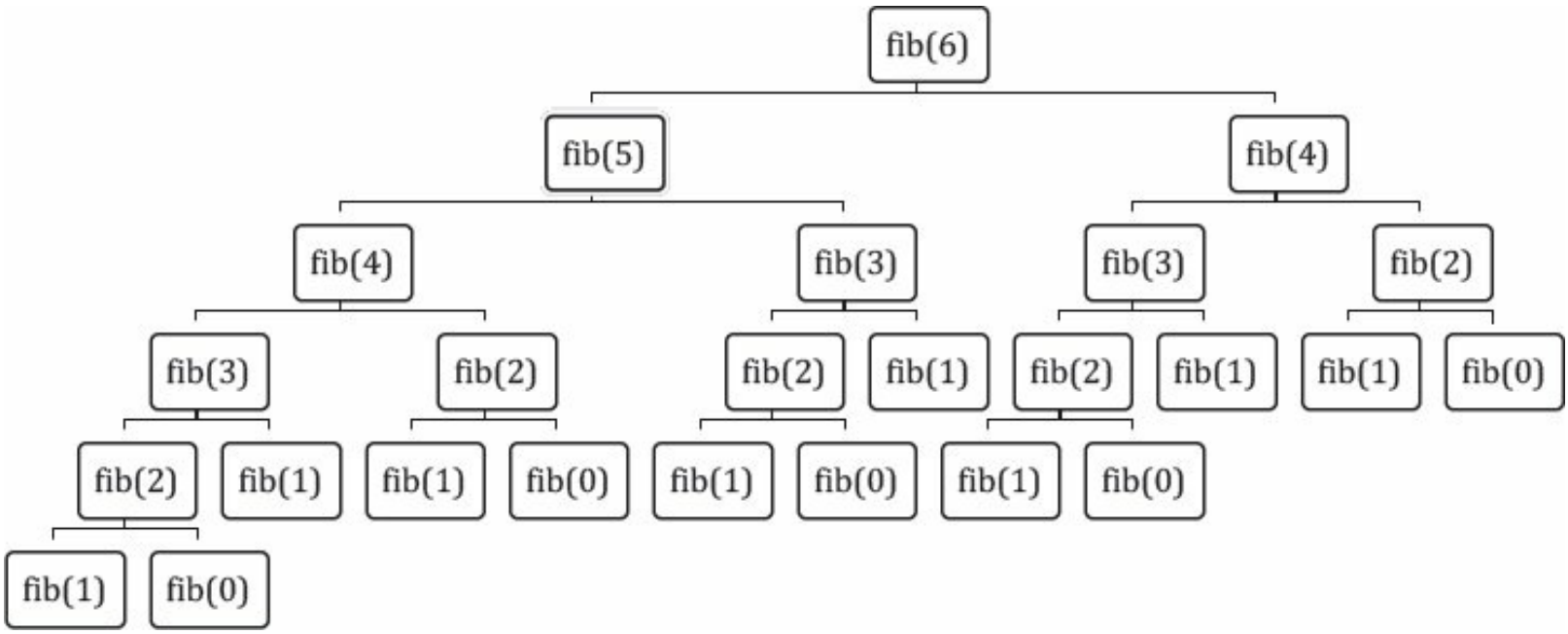

Figure 13.1 Tree of calls for recursive Fibonacci

Notice that we are computing the same values over and over again. For example, `fib` gets called with 3 three times, and each of these calls provokes four additional calls of `fib`. It doesn't require a genius to think that it might be a good idea to record the value returned by the first call, and then look it up rather than compute it each time it is needed. This is called **memoization**, and is the key idea behind dynamic programming.

Figure 13.2 contains an implementation of Fibonacci based on this idea. The function `fastFib` has a parameter, `memo`, that it uses to keep track of the numbers it has already evaluated. The parameter has a default value, the empty dictionary, so that clients of `fastFib` don't have to worry about supplying an initial value for `memo`. When `fastFib` is called with an `n > 1`, it attempts to look up `n` in `memo`. If it is not there (because this is the first time `fastFib` has been called with that value), an exception is raised. When this happens, `fastFib` uses the normal Fibonacci recurrence, and then stores the result in `memo`.

```
def fastFib(n, memo = {}):
    """Assumes n is an int >= 0, memo used only by recursive calls
       Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    try:
        return memo[n]
    except KeyError:
        result = fastFib(n-1, memo) + fastFib(n-2, memo)
        memo[n] = result
        return result
```

Figure 13.2 Implementing Fibonacci using a memo

If you try running `fastFib`, you will see that it is indeed quite fast: `fib(120)` returns almost instantly. What is the complexity of `fastFib`? It calls `fib` exactly once for each value from 0 to `n`. Therefore, under the assumption that dictionary lookup can be done in constant time, the time complexity of `fastFib(n)` is O(n).[81]

13.2 Dynamic Programming and the 0/1 Knapsack Problem

One of the optimization problems we looked at in Chapter 12 was the 0/1 knapsack problem. Recall that we looked at a greedy algorithm that ran in n log n time, but was not guaranteed to find an optimal solution. We also looked at a brute-force algorithm that was guaranteed to find an optimal solution, but ran in exponential time. Finally, we discussed the fact that the problem is inherently exponential in the size of the input. In the worst case, one cannot find an optimal solution without looking at all possible answers.

Fortunately, the situation is not as bad as it seems. Dynamic programming provides a practical method for solving most 0/1 knapsack problems in a reasonable amount of time. As a first step in deriving such a solution, we begin with an exponential solution based on exhaustive enumeration. The key idea is to think about exploring the space of possible solutions by constructing a rooted binary tree that enumerates all states that satisfy the weight constraint.

A **rooted binary tree** is an acyclic directed graph in which

- There is exactly one node with no parents. This is called the **root**.
- Each non-root node has exactly one parent.
- Each node has at most two children. A childless node is called a **leaf**.

Each node in the search tree for the 0/1 knapsack problem is labeled with a quadruple that denotes a partial solution to the knapsack problem. The elements of the quadruple are:

- A set of items to be taken,

- The list of items for which a decision has not been made,
- The total value of the items in the set of items to be taken (this is merely an optimization, since the value could be computed from the set), and
- The remaining space in the knapsack. (Again, this is an optimization, since it is merely the difference between the weight allowed and the weight of all the items taken so far.)

The tree is built top-down starting with the root.[82] One element is selected from the still-to-be-considered items. If there is room for that item in the knapsack, a node is constructed that reflects the consequence of choosing to take that item. By convention, we draw that node as the left child. The right child shows the consequences of choosing not to take that item. The process is then applied recursively until either the knapsack is full or there are no more items to consider. Because each edge represents a decision (to take or not to take an item), such trees are called **decision trees**.[83]

Figure 13.3 is a table describing a set of items.

Name	Value	Weight
a	6	3
b	7	3
c	8	2
d	9	5

Figure 13.3 Table of items with values and weights

Figure 13.4 is a decision tree for deciding which of those items to take under the assumption that the knapsack has a maximum weight of 5. The root of the tree (node 0) has a label <{}, [a,b,c,d], 0, 5>, indicating that no items have been taken, all items remain to be considered, the value of the items taken is 0, and a weight of 5 is still available. Node 1 indicates that item a has been taken, [b,c,d] remain to be considered, the value of the items taken is 6, and the knapsack can hold another 2 pounds. Node 1 has no left child since item b, which weighs 3 pounds, would not fit in the knapsack.

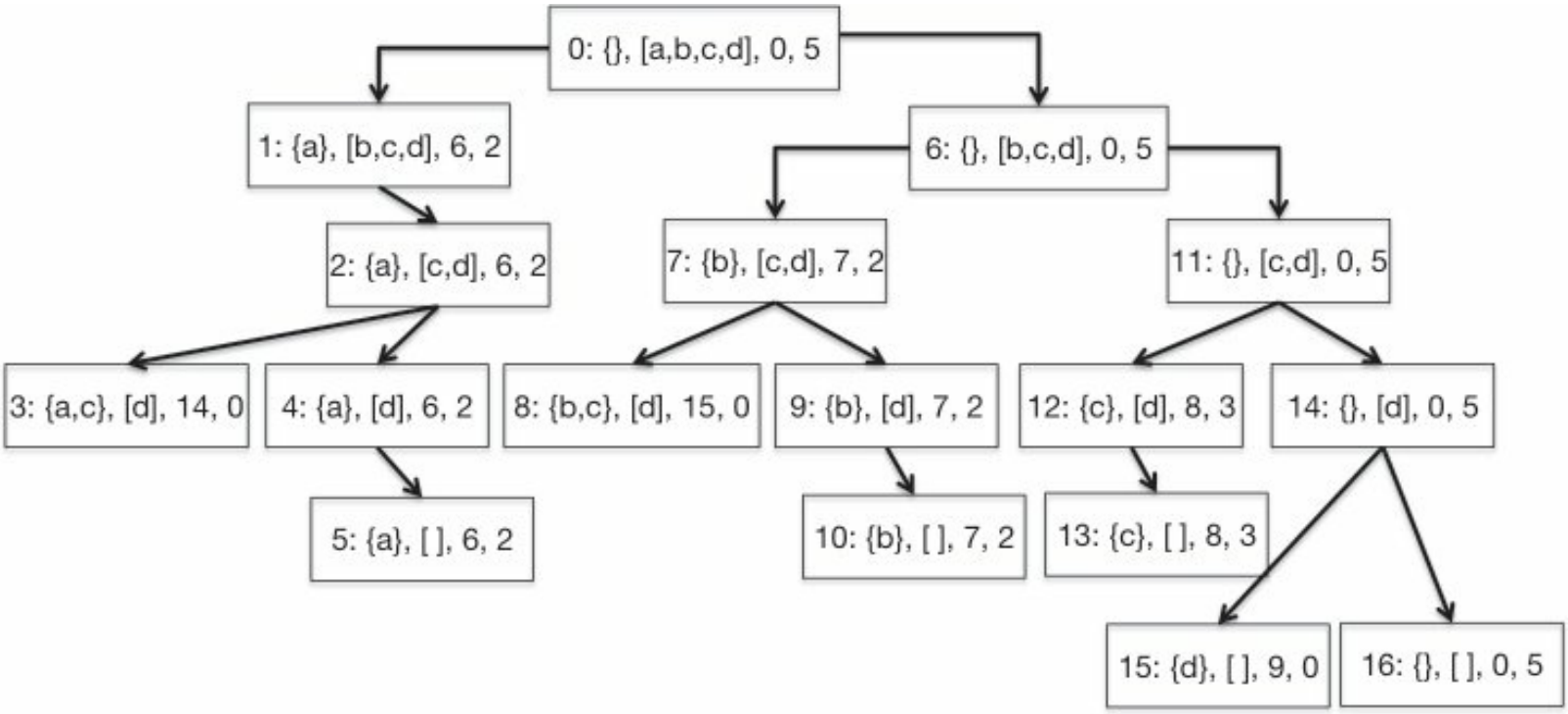

Figure 13.4 Decision tree for knapsack problem

In Figure 13.4, the numbers that precede the colon in each node indicate one order in which the nodes could be generated. This particular ordering is called left-first depth-first. At each node we attempt to generate a left node. If that is impossible, we attempt to generate a right node. If that too is impossible, we back up one node (to the parent) and repeat the process. Eventually, we find ourselves having generated all descendants of the root, and the process halts. When the process halts, each combination of items that could fit in the knapsack has been generated, and any leaf node with the greatest value represents an optimal solution. Notice that for each leaf node, either the second element is the empty list (indicating that there are no more items to consider taking) or the fourth element is 0 (indicating that there is no room left in the knapsack).

Unsurprisingly (especially if you read Chapter 12), the natural implementation of a depth-first tree search is recursive. Figure 13.5 contains such an implementation.

```
def maxVal(toConsider, avail):
    """Assumes toConsider a list of items, avail a weight
       Returns a tuple of the total value of a solution to the
         0/1 knapsack problem and the items of that solution"""
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getWeight() > avail:
        #Explore right branch only
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        #Explore left branch
        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getWeight())
        withVal += nextItem.getValue()
        #Explore right branch
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
        #Choose better branch
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result
```

Figure 13.5 Using a decision tree to solve a knapsack problem

The implementation uses class `Item` from Figure 12.2. The function `maxVal` returns two values, the set of items chosen and the total value of those items. It is called with two arguments, corresponding to the second and fourth elements of the labels of the nodes in the tree:

- `toConsider`. Those items that nodes higher up in the tree (corresponding to earlier calls in the recursive call stack) have not yet considered.
- `avail`. The amount of space still available.

Notice that the implementation of `maxVal` does not build the decision tree and then look for an optimal node. Instead, it uses the local variable `result` to record the best solution found so far. The code in Figure 13.6 can be used to test `maxVal`.

When `smallTest` (which uses the values in Figure 13.3) is run it prints a result indicating that node 8 in Figure 13.4 is an optimal solution:

```
<c, 8, 2>
<b, 7, 3>
Total value of items taken = 15
```

```
def smallTest():
    names = ['a', 'b', 'c', 'd']
    vals = [6, 7, 8, 9]
    weights = [3, 3, 2, 5]
    Items = []
    for i in range(len(vals)):
        Items.append(Item(names[i], vals[i], weights[i]))
    val, taken = maxVal(Items, 5)
    for item in taken:
        print(item)
    print('Total value of items taken =', val)

def buildManyItems(numItems, maxVal, maxWeight):
    items = []
    for i in range(numItems):
        items.append(Item(str(i),
                          random.randint(1, maxVal),
                          random.randint(1, maxWeight)))
    return items

def bigTest(numItems):
    items = buildManyItems(numItems, 10, 10)
    val, taken = maxVal(items, 40)
    print('Items Taken')
    for item in taken:
        print(item)
    print('Total value of items taken =', val)
```

Figure 13.6 Testing the decision tree-based implementation

The functions `buildManyItems` and `bigTest` can be used to test `maxVal` on randomly generated sets of items. Try `bigTest(10)`. Now try `bigTest(40)`. After you get tired of waiting for it to return, stop the computation and ask yourself what is going on.

Let's think about the size of the tree we are exploring. Since at each level of the tree we are deciding to keep or not keep one item, the maximum depth of the tree is `len(items)`. At level 0 we have only one node, at level 1 up to two nodes, at level 2 up to four nodes, at level 3 up to eight nodes. At level 39 we have up to 2^{39} nodes. No wonder it takes a long time to run!

What should we do about this? Let's start by asking whether this program has anything in common with our first implementation of Fibonacci. In particular, is there optimal substructure and are there overlapping subproblems?

Optimal substructure is visible both in Figure 13.4 and in Figure 13.5. Each parent node combines the solutions reached by its children to derive an optimal solution for the subtree rooted at that parent. This is reflected in Figure 13.5 by the code following the comment `#Choose better branch`.

Are there also overlapping subproblems? At first glance, the answer seems to be "no." At each level of the tree we have a different set of available items to consider. This implies that if common subproblems do exist, they must be at the same level of the tree. And indeed, at each level of the tree each node has the same set of items to consider taking. However, we can see by looking at the labels in Figure 13.4 that each node at a level represents a different set of choices about the items considered higher in the tree.

Think about what problem is being solved at each node. The problem being solved is finding the optimal items to take from those left to consider, given the remaining available weight. The available weight depends upon the total weight of the items taken, but not on which items are taken or the total value of the items taken. So, for example, in Figure 13.4, nodes 2 and 7 are actually solving the same problem: deciding which elements of [c,d] should be taken, given that the available weight is 2.

The code in Figure 13.7 exploits the optimal substructure and overlapping subproblems to provide a dynamic programming solution to the 0/1 knapsack problem. An extra parameter, `memo`, has been added to keep track of solutions to subproblems that have already been solved. It is implemented using a dictionary with a key constructed from the length of `toConsider` and the available weight. The expression `len(toConsider)` is a compact way of representing the items still to be considered. This works because items are always removed from the same end (the front) of the list `toConsider`.

Figure 13.8 shows the number of calls made when we ran the code on problems of various sizes. The growth is hard to quantify, but it is clearly far less than exponential.[84] But how can this be, since we know that the 0/1 knapsack problem is inherently exponential in the number of items? Have we found a way to overturn fundamental laws of the universe? No, but we have discovered that computational complexity can be a subtle notion.[85]

The running time of `fastMaxVal` is governed by the number of distinct pairs, `<toConsider, avail>`, generated. This is because the decision about what to do next depends only upon the items still available and the total weight of the items already taken.

```
def fastMaxVal(toConsider, avail, memo = {}):
    """Assumes toConsider a list of items, avail a weight
         memo supplied by recursive calls
       Returns a tuple of the total value of a solution to the
         0/1 knapsack problem and the items of that solution"""
    if (len(toConsider), avail) in memo:
        result = memo[(len(toConsider), avail)]
    elif toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getWeight() > avail:
        #Explore right branch only
        result = fastMaxVal(toConsider[1:], avail, memo)
    else:
        nextItem = toConsider[0]
        #Explore left branch
        withVal, withToTake =\
                 fastMaxVal(toConsider[1:],
                            avail - nextItem.getWeight(), memo)
        withVal += nextItem.getValue()
        #Explore right branch
        withoutVal, withoutToTake = fastMaxVal(toConsider[1:],
                                               avail, memo)
        #Choose better branch
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    memo[(len(toConsider), avail)] = result
    return result
```

Figure 13.7 Dynamic programming solution to knapsack problem

len(Items)	Number of items selected	Number of calls
4	4	31
8	6	337
16	9	1,493
32	12	3,650
64	19	8,707
128	27	18.306
256	40	36,675

Figure 13.8 Performance of dynamic programming solution

The number of possible values of `toConsider` is bounded by `len(items)`. The number of possible values of `avail` is more difficult to characterize. It is bounded from above by the maximum number of distinct totals of weights of the items that the knapsack can hold. If the knapsack can hold at most n items (based on the capacity of the knapsack and the weights of the available items), `avail` can take on at most 2^n different values. In principle, this could be a rather large number. However, in practice, it is not usually so large. Even if the knapsack has a large capacity, if the weights of the items are chosen from a reasonably small set of possible weights, many sets of items will have the same total weight, greatly reducing the running time.

This algorithm falls into a complexity class called **pseudo-polynomial**. A careful explanation of this concept is beyond the scope of this book. Roughly speaking, `fastMaxVal` is exponential in the number of bits needed to represent the possible values of `avail`.

To see what happens when the values of `avail` are chosen from a considerably larger space, change the call to `maxVal` in the function `bigTest` in Figure 13.6 to

```
val, taken = fastMaxVal(items, 1000)
```

Finding a solution now takes 1,802,817 calls of `fastMaxVal` when the number of items is 256.

To see what happens when the weights are chosen from an enormous space, we can choose the possible weights from the positive reals rather than the positive integers. To do this, replace the line,

```
items.append(Item(str(i),
                  random.randint(1, maxVal),
                  random.randint(1, maxWeight)))
```

in `buildManyItems` by the line

```
items.append(Item(str(i),
                  random.randint(1, maxVal),
                  random.randint(1, maxWeight)*random.random()))
```

Each time it is called, `random.random()` returns a random floating point number between `0.0` and `1.0`, so there are, for all intents and purposes, an infinite number of possible weights. Don't hold your breath waiting for this last test to finish. Dynamic programming may be a miraculous technique in the common sense of the word,[86] but it is not capable of performing miracles in the liturgical sense.

13.3 Dynamic Programming and Divide-and-Conquer

Like divide-and-conquer algorithms, dynamic programming is based upon solving independent subproblems and then combining those solutions. There are, however, some important differences.

Divide-and-conquer algorithms are based upon finding subproblems that are substantially smaller than the original problem. For example, merge sort works by dividing the problem size in half at each step. In contrast, dynamic programming involves solving problems that are only slightly smaller than the original problem. For example, computing the 19^{th} Fibonacci number is not a substantially smaller problem than computing the 20^{th} Fibonacci number.

Another important distinction is that the efficiency of divide-and-conquer algorithms does not depend upon structuring the algorithm so that identical problems are solved repeatedly. In contrast, dynamic programming is efficient only when the number of distinct subproblems is significantly smaller than the total number of subproblems.

---

[80] As quoted in Stuart Dreyfus "Richard Bellman on the Birth of Dynamic Programming," *Operations Research*, vol. 50, no. 1 (2002).

[81] Though cute and pedagogically interesting, this is not the best way to implement Fibonacci. There is a simple linear-time iterative implementation.

[82] It may seem odd to put the root of a tree at the top, but that is the way that mathematicians and computer scientists usually draw them. Perhaps it is evidence that those folks do not spend enough time contemplating nature.

[83] Decision trees, which need not be binary, provide a structured way to explore the consequences of making a series of sequential decisions. They are used extensively in many fields.

[84] Since $2^{128} = 340{,}282{,}366{,}920{,}938{,}463{,}463{,}374{,}607{,}431{,}768{,}211{,}456$

[85] OK, "discovered" may be too strong a word. People have known this for a long time. You probably figured it out around Chapter 9.

[86] Extraordinary and bringing welcome consequences.

14 RANDOM WALKS AND MORE ABOUT DATA VISUALIZATION

This book is about using computation to solve problems. Thus far, we have focused our attention on problems that can be solved by a **deterministic program**. A program is deterministic if whenever it is run on the same input, it produces the same output. Such computations are highly useful, but clearly not sufficient to tackle some kinds of problems. Many aspects of the world in which we live can be accurately modeled only as **stochastic processes**.[87] A process is stochastic if its next state can depend upon some random element. The outcome of a stochastic process is usually uncertain. Therefore, we can rarely make definitive statements about what they will do. Instead, we make probabilistic statements about what they might do. The rest of this book deals with building programs that help to understand uncertain situations. Many of these programs will be simulation models.

A simulation mimics the activity of a real system. For example, the code in Figure 8.11 simulates a person making a series of mortgage payments. Think of that code as an experimental device, called a **simulation model**, that provides useful information about the possible behaviors of the system being modeled. Among other things, simulations are widely used to predict a future state of a physical system (e.g., the temperature of the planet 50 years from now), and in lieu of physical experiments that would be too expensive, time consuming, or dangerous to perform (e.g., the impact of a change in the tax code).

It is important to remember that simulation models, like all models, are only an approximation of reality. One can never be sure that the actual system will behave in the way predicted by the model. In fact, one can usually be pretty confident that the actual system will not behave exactly as predicted by the model. For example, not every borrower will make all mortgage payments on time. It is a commonly quoted truism that "all models are wrong, but some are useful."[88]

14.1 Random Walks

In 1827, the Scottish botanist Robert Brown observed that pollen particles suspended in water seemed to float around at random. He had no plausible explanation for what came to be known as Brownian motion, and made no attempt to model it mathematically.[89] A clear mathematical model of the phenomenon was first presented in 1900 in Louis Bachelier's doctoral thesis, *The Theory of Speculation*. However, since this thesis dealt with the then disreputable problem of understanding

financial markets, it was largely ignored by respectable academics. Five years later, a young Albert Einstein brought this kind of stochastic thinking to the world of physics with a mathematical model almost the same as Bachelier's and a description of how it could be used to confirm the existence of atoms.[90] For some reason, people seemed to think that understanding physics was more important than making money, and the world started paying attention. Times were certainly different.

Brownian motion is an example of a **random walk**. Random walks are widely used to model physical processes (e.g., diffusion), biological processes (e.g., the kinetics of displacement of RNA from heteroduplexes by DNA), and social processes (e.g., movements of the stock market).

In this chapter we look at random walks for three reasons:

- Random walks are intrinsically interesting and widely used.
- It provides us with a good example of how to use abstract data types and inheritance to structure programs in general and simulation models in particular.
- It provides an opportunity to introduce a few more features of Python and to demonstrate some additional techniques for producing plots.

14.2 The Drunkard's Walk

Let's look at a random walk that actually involves walking. A drunken farmer is standing in the middle of a field, and every second the farmer takes one step in a random direction. What is her (or his) expected distance from the origin in 1000 seconds? If she takes many steps, is she likely to move ever farther from the origin, or is she more likely to wander back to the origin over and over, and end up not far from where she started? Let's write a simulation to find out.

Before starting to design a program, it is always a good idea to try to develop some intuition about the situation the program is intended to model. Let's start by sketching a simple model of the situation using Cartesian coordinates. Assume that the farmer is standing in a field where the grass has, mysteriously, been cut to resemble a piece of graph paper. Assume further that each step the farmer takes is of length one and is parallel to either the x-axis or y-axis.

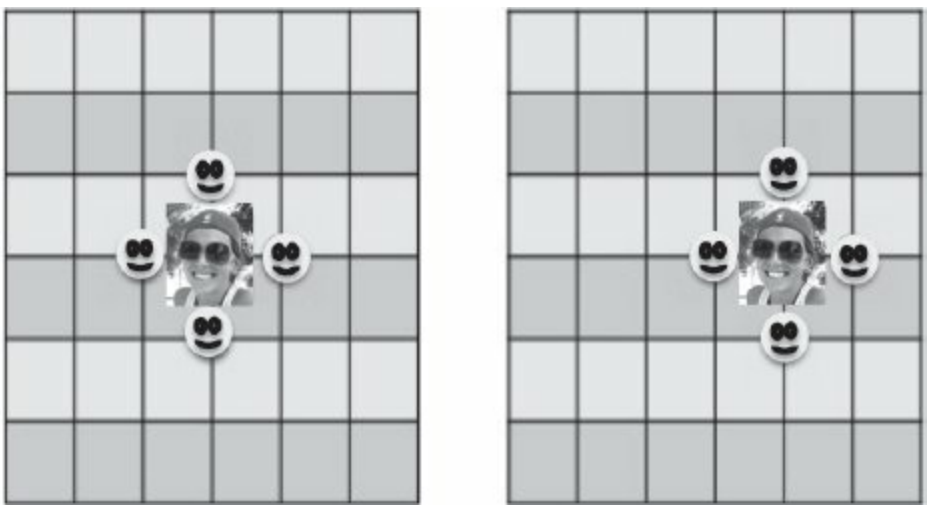

Figure 14.1 An unusual farmer

The picture on the left of Figure 14.1 depicts a farmer[91] standing in the middle of the field. The

smiley faces indicate all the places the farmer might be after one step. Notice that after one step she is always exactly one unit away from where she started. Let's assume that she wanders eastward from her initial location on her first step. How far away might she be from her initial location after her second step?

Looking at the smiley faces in the picture on the right, we see that with a probability of 0.25 she will be 0 units away, and with a probability of 0.25 she will be 2 units away, with a probability of 0.5 she will be $\sqrt{2}$ units away.[92] So, on average she will be farther away after two steps than after one step. What about the third step? If the second step is to the top or bottom smiley face, the third step will bring the farmer closer to the origin half the time and farther half the time. If the second step is to the left smiley face (the origin), the third step will be away from the origin. If the second step is to the right smiley face, the third step will be closer to the origin a quarter of the time, and farther away three quarters of the time.

It seems as if the more steps the drunk takes, the greater the expected distance from the origin. We could continue this exhaustive enumeration of possibilities and perhaps develop a pretty good intuition about how this distance grows with respect to the number of steps. However, it is getting pretty tedious, so it seems like a better idea to write a program to do it for us.

Let's begin the design process by thinking about some data abstractions that are likely to be useful in building this simulation and perhaps simulations of other kinds of random walks. As usual, we should try to invent types that correspond to the kinds of things that appear in the situation we are attempting to model. Three obvious types are `Location`, `Field`, and `Drunk`. As we look at the classes providing these types, it is worthwhile to think about what each might imply about the kinds of simulation models they will allow us to build.

Let's start with `Location`, Figure 14.2. This is a simple class, but it does embody two important decisions. It tells us that the simulation will involve at most two dimensions. E.g., the simulation will not model changes in altitude. This is consistent with the pictures above. Also, since the values supplied for `deltaX` and `deltaY` could be floats rather than integers, there is no built-in assumption in this class about the set of directions in which a drunk might move. This is a generalization of the informal model in which each step was of length one and was parallel to the x-axis or y-axis.

Class `Field`, Figure 14.2, is also quite simple, but it too embodies notable decisions. It simply maintains a mapping of drunks to locations. It places no constraints on locations, so presumably a `Field` is of unbounded size. It allows multiple drunks to be added into a `Field` at random locations. It says nothing about the patterns in which drunks move, nor does it prohibit multiple drunks from occupying the same location or moving through spaces occupied by other drunks.

```
class Location(object):
    def __init__(self, x, y):
        """x and y are numbers"""
        self.x, self.y = x, y

    def move(self, deltaX, deltaY):
        """deltaX and deltaY are numbers"""
        return Location(self.x + deltaX, self.y + deltaY)
```

```
    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distFrom(self, other):
        ox, oy = other.x, other.y
        xDist, ydist = self.x - ox, self.y - oy
        return (xDist**2 + yDist**2)**0.5

    def __str__(self):
        return '<' + str(self.x) + ', ' + str(self.y) + '>'

class Field(object):
    def __init__(self):
        self.drunks = {}

    def addDrunk(self, drunk, loc):
        if drunk in self.drunks:
            raise ValueError('Duplicate drunk')
        else:
            self.drunks[drunk] = loc

    def moveDrunk(self, drunk):
        if drunk not in self.drunks:
            raise ValueError('Drunk not in field')
        xDist, yDist = drunk.takeStep()
        currentLocation = self.drunks[drunk]
        #use move method of Location to get new location
        self.drunks[drunk] = currentLocation.move(xDist, yDist)

    def getLoc(self, drunk):
        if drunk not in self.drunks:
            raise ValueError('Drunk not in field')
        return self.drunks[drunk]
```

Figure 14.2 `Location` **and** `Field` **classes**

The classes `Drunk` and `UsualDrunk` in Figure 14.3 define the ways in which a drunk might wander through the field. In particular the value of `stepChoices` in `UsualDrunk` introduces the restriction that each step is of length one and is parallel to either the x-axis or y-axis. Since the function `random.choice` returns a randomly chosen member of the sequence that it is passed, each kind of step is equally likely and not influenced by previous steps. A bit later we will look at subclasses of `Drunk` with different kinds of behaviors.

```
import random

class Drunk(object):
    def __init__(self, name = None):
        """Assumes name is a str"""
        self.name = name

    def __str__(self):
        if self != None:
            return self.name
        return 'Anonymous'

class UsualDrunk(Drunk):
    def takeStep(self):
        stepChoices = [(0,1), (0,-1), (1, 0), (-1, 0)]
        return random.choice(stepChoices)
        return random.choice(stepChoices)
```

Figure 14.3 Classes defining Drunks

The next step is to use these classes to build a simulation that answers the original question. Figure 14.4 contains three functions used in this simulation.

The function `walk` simulates one walk of `numSteps` steps. The function `simWalks` calls `walk` to simulate `numTrials` walks of `numSteps` steps each. The function `drunkTest` calls `simWalks` to simulate walks of varying lengths.

The parameter `dClass` of `simWalks` is of type `class`, and is used in the first line of code to create a `Drunk` of the appropriate subclass. Later, when `drunk.takeStep` is invoked from `Field.moveDrunk`, the method from the appropriate subclass is automatically selected.

The function `drunkTest` also has a parameter, `dClass`, of type `class`. It is used twice, once in the call to `simWalks` and once in the first `print` statement. In the `print` statement, the built-in `class` attribute `__name__` is used to get a string with the name of the class.

```
def walk(f, d, numSteps):
    """Assumes: f a Field, d a Drunk in f, and numSteps an int >= 0.
       Moves d numSteps times; returns the distance between the
       final location and the location at the start of the  walk."""
    start = f.getLoc(d)
    for s in range(numSteps):
        f.moveDrunk(d)
    return start.distFrom(f.getLoc(d))

def simWalks(numSteps, numTrials, dClass):
    """Assumes numSteps an int >= 0, numTrials an int > 0,
         dClass a subclass of Drunk
       Simulates numTrials walks of numSteps steps each.
       Returns a list of the final distances for each trial"""
    Homer = dClass()
    origin = Location(0, 0)
    distances = []
    for t in range(numTrials):
        f = Field()
        f.addDrunk(Homer, origin)
        distances.append(round(walk(f, Homer, numTrials), 1))
    return distances

def drunkTest(walkLengths, numTrials, dClass):
    """Assumes walkLengths a sequence of ints >= 0
         numTrials an int > 0, dClass a subclass of Drunk
       For each number of steps in walkLengths, runs simWalks with
         numTrials walks and prints results"""
    for numSteps in walkLengths:
        distances = simWalks(numSteps, numTrials, dClass)
        print(dClass.__name__, 'random walk of', numSteps, 'steps')
        print(' Mean =', round(sum(distances)/len(distances), 4))
        print(' Max =', max(distances), 'Min =', min(distances))
```

Figure 14.4 The drunkard's walk (with a bug)

When we executed `drunkTest((10, 100, 1000, 10000), 100, UsualDrunk)`, it printed

```
UsualDrunk random walk of 10 steps
 Mean = 8.634
 Max = 21.6 Min = 1.4
```

```
UsualDrunk random walk of 100 steps
 Mean = 8.57
 Max = 22.0 Min = 0.0
UsualDrunk random walk of 1000 steps
 Mean = 9.206
 Max = 21.6 Min = 1.4
UsualDrunk random walk of 10000 steps
 Mean = 8.727
 Max = 23.5 Min = 1.4
```

This is surprising, given the intuition we developed earlier that the mean distance should grow with the number of steps. It could mean that our intuition is wrong, or it could mean that our simulation is buggy, or both.

The first thing to do at this point is to run the simulation on values for which we already think we know the answer, and make sure that what the simulation produces matches the expected result. Let's try walks of zero steps (for which the mean, minimum and maximum distances from the origin should all be 0) and one step (for which the mean, minimum and maximum distances from the origin should all be `1`).

When we ran `drunkTest((0,1), 100, UsualDrunk)`, we got the highly suspect result

```
UsualDrunk random walk of 0 steps
 Mean = 8.634
 Max = 21.6 Min = 1.4
UsualDrunk random walk of 1 steps
 Mean = 8.57
 Max = 22.0 Min = 0.0
```

How on earth can the mean distance of a walk of zero steps be over 8? We must have at least one bug in our simulation. After some investigation, the problem is clear. In `simWalks`, the function call `walk(f, Homer, numTrials)` should have been `walk(f, Homer, numSteps)`.

The moral here is an important one: Always bring some skepticism to bear when looking at the results of a simulation. Ask if the results are

plausible, and "smoke test"[93] the simulation on parameters for which you have a strong intuition about what the results should be.

When the corrected version of the simulation is run on our two simple cases, it yields exactly the expected answers:

```
UsualDrunk random walk of 0 steps
 Mean = 0.0
 Max = 0.0 Min = 0.0
UsualDrunk random walk of 1 steps
 Mean = 1.0
 Max = 1.0 Min = 1.0
```

When run on longer walks it printed

```
 UsualDrunk random walk of 10 steps
  Mean = 2.863
  Max = 7.2 Min = 0.0
 UsualDrunk random walk of 100 steps
  Mean = 8.296
  Max = 21.6 Min = 1.4
 UsualDrunk random walk of 1000 steps
  Mean = 27.297
  Max = 66.3 Min = 4.2
 UsualDrunk random walk of 10000 steps
  Mean = 89.241
  Max = 226.5 Min = 10.0
```

As anticipated, the mean distance from the origin grows with the number of steps.

Now let's look at a plot of the mean distances from the origin, Figure 14.5. To give a sense of how fast the distance is growing, we have placed on the plot a line showing the square root of the number of steps (and increased the number of steps to 100,000). The plot showing the square root of the number of steps versus the distance from the origin is a

straight line because we used a logarithmic scale on both axes.

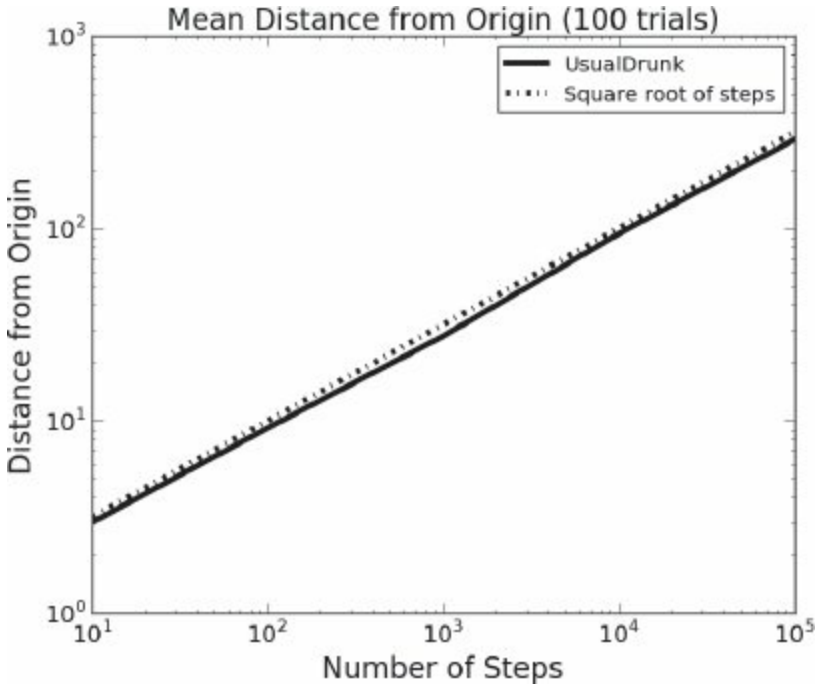

Figure 14.5 Distance from starting point versus steps taken

Does this plot provide any information about the expected final location of a drunk? It does tell us that on average the drunk will be somewhere on a circle with its center at the origin and with a radius equal to the expected distance from the origin. However, it tells us very little about where we might actually find the drunk at the end of any particular walk. We return to this topic in the next section.

14.3 Biased Random Walks

Now that we have a working simulation, we can start modifying it to investigate other kinds of random walks. Suppose, for example, that we want to consider the behavior of a drunken farmer in the northern hemisphere who hates the cold, and even in his drunken stupor is able to move twice as fast when his random movements take him in a southward direction. Or maybe a phototropic drunk who always moves towards the sun (east in the morning and west in the afternoon). These are examples of **biased random walks**. The walk is still stochastic, but there is a bias

in the outcome.

Figure 14.6 defines two additional subclasses of `Drunk`. In each case the specialization involves choosing an appropriate value for `stepChoices`. The function `simAll` iterates over a sequence of subclasses of `Drunk` to generate information about how each kind behaves.

```
class ColdDrunk(Drunk):
    def takeStep(self):
        stepChoices = [(0.0,1.0), (0.0,-2.0), (1.0, 0.0),\
                       (-1.0, 0.0)]
        return random.choice(stepChoices)

class EWDrunk(Drunk):
    def takeStep(self):
        stepChoices = [(1.0, 0.0), (-1.0, 0.0)]
        return random.choice(stepChoices)

def simAll(drunkKinds, walkLengths, numTrials):
    for dClass in drunkKinds:
        drunkTest(walkLengths, numTrials, dClass)
```

Figure 14.6 Subclasses of `Drunk` base class

When we ran

```
simAll((UsualDrunk, ColdDrunk, EWDrunk), (100,
1000), 10)
```

it printed

```
 UsualDrunk random walk of 100 steps
  Mean = 9.64
  Max = 17.2 Min = 4.2
 UsualDrunk random walk of 1000 steps
  Mean = 22.37
  Max = 45.5 Min = 4.5
 ColdDrunk random walk of 100 steps
  Mean = 27.96
  Max = 51.2 Min = 4.1
 ColdDrunk random walk of 1000 steps
  Mean = 259.49
  Max = 320.7 Min = 215.1
 EWDrunk random walk of 100 steps
  Mean = 7.8
  Max = 16.0 Min = 0.0
 EWDrunk random walk of 1000 steps
  Mean = 20.2
  Max = 48.0 Min = 4.0
```

It appears that our heat-seeking drunk moves away from the origin faster than the other two kinds of drunk. However, it is not easy to digest all of the information in this output. It is once again time to move away from textual output and start using plots.

Since we are showing a number of different kinds of drunks on the same plot, we will associate a distinct style with each type of drunk so that it is easy to differentiate among them. The style will have three aspects:

- The color of the line and marker,
- The shape of the marker, and
- The kind of the line, e.g., solid or dotted.

The class `styleIterator`, Figure 14.7, rotates through a sequence of styles defined by the argument to `styleIterator.__init__`.

```
class styleIterator(object):
    def __init__(self, styles):
        self.index = 0
        self.styles = styles

    def nextStyle(self):
        result = self.styles[self.index]
        if self.index == len(self.styles) - 1:
            self.index = 0
        else:
            self.index += 1
        return result
```

Figure 14.7 Iterating over styles

The code in Figure 14.8 is similar in structure to that in Figure 14.4. The `print` statements in `simDrunk` and `simAll1` contribute nothing to the result of the simulation. They are there because this simulation can take a rather long time to complete, and printing an occasional message indicating that progress is being made can be quite reassuring to a user who might be wondering if the program is actually making progress.

The code in Figure 14.8 produces the plot in Figure 14.9. Notice that both the x and y axes are on a **logarithmic scale**. This was done by calling the plotting functions `pylab.semilogx` and `pylab.semilogy`. These functions are always applied to the current figure.

```
def simDrunk(numTrials, dClass, walkLengths):
    meanDistances = []
    for numSteps in walkLengths:
        print('Starting simulation of', numSteps, 'steps')
        trials = simWalks(numSteps, numTrials, dClass)
        mean = sum(trials)/len(trials)
        meanDistances.append(mean)
    return meanDistances

def simAll1(drunkKinds, walkLengths, numTrials):
    styleChoice = styleIterator(('m-', 'r:', 'k-.'))
    for dClass in drunkKinds:
        curStyle = styleChoice.nextStyle()
        print('Starting simulation of', dClass.__name__)
        means = simDrunk(numTrials, dClass, walkLengths)
        pylab.plot(walkLengths, means, curStyle,
                   label = dClass.__name__)
    pylab.title('Mean Distance from Origin ('
                + str(numTrials) + ' trials)')
    pylab.xlabel('Number of Steps')
    pylab.ylabel('Distance from Origin')
    pylab.legend(loc = 'best')
    pylab.semilogx()
    pylab.semilogy()

simAll1((UsualDrunk, ColdDrunk, EWDrunk),
        (10,100,1000,10000,100000), 100)
```

Figure 14.8 Plotting the walks of different drunks

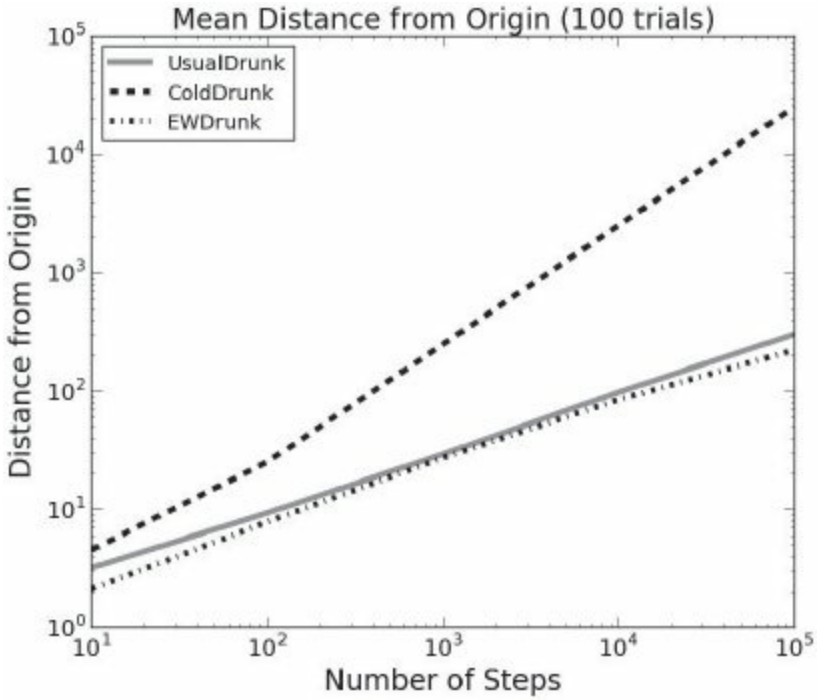

Figure 14.9 Mean distance for different kinds of drunks

The usual drunk and the phototropic drunk (`EWDrunk`) seem to be moving away from the origin at approximately the same pace, but the heat-seeking drunk (`ColdDrunk`) seems to be moving away orders of magnitude faster. This is interesting, since on average he is only moving 25% faster (he takes, on average, five steps for every four taken by the others).

Let's construct a different plot, that may help us get more insight into the behavior of these three classes. Instead of plotting the change in distance over time for an increasing number of steps, the code in Figure 14.10 plots the distribution of final locations for a single number of steps.

```
def getFinalLocs(numSteps, numTrials, dClass):
    locs = []
    d = dClass()
    for t in range(numTrials):
        f = Field()
        f.addDrunk(d, Location(0, 0))
        for s in range(numSteps):
            f.moveDrunk(d)
        locs.append(f.getLoc(d))
    return locs

def plotLocs(drunkKinds, numSteps, numTrials):
    styleChoice = styleIterator(('k+', 'r^', 'mo'))
    for dClass in drunkKinds:
        locs = getFinalLocs(numSteps, numTrials, dClass)
        xVals, yVals = [], []
        for loc in locs:
            xVals.append(loc.getX())
            yVals.append(loc.getY())
        meanX = sum(xVals)/len(xVals)
        meanY = sum(yVals)/len(yVals)
        curStyle = styleChoice.nextStyle()
        pylab.plot(xVals, yVals, curStyle,
                   label = dClass.__name__ + ' mean loc. = <'
                   + str(meanX) + ', ' + str(meanY) + '>')
    pylab.title('Location at End of Walks ('
                + str(numSteps) + ' steps)')
    pylab.xlabel('Steps East/West of Origin')
    pylab.ylabel('Steps North/South of Origin')
    pylab.legend(loc = 'lower left')

plotLocs((UsualDrunk, ColdDrunk, EWDrunk), 100, 200)
```

Figure 14.10 Plotting final locations

The first thing `plotLocs` does is create an instance of `styleIterator` with three different styles of markers. It then uses `pylab.plot` to place a marker at a location corresponding to the end of

each trial. The call to `pylab.plot` sets the color and shape of the marker to be plotted using the values returned by the iterator `styleIterator`.

The call `plotLocs((UsualDrunk, ColdDrunk, EWDrunk), 100, 200)` produces the plot in Figure 14.11.

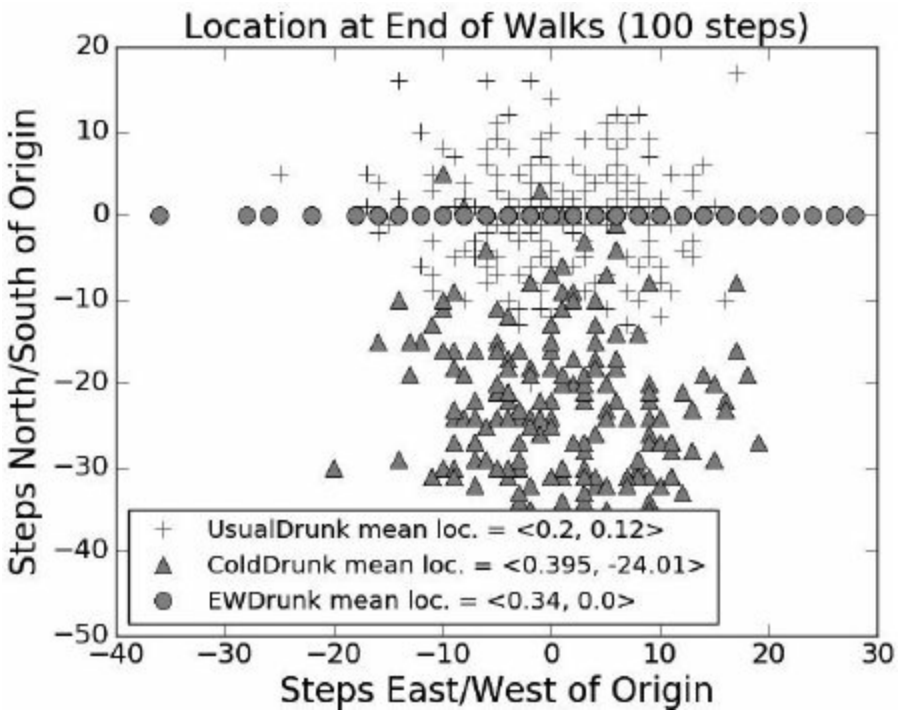

Figure 14.11 Where the drunk stops

The first thing to say is that our drunks seem to be behaving as advertised. The `EWDrunk` ends up on the x-axis, the `ColdDrunk` seem to have made progress southwards, and the `UsualDrunk` seem to have wandered aimlessly.

But why do there appear to be far fewer circle markers than triangle or + markers? Because many of the `EWDrunk`'s walks ended up at the same place. This is not surprising, given the small number of possible endpoints (200) for the `EWDrunk`. Also the circle markers seem to be fairly uniformly spaced across the x-axis.

It is still not immediately obvious, at least to us, why the `ColdDrunk` manages, on average, to get so much farther from the origin than the other kinds of drunks. Perhaps it's time to look not at the endpoints of many walks, but at the path followed by a single walk. The code in

Figure 14.12 produces the plot in Figure 14.13.

```
def traceWalk(drunkKinds, numSteps):
    styleChoice = styleIterator(('k+', 'r^', 'mo'))
    f = Field()
    for dClass in drunkKinds:
        d = dClass()
        f.addDrunk(d, Location(0, 0))
        locs = []
        for s in range(numSteps):
            f.moveDrunk(d)
            locs.append(f.getLoc(d))
        xVals, yVals = [], []
        for loc in locs:
            xVals.append(loc.getX())
            yVals.append(loc.getY())
        curStyle = styleChoice.nextStyle()
        pylab.plot(xVals, yVals, curStyle,
                   label = dClass.__name__)
    pylab.title('Spots Visited on Walk ('
                + str(numSteps) + ' steps)')
    pylab.xlabel('Steps East/West of Origin')
    pylab.ylabel('Steps North/South of Origin')
    pylab.legend(loc = 'best')

traceWalk((UsualDrunk, ColdDrunk, EWDrunk), 200)
```

Figure 14.12 Tracing walks

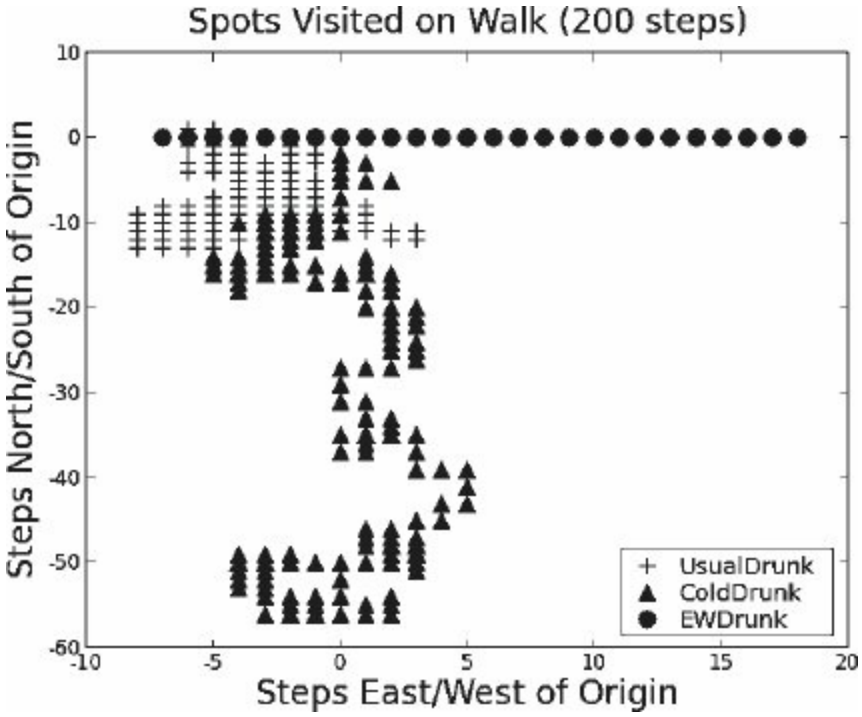

Figure 14.13 Trajectory of walks

Since the walk is 200 steps long and the `EWDrunk`'s walk visits fewer than 30 different locations, it's clear that he is spending a lot of time retracing his steps. The same kind of observation holds for the `UsualDrunk`. In contrast, while the `ColdDrunk` is not exactly making a beeline for Florida, he is managing to spend relatively less time visiting places he has already been.

None of these simulations is interesting in its own right. (In Chapter 16, we will look at more intrinsically interesting simulations.) But there are some points worth taking away:

- Initially we divided our simulation code into four separate chunks. Three of them were classes (`Location`, `Field`, and `Drunk`) corresponding to abstract data types that appeared in the informal description of the problem. The fourth chunk was a group of functions that used these classes to perform a simple simulation.
- We then elaborated `Drunk` into a hierarchy of classes so that we could observe different kinds of biased random walks. The code for `Location` and `Field` remained untouched, but the simulation code was changed to iterate through the different subclasses of `Drunk`. In

doing this, we took advantage of the fact that a class is itself an object, and therefore can be passed as an argument.

- Finally, we made a series of incremental changes to the simulation that did not involve any changes to the classes representing the abstract types. These changes mostly involved introducing plots designed to provide insight into the different walks. This is very typical of the way in which simulations are developed. One gets the basic simulation working first, and then starts adding features.

14.4 Treacherous Fields

Did you ever play the board game known as *Chutes and Ladders* in the U.S. and *Snakes and Ladders* in the UK? This children's game originated in India (perhaps in the 2nd century BCE), where it was called *Moksha-patamu*. Landing on a square representing virtue (e.g., generosity) sent a player up a ladder to a higher tier of life. Landing on a square representing evil (e.g., lust), sent a player back to a lower tier of life.

We can easily add this kind of feature to our random walks by creating a `Field` with wormholes,[94] as shown in Figure 14.14, and replacing the second line of code in the function `traceWalk` by the line of code

```
f = oddField(1000, 100, 200)
```

In an `oddField`, a drunk who steps into a wormhole location is transported to the location at the other end of the wormhole.

```
class oddField(Field):
    def __init__(self, numHoles, xRange, yRange):
        Field.__init__(self)
        self.wormholes = {}
        for w in range(numHoles):
            x = random.randint(-xRange, xRange)
            y = random.randint(-yRange, yRange)
            newX = random.randint(-xRange, xRange)
            newY = random.randint(-yRange, yRange)
            newLoc = Location(newX, newY)
            self.wormholes[(x, y)] = newLoc

    def moveDrunk(self, drunk):
        Field.moveDrunk(self, drunk)
        x = self.drunks[drunk].getX()
        y = self.drunks[drunk].getY()
        if (x, y) in self.wormholes:
            self.drunks[drunk] = self.wormholes[(x, y)]
```

Figure 14.14 Fields with strange properties

When we ran `traceWalk((UsualDrunk, ColdDrunk, EWDrunk), 500)`, we got the rather odd-looking plot in Figure 14.15.

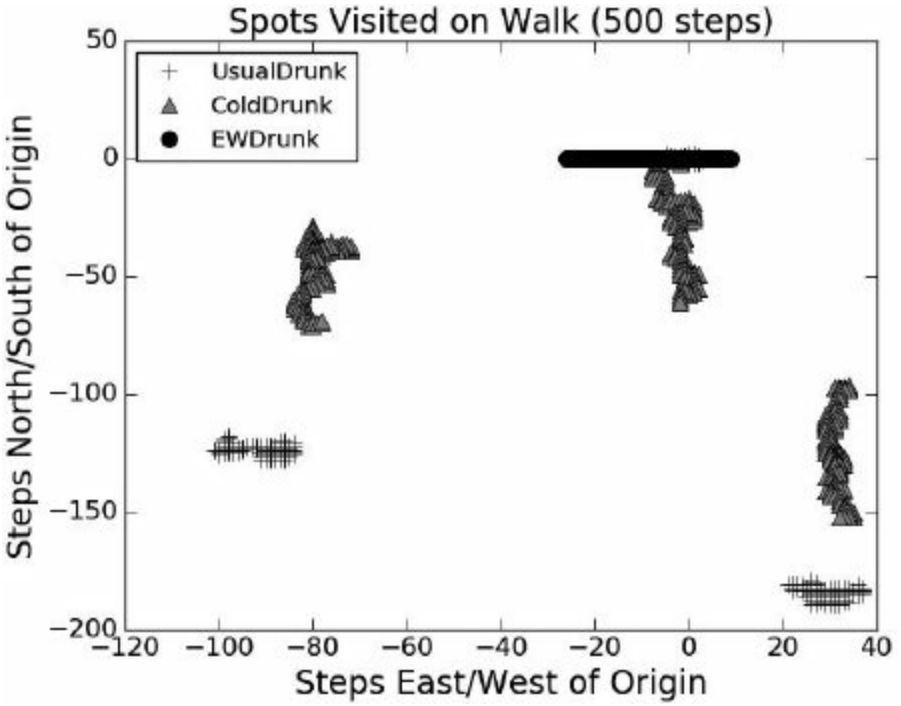

Figure 14.15 A strange walk

Clearly changing the properties of the field has had a dramatic effect. However, that is not the point of this example. The main points are:

- Because of the way we structured our code, it was easy to accommodate a significant change to the situation being modeled. Just as we could add different kinds of drunks without touching `Field`, we can add a new kind of `Field` without touching `Drunk` or any of its subclasses. (Had we been sufficiently prescient to make the field a parameter of `traceWalk`, we wouldn't have had to change `traceWalk` either.)
- While it would have been feasible to analytically derive different kinds of information about the expected behavior of the simple random walk and even the biased random walks, it would have been challenging to do so once the wormholes were introduced. Yet it was exceedingly simple to change the simulation to model the new situation. Simulation models often enjoy this advantage relative to analytic models.

[87] The word stems from the Greek word *stokhastikos*, which means something like "capable of divining." A stochastic program, as we shall see, is aimed at getting a good result, but the exact results are not guaranteed.

[88] Usually attributed to the statistician George E.P. Box.

[89] Nor was he the first to observe it. As early as 60 BCE, the Roman Titus Lucretius, in his poem "On the Nature of Things," described a similar phenomenon, and even implied that it was caused by the random movement of atoms.

[90] "On the movement of small particles suspended in a stationary liquid demanded by the molecular-kinetic theory of heat," *Annalen der Physik*, May 1905. Einstein would come to describe 1905 as his "*annus mirabilis*." That year, in addition to his paper on Brownian motion, he published papers on the production and transformation of light (pivotal to the development of quantum theory), on the electrodynamics of moving bodies (special relativity), and on the equivalence of matter and energy ($E = mc^2$). Not a bad year for a newly minted PhD.

[91] To be honest, the person pictured here is an actor impersonating a farmer.

[92] Why $\sqrt{2}$? We are using the Pythagorean theorem.

[93] In the 19th century, it became standard practice for plumbers to test closed systems of pipes for leaks by filling the system with smoke. Later, electronic engineers adopted the term to cover the very first test of a piece of electronics—turning on the power and looking for smoke. Still later, software developers starting using the term for a quick test to see if a program did anything useful.

[94] This kind of wormhole is a hypothetical concept invented by

theoretical physicists (or maybe science fiction writers). It provides shortcuts through the time/space continuum.

15 STOCHASTIC PROGRAMS, PROBABILITY, AND DISTRIBUTIONS

There is something very comforting about Newtonian mechanics. You push down on one end of a lever, and the other end goes up. You throw a ball up in the air; it travels a parabolic path, and eventually comes down. $\vec{F} = m\vec{a}$. In short, everything happens for a reason. The physical world is a completely predictable place—all future states of a physical system can be derived from knowledge about its current state.

For centuries, this was the prevailing scientific wisdom; then along came quantum mechanics and the Copenhagen Doctrine. The doctrine's proponents, led by Bohr and Heisenberg, argued that at its most fundamental level the behavior of the physical world cannot be predicted. One can make probabilistic statements of the form "x is highly likely to occur," but not statements of the form "x is certain to occur." Other distinguished physicists, most notably Einstein and Schrödinger, vehemently disagreed.

This debate roiled the worlds of physics, philosophy, and even religion. The heart of the debate was the validity of **causal nondeterminism**, i.e., the belief that not every event is caused by previous events. Einstein and Schrödinger found this view philosophically unacceptable, as exemplified by Einstein's often-repeated comment, "God does not play dice." What they could accept was **predictive nondeterminism**, i.e., the concept that our inability to make accurate measurements about the physical world makes it impossible to make precise predictions about future states. This distinction was nicely summed up by Einstein, who said, "The essentially statistical character of contemporary theory is solely to be ascribed to the fact that this theory operates with an incomplete description of physical systems."

The question of causal nondeterminism is still unsettled. However, whether the reason we cannot predict events is because they are truly unpredictable or is because we simply don't have enough information to predict them is of no practical importance.

While the Bohr/Einstein debate was about how to understand the lowest levels of the physical world, the same issues arise at the macroscopic level. Perhaps the outcomes of horse races, spins of roulette wheels, and stock market investments are causally deterministic. However, there is ample evidence that it is perilous to treat them as predictably deterministic.[95]

15.1 Stochastic Programs

A program is **deterministic** if whenever it is run on the same input, it produces the same output. Notice that this is not the same as saying that the output is completely defined by the specification of the problem. Consider, for example, the specification of `squareRoot`:

```
def squareRoot(x, epsilon):
    """Assumes x and epsilon are of type float; x >= 0 and epsilon > 0
       Returns float y such that x-epsilon <= y*y <= x+epsilon"""
```

This specification admits many possible return values for the function call `squareRoot(2, 0.001)`. However, the successive approximation algorithm that we looked at in Chapter 3 will always return the same value. The specification doesn't require that the implementation be deterministic, but it does allow deterministic implementations.

Not all interesting specifications can be met by deterministic implementations. Consider, for example, implementing a program to play a dice game, say backgammon or craps. Somewhere in the program there may be a function that simulates a fair roll of a single six-sided die.[96] Suppose it had a specification something like

```
def rollDie():
    """Returns an int between 1 and 6"""
```

This would be problematic, since it allows the implementation to return the same number each time it is called, which would make for a pretty boring game. It would be better to specify that `rollDie` "`returns a randomly chosen int between 1 and 6`," thus requiring a stochastic implementation.

Most programming languages, including Python, include simple ways to write stochastic programs, i.e., programs that exploit randomness. The tiny program in Figure 15.1 is a simulation model. Rather than asking some person to roll a die multiple times, we wrote a program to simulate that activity. The code uses one of several useful functions found in the imported Python standard library module `random`. As we saw earlier, the function `random.choice` takes a nonempty sequence as its argument and returns a randomly chosen member of that sequence. Almost all of the functions in `random` are built using the function `random.random`, which, as we saw earlier in the book, generates a random floating point number between `0.0` and `1.0`.[97]

```
import random

def rollDie():
    """Returns a random int between 1 and 6"""
    return random.choice([1,2,3,4,5,6])

def rollN(n):
    result = ''
    for i in range(n):
        result = result + str(rollDie())
    print(result)
```

Figure 15.1 Roll die

Now, imagine running `rollN(10)`. Would you be more surprised to see it print `1111111111` or `5442462412`? Or, to put it another way, which of these two sequences is more random? It's a trick question. Each of these sequences is equally likely, because the value of each roll is independent of the values of earlier rolls. In a stochastic process, two events are **independent** if the outcome of one event has no influence on the outcome of the other.

This is a bit easier to see if we simplify the situation by thinking about a two-sided die (also known as a coin) with the values 0 and 1. This allows us to think of the output of a call of `rollN` as a binary number. When we use a binary die, there are 2^n possible sequences that `testN` might return. Each of these sequences is equally likely; therefore each has a probability of occurring of $(1/2)^n$.

Let's go back to our six-sided die. How many different sequences are there of length 10? 6^{10}. So, the probability of rolling ten consecutive 1's is $1/6^{10}$. Less than one out of sixty million. Pretty low, but no lower than the probability of any other sequence, e.g., `5442462412`.

15.2 Calculating Simple Probabilities

In general, when we talk about the **probability** of a result having some property (e.g., all 1's) we are asking what fraction of all possible results has that property. This is why probabilities range from 0 to 1. Suppose we want to know the probability of getting any sequence other than all 1's when rolling the die. It is simply $1 - (1/6^{10})$, because the probability of something happening and the probability of the same thing not happening must add up to 1.

Suppose we want to know the probability of rolling the die ten times without getting a single 1. One way to answer this question is to transform it into the question of how many of the 6^{10} possible sequences don't contain a 1. This can be computed as follows:

1. The probability of not rolling a 1 on any single roll is $5/6$.
2. The probability of not rolling a 1 on either the first or the second roll is $(5/6)*(5/6)$, or $(5/6)^2$.
3. So, the probability of not rolling a 1 ten times in a row is $(5/6)^{10}$, slightly more than 0.16.

Step 2 is an application of the **multiplicative law** for independent probabilities. Consider, for example, two independent events A and B. If A occurs one $1/3$ of the time and B occurs $1/4$ of the time, the probability that both A and B occur is $1/4$ of $1/3$, i.e., $(1/3)/4$ or $(1/3)*(1/4)$.

What about the probability of rolling at least one 1? It is simply 1 minus the probability of not rolling at least one 1, i.e., $1 - (5/6)^{10}$. Notice that this cannot be correctly computed by saying that the probability of rolling a 1 on any roll is $1/6$, so the probability of rolling at least one 1 is $10*(1/6)$, i.e., $10/6$. This is obviously incorrect, since a probability cannot be greater than 1.

How about the probability of rolling exactly two 1's in ten rolls? This is equivalent to asking what fraction of the first 6^{10} integers has exactly two 1's in its base 6 representation. We could easily write a program to generate all of these sequences and count the number that contained exactly one 1. Deriving the probability analytically is a bit tricky, and we defer it to Section 15.4.4.

15.3 Inferential Statistics

As we just saw, one can use a systematic process to derive the precise probability of some complex event based upon knowing the probability of one or more simpler events. For example, one can easily compute the probability of flipping a coin and getting ten consecutive heads based on the assumption that flips are independent and we know the probability of each flip coming up heads. Suppose, however, that we don't actually know the probability of the relevant simpler event. Suppose, for example, that we don't know whether the coin is fair (i.e., a coin where heads and tails are equally likely).

All is not lost. If we have some data about the behavior of the coin, we can combine that data with our knowledge of probability to derive an estimate of the true probability. We can use **inferential statistics** to estimate the probability of a single flip coming up heads, and then conventional probability to compute the probability of a coin with that behavior coming up heads ten times in a row.

In brief (since this is not a book about statistics), the guiding principle of inferential statistics is that a random sample tends to exhibit the same properties as the population from which it is drawn.

Suppose Harvey Dent (also known as Two-Face) flipped a coin, and it came up heads. You would not infer from this that the next flip would also come up heads. Suppose he flipped it twice, and it came up heads both time. You might reason that the probability of this happening for a fair coin was 0.25, so there was still no reason to assume the next flip would be heads. Suppose, however, 100 out of 100 flips came up heads. $(1/2)^{100}$ (the probability of this event, assuming a fair coin) is a pretty small number, so you might feel safe in inferring that the coin has a head on both sides.

Your belief in whether the coin is fair is based on the intuition that the behavior of a single sample of 100 flips is similar to the behavior of the population of all samples of 100 flips. This belief seems pretty sound when all 100 flips are heads. Suppose that 52 flips came up heads and 48 tails. Would you feel comfortable in predicting that the next 100 flips would have the same ratio of heads to tails? For that matter, how comfortable would you feel about even predicting that there would be more heads than tails in the next 100 flips? Take a few minutes to think about this, and then try the experiment. Or, if you don't happen to have a coin handy, simulate the flips using the code in Figure 15.2.

The function `flip` in Figure 15.2 simulates flipping a fair coin `numFlips` times, and returns the fraction of those flips that came up heads. For each flip, the call `random.choice(('H', 'T'))` randomly returns either `'H'` or `'T'`.

```
def flip(numFlips):
    """Assumes numFlips a positive int"""
    heads = 0
    for i in range(numFlips):
        if random.choice(('H', 'T')) == 'H':
            heads += 1
    return heads/numFlips

def flipSim(numFlipsPerTrial, numTrials):
    """Assumes numFlipsPerTrial and numTrials positive ints"""
    fracHeads = []
    for i in range(numTrials):
        fracHeads.append(flip(numFlipsPerTrial))
    mean = sum(fracHeads)/len(fracHeads)
    return mean
```

Figure 15.2 Flipping a coin

Try executing the function `flipSim(10, 1)` a couple of times. Here's what we saw the first two times we tried `print('Mean =', flipSim(10, 1))`:

```
Mean = 0.2
Mean = 0.6
```

It seems that it would be inappropriate to assume much (other than that the coin has both heads and tails) from any one trial of 10 flips. That's why we typically structure our simulations to include multiple trials and compare the results. Let's try `flipSim(10, 100) a couple of times`:

```
Mean = 0.5029999999999999
Mean = 0.496
```

Do you feel better about these results? When we tried `flipSim(100, 100000)`, we got

```
Mean = 0.5005000000000038
Mean = 0.5003139999999954
```

This looks really good (especially since we know that the answer should be 0.5—but that's cheating). Now it seems we can safely conclude something about the next flip, i.e., that heads and tails are about equally likely. But why do we think that we can conclude that?

What we are depending upon is the **law of large numbers** (also known as **Bernoulli's theorem**[98]). This law states that in repeated independent tests (flips in this case) with the same actual probability p of a particular outcome in each test (e.g., an actual probability of 0.5 of getting a head for each flip), the chance that the fraction of times that outcome occurs differs from p converges to zero as the number of trials goes to infinity.

It is worth noting that the law of large numbers does not imply, as too many seem to think, that if deviations from expected behavior occur, these deviations are likely to be "evened out" by opposite deviations in the future. This misapplication of the law of large numbers is known as the **gambler's fallacy**.[99]

People often confuse the gambler's fallacy with regression to the mean. **Regression to the mean**[100] states that following an extreme random event, the next random event is likely to be less extreme. If you were to flip a fair coin six times and get six heads, regression to the mean implies that the next sequence of six flips is likely to have closer to the expected value of three heads. It does not imply, as the gambler's fallacy suggests, that the next sequence of flips is likely to have fewer heads than tails.

Success in most endeavors requires a combination of skill and luck. The skill component determines the mean and the luck component accounts for the variability. The randomness of luck leads to regression to the mean.

The code in Figure 15.3 produces a plot, Figure 15.4, illustrating regression to the mean. The function `regressToMean` first generates `numTrials` trials of `numFlips` coin flips each. It then identifies all trials where the fraction of heads was either less than `1/3` or more than `2/3` and plots these extremal values as circles. Then, for each of these points, it plots the value of the subsequent trial as a triangle in the same column as the circle.

The horizontal line at 0.5, the expected mean, is created using the `axhline` function. The function `pylab.xlim` controls the extent of the x-axis. The function call `pylab.xlim(xmin, xmax)` sets the minimum and maximum values of the x-axis of the current figure. The function call `pylab.xlim()` returns a tuple composed of the minimum and maximum values of the x-axis of the current figure. The function `pylab.ylim` works the same way.

```
def regressToMean(numFlips, numTrials):
    #Get fraction of heads for each trial of numFlips
    fracHeads = []
    for t in range(numTrials):
        fracHeads.append(flip(numFlips))
    #Find trials with extreme results and for each the next trial
    extremes, nextTrials = [], []
    for i in range(len(fracHeads) - 1):
        if fracHeads[i] < 0.33 or fracHeads[i] > 0.66:
            extremes.append(fracHeads[i])
            nextTrials.append(fracHeads[i+1])
    #Plot results
    pylab.plot(range(len(extremes)), extremes, 'ko',
               label = 'Extreme')
    pylab.plot(range(len(nextTrials)), nextTrials, 'k^',
               label = 'Next Trial')
    pylab.axhline(0.5)
    pylab.ylim(0, 1)
    pylab.xlim(-1, len(extremes) + 1)
    pylab.xlabel('Extreme Example and Next Trial')
    pylab.ylabel('Fraction Heads')
    pylab.title('Regression to the Mean')
    pylab.legend(loc = 'best')

regressToMean(15, 40)
```

Figure 15.3: Regression to the mean

Notice that while the trial following an extreme result is typically followed by a trial closer to the mean than the extreme result, that doesn't always occur—as shown by the boxed pair.

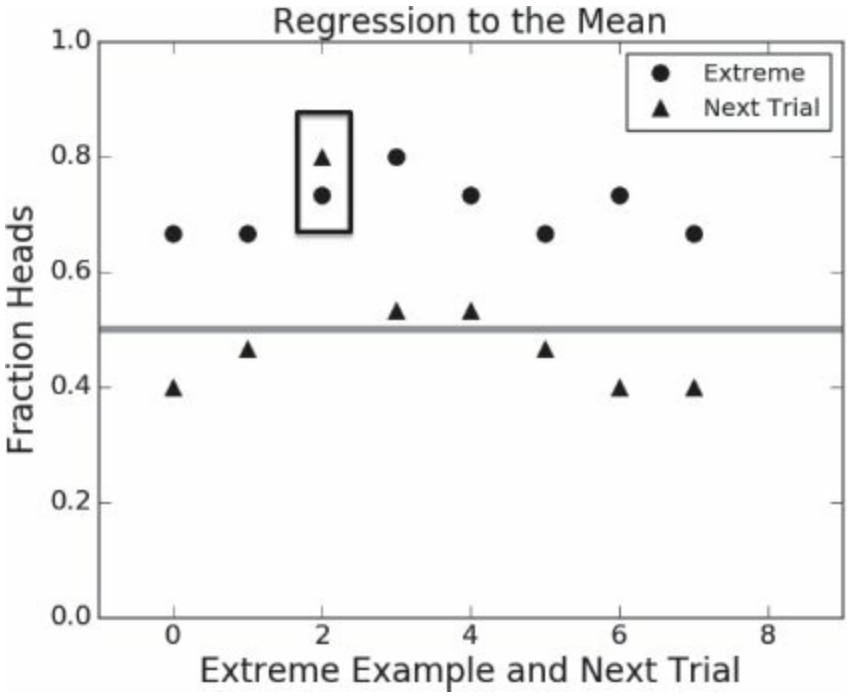

Figure 15.4: Illustration of regression to mean

Finger exercise: Sally averages 5 strokes a hole when she plays golf. One day, she took 40 strokes to complete the first nine holes. Her partner conjectured that she would probably regress to the mean and take 50 strokes to complete the next nine holes. Do you agree with her partner?

Figure 15.5 contains a function, `flipPlot`, that produces two plots, Figure 15.6, intended to show the law of large numbers at work. The first plot shows how the absolute value of the difference between the number of heads and number of tails changes with the number of flips. The second plot compares the ratio of heads to tails versus the number of flips. The line `random.seed(0)` near the bottom ensures that the pseudorandom number generator used by `random.random` will generate the same sequence of pseudorandom numbers each time this code is executed.[101] This is convenient for debugging. The function `random.seed` can be called with any number. If it is called with no argument, the seed is chosen at random.

```
def flipPlot(minExp, maxExp):
    """Assumes minExp and maxExp positive integers; minExp < maxExp
       Plots results of 2**minExp to 2**maxExp coin flips"""
    ratios, diffs, xAxis = [], [], []
    for exp in range(minExp, maxExp + 1):
        xAxis.append(2**exp)
    for numFlips in xAxis:
        numHeads = 0
        for n in range(numFlips):
            if random.choice(('H', 'T')) == 'H':
                numHeads += 1
        numTails = numFlips - numHeads
        try:
            ratios.append(numHeads/numTails)
            diffs.append(abs(numHeads - numTails))
        except ZeroDivisionError:
            continue
    pylab.title('Difference Between Heads and Tails')
    pylab.xlabel('Number of Flips')
    pylab.ylabel('Abs(#Heads - #Tails)')
    pylab.plot(xAxis, diffs, 'k')
    pylab.figure()
    pylab.title('Heads/Tails Ratios')
    pylab.xlabel('Number of Flips')
    pylab.ylabel('#Heads/#Tails')
    pylab.plot(xAxis, ratios, 'k')

random.seed(0)
flipPlot(4, 20)
```

Figure 15.5 Plotting the results of coin flips

The plot on the left seems to suggest that the absolute difference between the number of heads and the number of tails fluctuates in the beginning, crashes downwards, and then moves rapidly upwards. However, we need to keep in mind that we have only two data points to the right of x = 300,000. The fact that `pylab.plot` connected these points with lines may mislead us into seeing trends when all we have are isolated points. This is not an uncommon phenomenon, so you should always ask how many points a plot actually contains before jumping to any conclusion about what it means.

It's hard to see much of anything in the plot on the right, which is mostly a flat line. This too is deceptive. Even though there are sixteen data points, most of them are crowded into a small amount of real estate on the left side of the plot, so that the detail is impossible to see. This occurs because the

plotted points have x values of 2^4, 2^5, 2^6, ..., 2^{20}, so the values on the x-axis range from 16 to over a million, and unless instructed otherwise PyLab will place these points based on their relative distance from the origin. This is called **linear scaling**. Because most of the points have x values that are small relative to 2^{20}, they will appear relatively close to the origin.

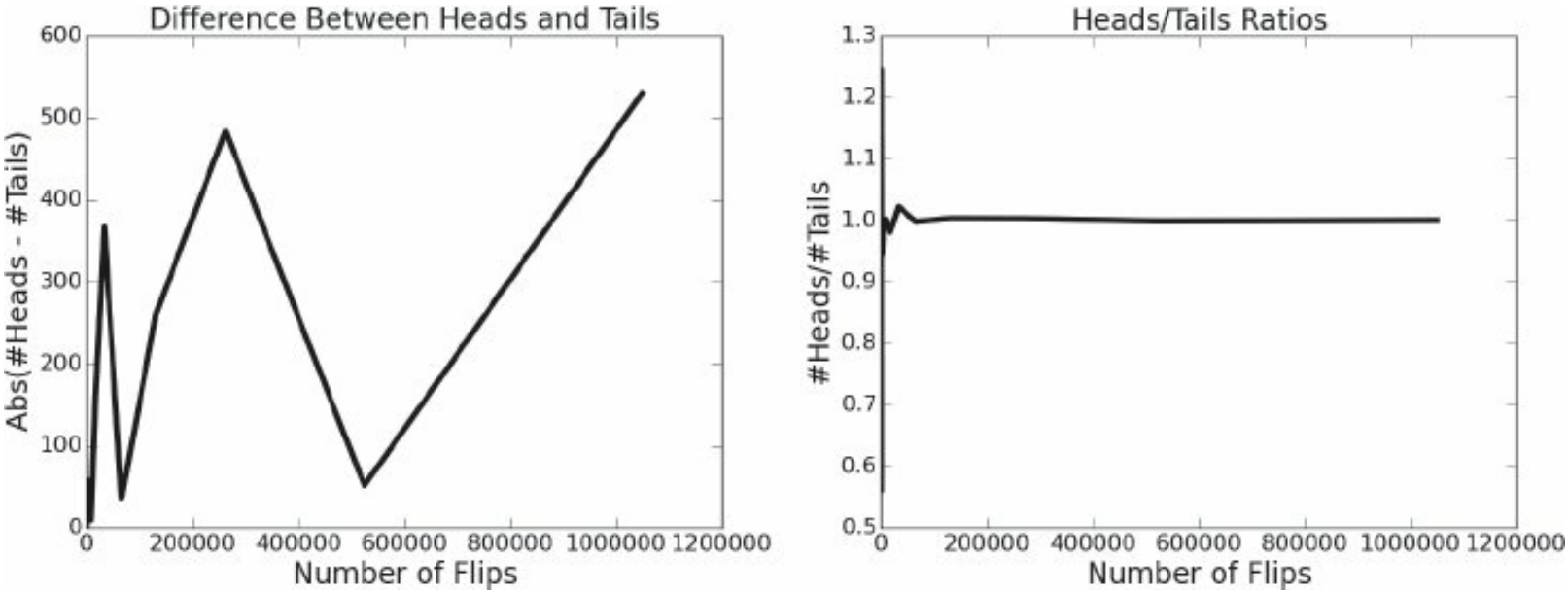

Figure 15.6 The law of large numbers at work

Fortunately, these visualization problems are easy to address in PyLab. As we saw in Chapter 11 and earlier in this chapter, we can easily instruct our program to plot unconnected points, e.g., by writing `pylab.plot(xAxis, diffs, 'ko')`.

Both plots in Figure 15.7 use a logarithmic scale on the x-axis. Since the x values generated by `flipPlot` are 2^{minExp}, $2^{minExp+1}$, .., 2^{maxExp}, using a logarithmic x-axis causes the points to be evenly spaced along the x-axis—providing maximum separation between points. The left-hand plot in Figure 15.7 uses a logarithmic scale on the y-axis as well as on the x-axis. The y values on this plot range from nearly 0 to around 550. If the y-axis were linearly scaled, it would be difficult to see the relatively small differences in y values on the left side of the plot. On the other hand, on the plot on the right the y values are fairly tightly grouped, so we use a linear y-axis.

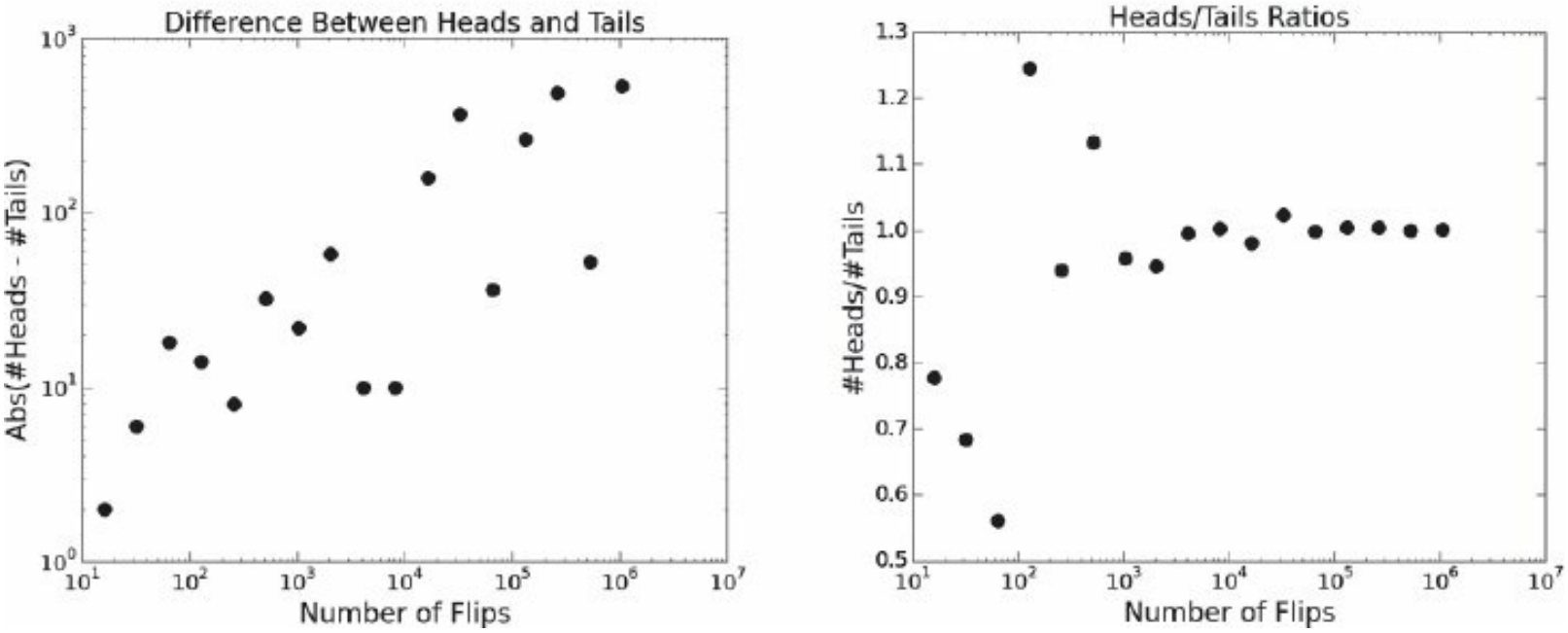

Figure 15.7 Impact of number of flips

Finger exercise: Modify the code in Figure 15.5 so that it produces plots like those shown in Figure 15.7.

These plots are easier to interpret than the earlier plots. The plot on the right suggests pretty strongly that the ratio of heads to tails converges to 1.0 as the number of flips gets large. The meaning of the plot on the left is a bit less clear. It appears that the absolute difference grows with the number of flips, but it is not completely convincing.

It is never possible to achieve perfect accuracy through sampling without sampling the entire population. No matter how many samples we examine, we can never be sure that the sample set is typical until we examine every element of the population (and since we are often dealing with infinite populations, e.g., all possible sequences of coin flips, this is often impossible). Of course, this is not to say that an estimate cannot be precisely correct. We might flip a coin twice, get one heads and one tails, and conclude that the true probability of each is 0.5. We would have reached the right conclusion, but our reasoning would have been faulty.

How many samples do we need to look at before we can have justified confidence in our answer? This depends on the **variance** in the underlying distribution. Roughly speaking, variance is a measure of how much spread there is in the possible different outcomes. More formally, the variance of a collection of values, *X*, is defined as

$$variance(X) = \frac{\sum_{x \epsilon X}(x - \mu)^2}{|X|}$$

where |*X*| is the size of the collection and μ (mu) its mean. Informally, the variance describes what fraction of the values are close to the mean. If many values are relatively close to the mean, the variance is relatively small. If many values are relatively far from the mean, the variance is relatively

large. If all values are the same, the variance is zero.

The **standard deviation** of a collection of values is the square root of the variance. While it contains exactly the same information as the variance, the standard deviation is easier to interpret because it is in the same units as the original data. For example, is easier to understand the statement "the mean height of a population is 70 inches with a standard deviation of 4 inches," than the sentence "the mean of height of a population is 70 inches with a variance of 16 inches^2."

Figure 15.8 contains implementations of variance and standard deviation.[102]

```
def variance(X):
    """Assumes that X is a list of numbers.
       Returns the standard deviation of X"""
    mean = sum(X)/len(X)
    tot = 0.0
    for x in X:
        tot += (x - mean)**2
    return tot/len(X)

def stdDev(X):
    """Assumes that X is a list of numbers.
       Returns the standard deviation of X"""
    return variance(X)**0.5
```

Figure 15.8 Variance and standard deviation

We can use the notion of standard deviation to think about the relationship between the number of samples we have looked at and how much confidence we should have in the answer we have computed. Figure 15.9 contains a modified version of `flipPlot`. It uses the helper functions defined at the top of the figure to run multiple trials of each number of coin flips, and then plots the means for `abs(heads - tails)` and the `heads/tails` ratio. It also plots the standard deviation of each. The helper function `makePlot` contains the code used to produce the plots. The function `runTrial` simulates one trial of `numFlips` coins.

```
def makePlot(xVals, yVals, title, xLabel, yLabel, style,
             logX = False, logY = False):
    pylab.figure()
    pylab.title(title)
    pylab.xlabel(xLabel)
    pylab.ylabel(yLabel)
    pylab.plot(xVals, yVals, style)
    if logX:
        pylab.semilogx()
    if logY:
        pylab.semilogy()
```

```
def runTrial(numFlips):
    numHeads = 0
    for n in range(numFlips):
        if random.choice(('H', 'T')) == 'H':
            numHeads += 1
    numTails = numFlips - numHeads
    return (numHeads, numTails)

def flipPlot1(minExp, maxExp, numTrials):
    """Assumes minExp, maxExp, numTrials ints >0; minExp < maxExp
       Plots summaries of results of numTrials trials of
       2**minExp to 2**maxExp coin flips"""
    ratiosMeans, diffsMeans, ratiosSDs, diffsSDs = [], [], [], []
    xAxis = []
    for exp in range(minExp, maxExp + 1):
        xAxis.append(2**exp)
    for numFlips in xAxis:
        ratios, diffs = [], []
        for t in range(numTrials):
            numHeads, numTails = runTrial(numFlips)
            ratios.append(numHeads/numTails)
            diffs.append(abs(numHeads - numTails))
        ratiosMeans.append(sum(ratios)/numTrials)
        diffsMeans.append(sum(diffs)/numTrials)
        ratiosSDs.append(stdDev(ratios))
        diffsSDs.append(stdDev(diffs))
    numTrialsString = ' (' + str(numTrials) + ' Trials)'
    title = 'Mean Heads/Tails Ratios' + numTrialsString
    makePlot(xAxis, ratiosMeans, title, 'Number of flips',
             'Mean Heads/Tails', 'ko', logX = True)
    title = 'SD Heads/Tails Ratios' + numTrialsString
    makePlot(xAxis, ratiosSDs, title, 'Number of Flips',
            'Standard Deviation', 'ko', logX = True, logY = True)
```

Figure 15.9 Coin-flipping simulation

Let's try `flipPlot1(4, 20, 20)`. It generates the plots in Figure 15.10.

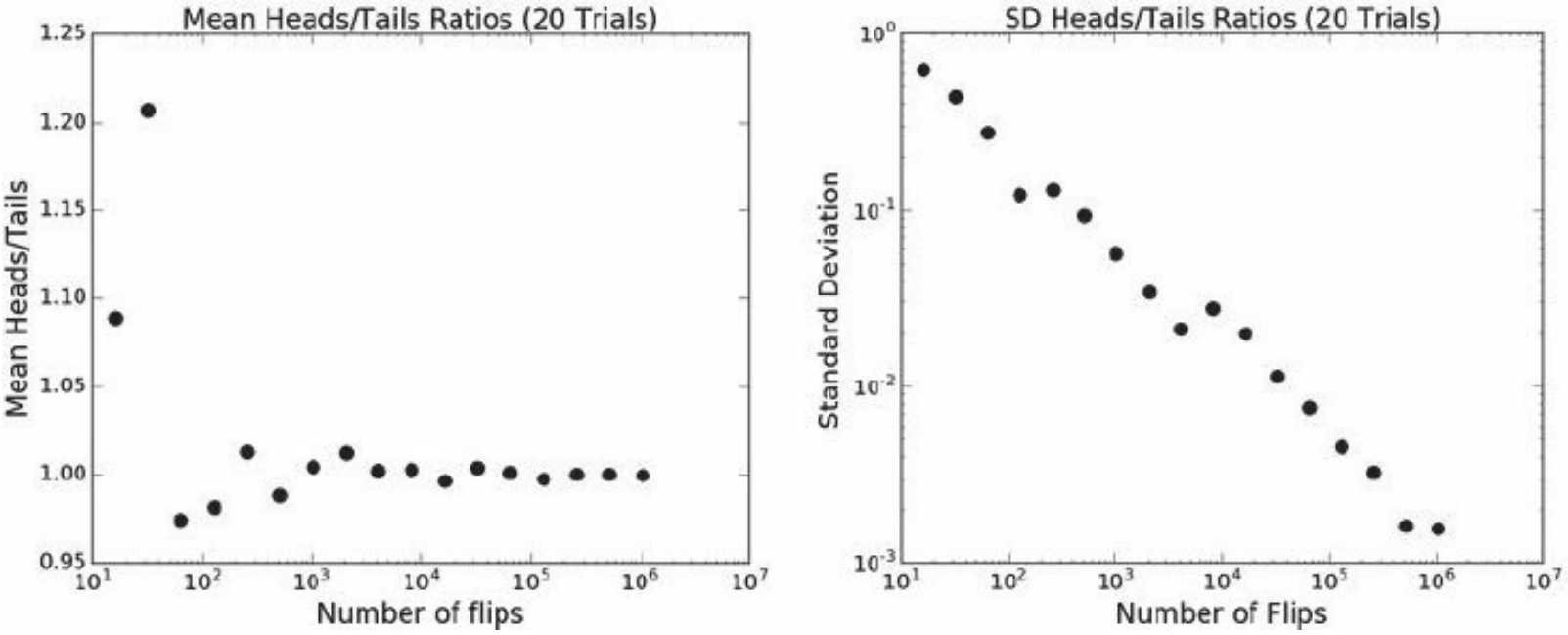

Figure 15.10 Convergence of heads/tails ratios

This is encouraging. The mean heads/tails ratio is converging towards 1 and the log of the standard deviation is falling linearly with the log of the number of flips per trial. By the time we get to about 10^6 coin flips per trial, the standard deviation (about 10^{-3}) is roughly three decimal orders of magnitude smaller than the mean (about 1), indicating that the variance across the trials was small. We can, therefore, have considerable confidence that the expected heads/tails ratio is quite close to 1.0. As we flip more coins, not only do we have a more precise answer, but more important, we also have reason to be more confident that it is close to the right answer.

What about the absolute difference between the number of heads and the number of tails? We can take a look at that by adding to the end of `flipPlot1` the code in Figure 15.11. This produces the plots in Figure 15.12.

```
title = 'Mean abs(#Heads - #Tails)' + numTrialsString
makePlot(xAxis, diffsMeans, title,
     'Number of Flips', 'Mean abs(#Heads - #Tails)', 'ko',
     logX = True, logY = True)
title = 'SD abs(#Heads - #Tails)' + numTrialsString
makePlot(xAxis, diffsSDs, title,
     'Number of Flips', 'Standard Deviation', 'ko',
     logX = True, logY = True)
```

Figure 15.11 Absolute differences

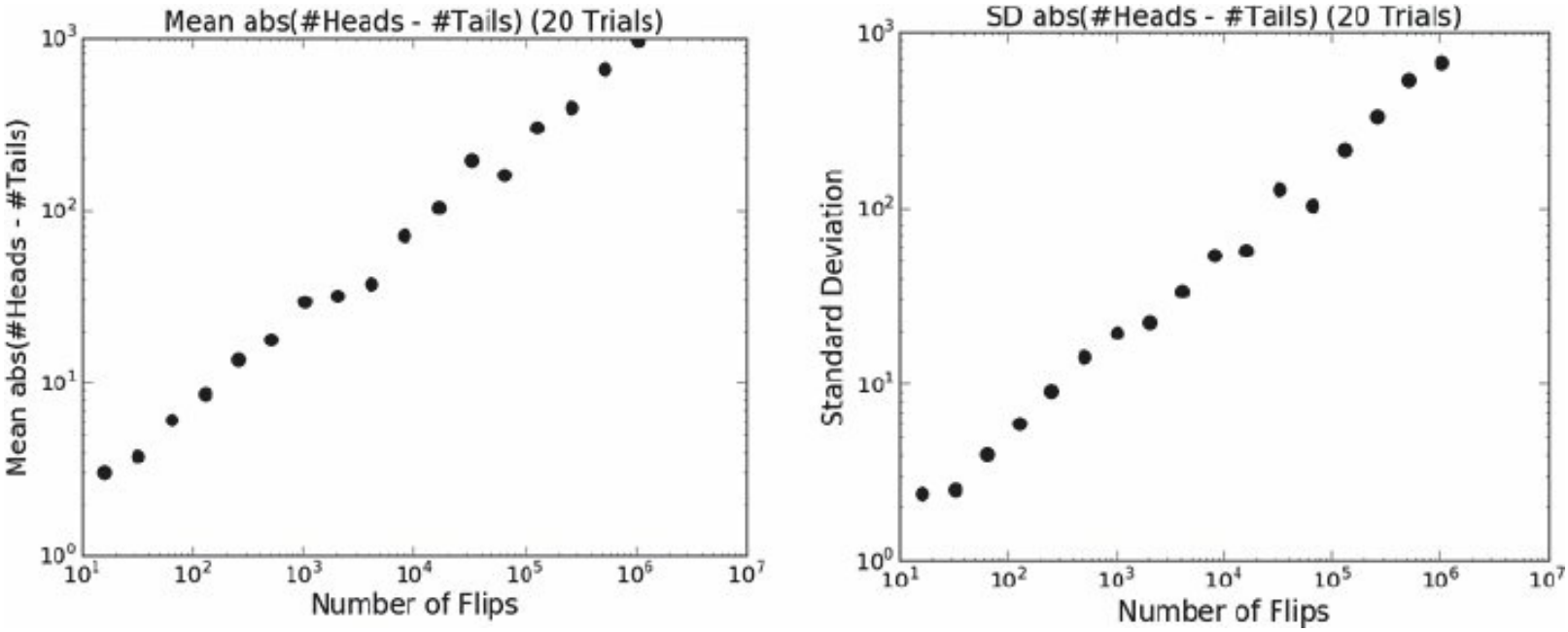

Figure 15.12 Mean and standard deviation of heads - tails

As expected, the absolute difference between the numbers of heads and tails grows with the number of flips. Furthermore, since we are averaging the results over twenty trials, the plot is considerably smoother than when we plotted the results of a single trial in Figure 15.7. But what's up with the plot on the right of Figure 15.12? The standard deviation is growing with the number of flips. Does this mean that as the number of flips increases we should have less rather than more confidence in the estimate of the expected value of the difference between heads and tails?

No, it does not. The standard deviation should always be viewed in the context of the mean. If the mean were a billion and the standard deviation 100, we would view the dispersion of the data as small. But if the mean were 100 and the standard deviation 100, we would view the dispersion as large.

The **coefficient of variation** is the standard deviation divided by the mean. When comparing data sets with different means (as here), the coefficient of variation is often more informative than the standard deviation. As you can see from its implementation in Figure 15.13, the coefficient of variation is not defined when the mean is 0.

```
def CV(X):
    mean = sum(X)/len(X)
    try:
        return stdDev(X)/mean
    except ZeroDivisionError:
        return float('nan')
```

Figure 15.13 Coefficient of variation

Figure 15.14 contains a function that plots coefficients of variation. In addition to the plots produced by `flipPlot1`, it produces the plots in Figure 15.15.

```
def flipPlot2(minExp, maxExp, numTrials):
    """Assumes minExp and maxExp positive ints; minExp < maxExp
         numTrials a positive integer
       Plots summaries of results of numTrials trials of
         2**minExp to 2**maxExp coin flips"""
    ratiosMeans, diffsMeans, ratiosSDs, diffsSDs = [], [], [], []
    ratiosCVs, diffsCVs, xAxis = [], [], []
    for exp in range(minExp, maxExp + 1):
        xAxis.append(2**exp)
    for numFlips in xAxis:
        ratios, diffs = [], []
        for t in range(numTrials):
            numHeads, numTails = runTrial(numFlips)
            ratios.append(numHeads/float(numTails))
            diffs.append(abs(numHeads - numTails))
        ratiosMeans.append(sum(ratios)/numTrials)
        diffsMeans.append(sum(diffs)/numTrials)
        ratiosSDs.append(stdDev(ratios))
        diffsSDs.append(stdDev(diffs))
        ratiosCVs.append(CV(ratios))
        diffsCVs.append(CV(diffs))
    numTrialsString = ' (' + str(numTrials) + ' Trials)'
    title = 'Mean Heads/Tails Ratios' + numTrialsString
    makePlot(xAxis, ratiosMeans, title, 'Number of flips',
             'Mean Heads/Tails', 'ko', logX = True)
    title = 'SD Heads/Tails Ratios' + numTrialsString
    makePlot(xAxis, ratiosSDs, title, 'Number of flips',
             'Standard Deviation', 'ko', logX = True, logY = True)
    title = 'Mean abs(#Heads - #Tails)' + numTrialsString
    makePlot(xAxis, diffsMeans, title,'Number of Flips',
             'Mean abs(#Heads - #Tails)', 'ko',
             logX = True, logY = True)
    title = 'SD abs(#Heads - #Tails)' + numTrialsString
    makePlot(xAxis, diffsSDs, title, 'Number of Flips',
             'Standard Deviation', 'ko', logX = True, logY = True)
    title = 'Coeff. of Var. abs(#Heads - #Tails)' + numTrialsString
    makePlot(xAxis, diffsCVs, title, 'Number of Flips',
             'Coeff. of Var.', 'ko', logX = True)
    title = 'Coeff. of Var. Heads/Tails Ratio' + numTrialsString
    makePlot(xAxis, ratiosCVs, title, 'Number of Flips',
             'Coeff. of Var.', 'ko', logX = True, logY = True)
```

Figure 15.14 Final version of `flipPlot1`

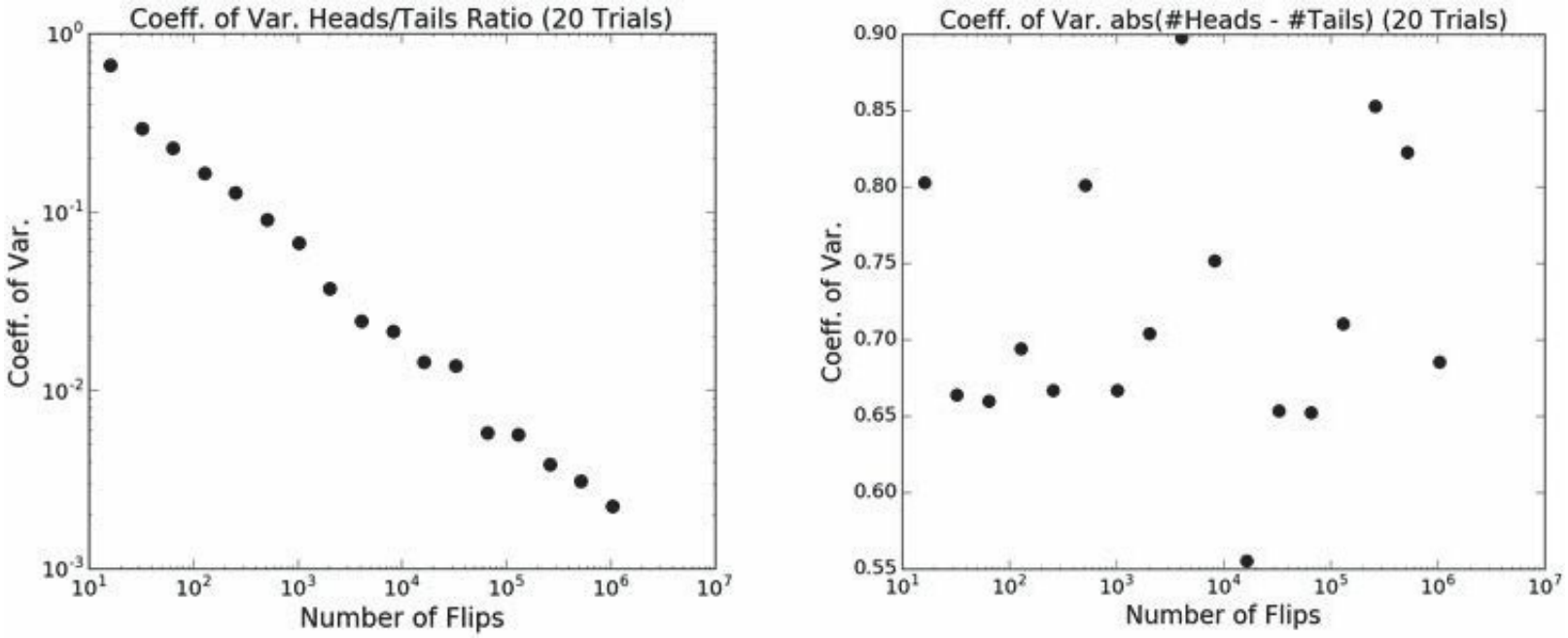

Figure 15.15 Coefficient of variation of heads/tails and abs(heads – tails)

In this case we see that the plot of coefficient of variation for the heads/tails ratio is not much different from the plot of the standard deviation in Figure 15.10. This is not surprising, since the only difference between the two is the division by the mean, and since the mean is close to 1 that makes little difference.

On the other hand, the plot of the coefficient of variation for the absolute difference between heads and tails is a different story. While the standard deviation exhibited a clear trend in Figure 15.12, it would take a brave person to argue that the coefficient of variation is trending in any direction. It seems to be fluctuating wildly. This suggests that dispersion in the values of `abs(heads - tails)` is independent of the number of flips. It's not growing, as the standard deviation might have misled us to believe, but it's not shrinking either. Perhaps a trend would appear if we tried 1000 trials instead of 20. Let's see.

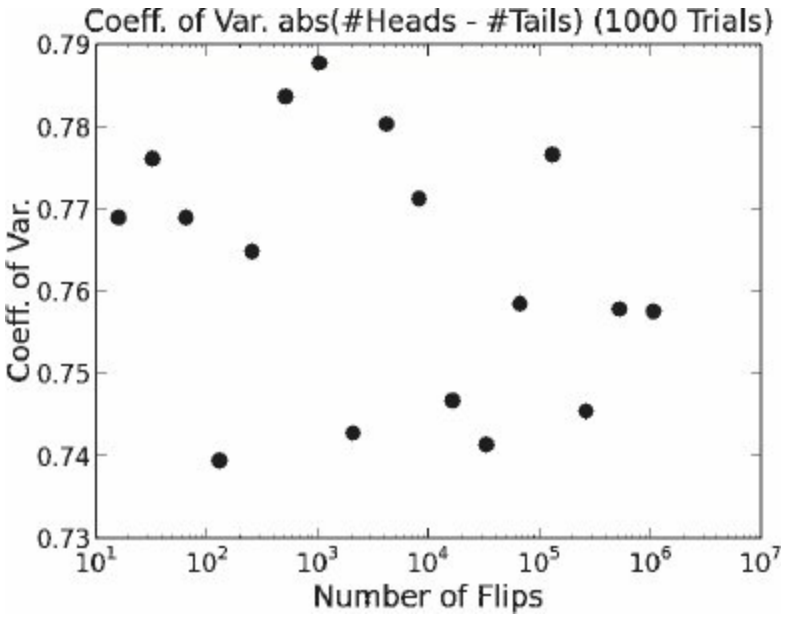

Figure 15.16 A large number of trials

In Figure 15.16, it looks as if the coefficient of variation settles in somewhere in the neighborhood of 0.74-0.78. In general, distributions with a coefficient of variation of less than 1 are considered low-variance.

The main advantage of the coefficient of variation over the standard deviation is that it allows us to compare the dispersion of sets with different means. Consider, for example, the distribution of weekly income in different regions of Australia, as depicted in Figure 15.17.

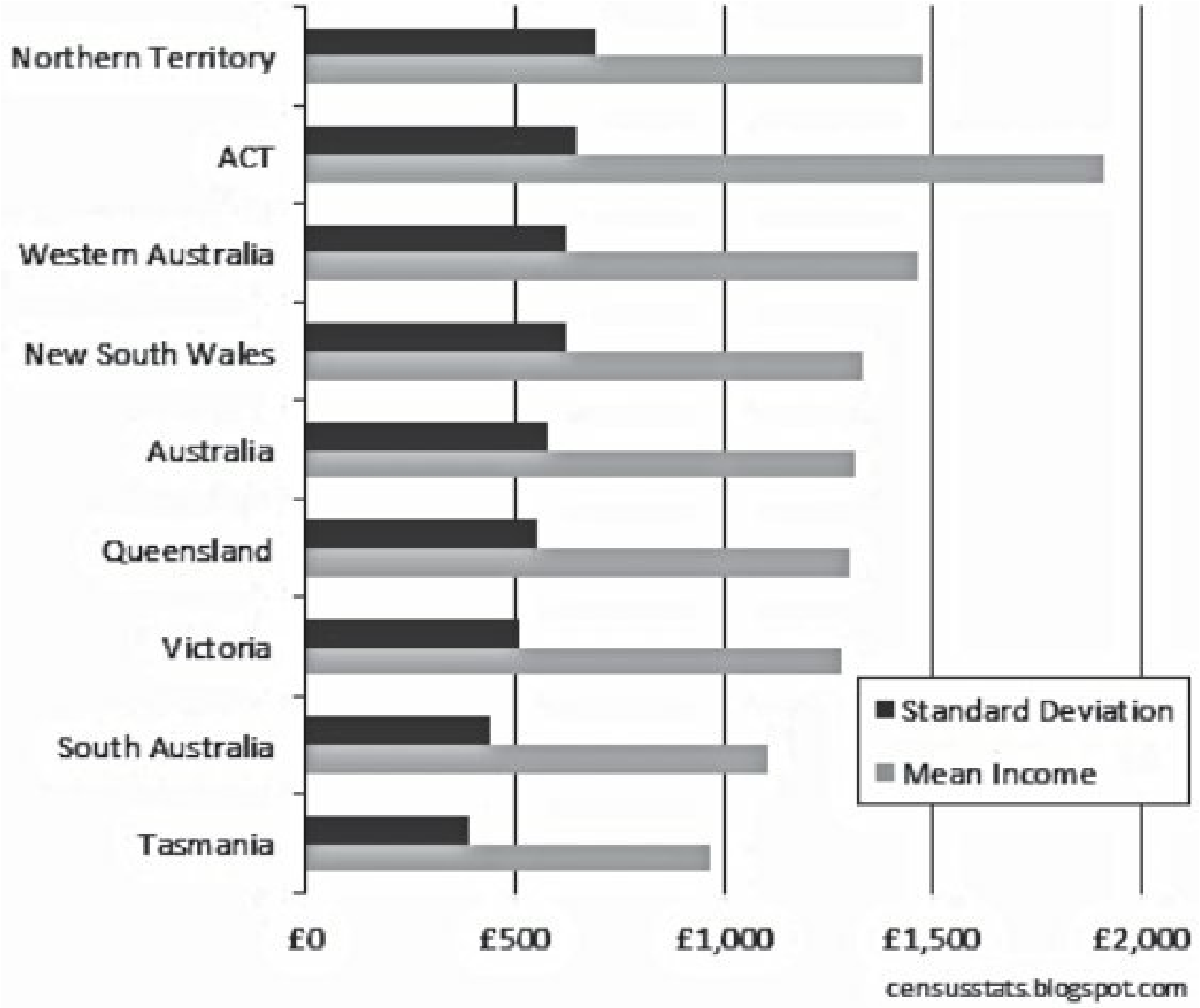

Figure 15.17 Income distribution in Australia

If we use standard deviation as a measure of income inequality, it appears that there is considerably less income inequality in Tasmania than in the ACT (Australian Capital Territory). However, if we look at the coefficients of variation (about 0.32 for ACT and 0.42 for Tasmania), we reach a rather different conclusion.

That isn't to say that the coefficient of variation is always more useful than the standard deviation. If the mean is near 0, small changes in the mean lead to large (but not necessarily meaningful) changes in the coefficient of variation, and when the mean is 0, the coefficient of variation is undefined. Also, as we shall see in Section 15.4.2, the standard deviation can be used to construct a confidence interval, but the coefficient of variation cannot.

15.4 Distributions

A **histogram** is a plot designed to show the distribution of values in a set of data. The values are first sorted, and then divided into a fixed number of equal-width bins. A plot is then drawn that shows the number of elements in each bin. The code on the left of Figure 15.18 produces the plot on the right of that figure.

```
vals = []
for i in range(1000):
    num1 = random.choice(range(0, 101))
    num2 = random.choice(range(0, 101))
    vals.append(num1+num2)
pylab.hist(vals, bins = 10)
pylab.xlabel('Number of Occurrences')
```

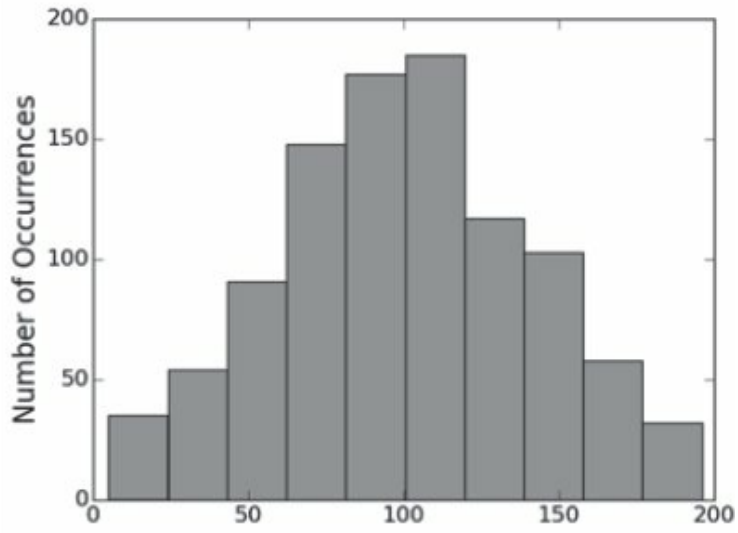

Figure 15.18 Code and the histogram it generates

The function call `pylab.hist(vals, bins = 10)` produces a histogram with ten bins. PyLab has automatically chosen the width of each bin based on the number of bins and the range of values. Looking at the code, we know that the smallest number that might appears in `vals` is 0 and the largest number 200. Therefore, the possible values on the x-axis range from 0 to 200. Each bin represents an equal fraction of the values on the x-axis, so the first bin will contain the elements 0-19, the next bin the elements 20-39, etc.

Finger exercise: In Figure 15.18, why are the bins near the middle of the histogram taller than the the bins near the sides? Hint: think about why 7 is the most common outcome of rolling a pair of dice.

By now you must be getting awfully bored with flipping coins. Nevertheless, we are going to ask you to look at yet one more coin-flipping simulation. The simulation in Figure 15.19 illustrates more of PyLab's plotting capabilities, and gives us an opportunity to get a visual notion of what standard deviation means. It produces two histograms. The first shows the result of a simulation of 100,000 trials of 100 flips of a fair coin. The second shows the result of a simulation of 100,000 trials of 1,000 flips of a fair coin.

The method `pylab.annotate` is used to place some statistics on the figure showing the histogram. The first argument is the string to be displayed on the figure. The next two arguments control where the string is placed. The argument `xycoords = 'axes fraction'` indicates the placement of the text will be expressed as a fraction of the width and height of the figure. The argument `xy = (0.67, 0.5)` indicates that the text should begin two thirds of the way from the left

edge of the figure and half way from the bottom edge of the figure.

```
def flip(numFlips):
    """Assumes numFlips a positive int"""
    heads = 0
    for i in range(numFlips):
        if random.choice(('H', 'T')) == 'H':
            heads += 1
    return heads/float(numFlips)

def flipSim(numFlipsPerTrial, numTrials):
    fracHeads = []
    for i in range(numTrials):
        fracHeads.append(flip(numFlipsPerTrial))
    mean = sum(fracHeads)/len(fracHeads)
    sd = stdDev(fracHeads)
    return (fracHeads, mean, sd)

def labelPlot(numFlips, numTrials, mean, sd):
    pylab.title(str(numTrials) + ' trials of '
                + str(numFlips) + ' flips each')
    pylab.xlabel('Fraction of Heads')
    pylab.ylabel('Number of Trials')
    pylab.annotate('Mean = ' + str(round(mean, 4))\
                   + '\nSD = ' + str(round(sd, 4)), size='x-large',
                   xycoords = 'axes fraction', xy = (0.67, 0.5))

def makePlots(numFlips1, numFlips2, numTrials):
    val1, mean1, sd1 = flipSim(numFlips1, numTrials)
    pylab.hist(val1, bins = 20)
    xmin,xmax = pylab.xlim()
    labelPlot(numFlips1, numTrials, mean1, sd1)
    pylab.figure()
    val2, mean2, sd2 = flipSim(numFlips2, numTrials)
    pylab.hist(val2, bins = 20)
    pylab.xlim(xmin, xmax)
    labelPlot(numFlips2, numTrials, mean2, sd2)

makePlots(100, 1000, 100000)
```

Figure 15.19 Plot histograms of coin flips

To facilitate comparing the two figures, we have used `pylab.xlim` to force the bounds of the x-axis in the second plot to match those in the first plot, rather than letting PyLab choose the bounds.

When the code in Figure 15.19 is run, it produces the plots in Figure 15.20. Notice that while the means in both plots are about the same, the standard deviations are quite different. The spread of outcomes is much tighter when we flip the coin 1000 times per trial than when we flip the coin 100 times per trial.

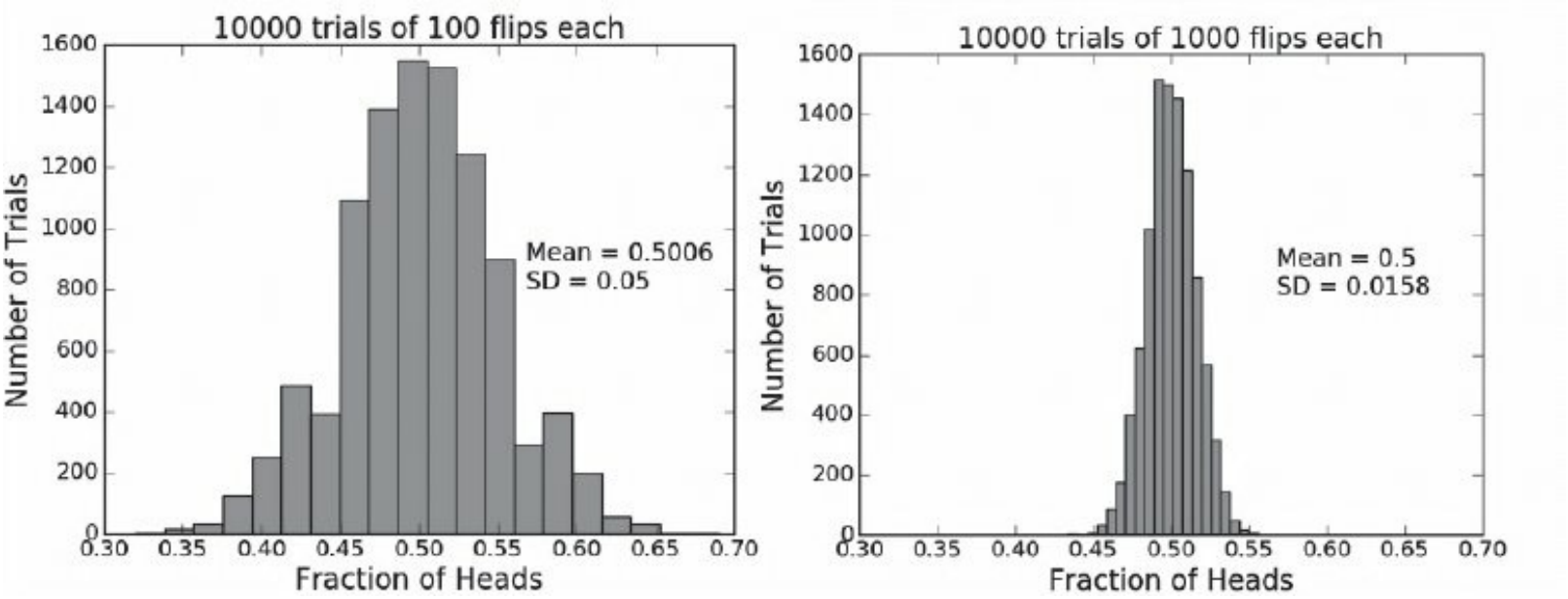

Figure 15.20 Histograms of coin flips

15.4.1 Probability Distributions

A histogram is a depiction of a **frequency distribution**. It tells us how often a random variable has taken on a value in some range, e.g., how often the fraction of times a coin came up heads was between 0.4 and 0.5. It also provides information about the relative frequency of various ranges. For example, we can easily see that the fraction of heads falls between 0.4 and 0.5 far more frequently than it falls between 0.3 and 0.4. A **probability distribution** captures the notion of relative frequency by giving the probability of a random value taking on a value within a range.

Probability distributions fall into two groups: discrete probability distributions and continuous probability distributions, depending upon whether they define the probability distribution for a discrete or a continuous random variable. A **discrete random variable** can take on one of a finite set of values, e.g., the values associated with a roll of a die. A **continuous random variable** can take on any of the infinite real values between two real numbers, e.g., the speed of a car traveling between 0 miles per hour and the car's maximum speed.

Discrete probability distributions are easier to describe. Since there are a finite number of values that the variable can take on, the distribution can be described by simply listing the probability of each value.

Continuous probability distributions are trickier. Since there are an infinite number of possible values, the probability that a continuous random variable will take on a specific value is usually 0. For example, the probability that a car is travelling at exactly 81.3457283 miles per hour is probably 0. Mathematicians like to describe continuous probability distributions using a **probability density function**, often abbreviated as **PDF**. A PDF describes the probability of a random variable lying between two values. Think of the PDF as defining a curve where the values on the x-axis lie between the minimum and maximum value of the random variable. (In some cases the x-axis is infinitely long.) Under the assumption that `x1` and `x2` lie in the domain of the random variable, the probability of the variable having a value between `x1` and `x2` is the area under the curve between `x1` and `x2`. Figure 15.21 shows the probability density functions for the expressions `random.random()` and `random.random() + random.random()`.

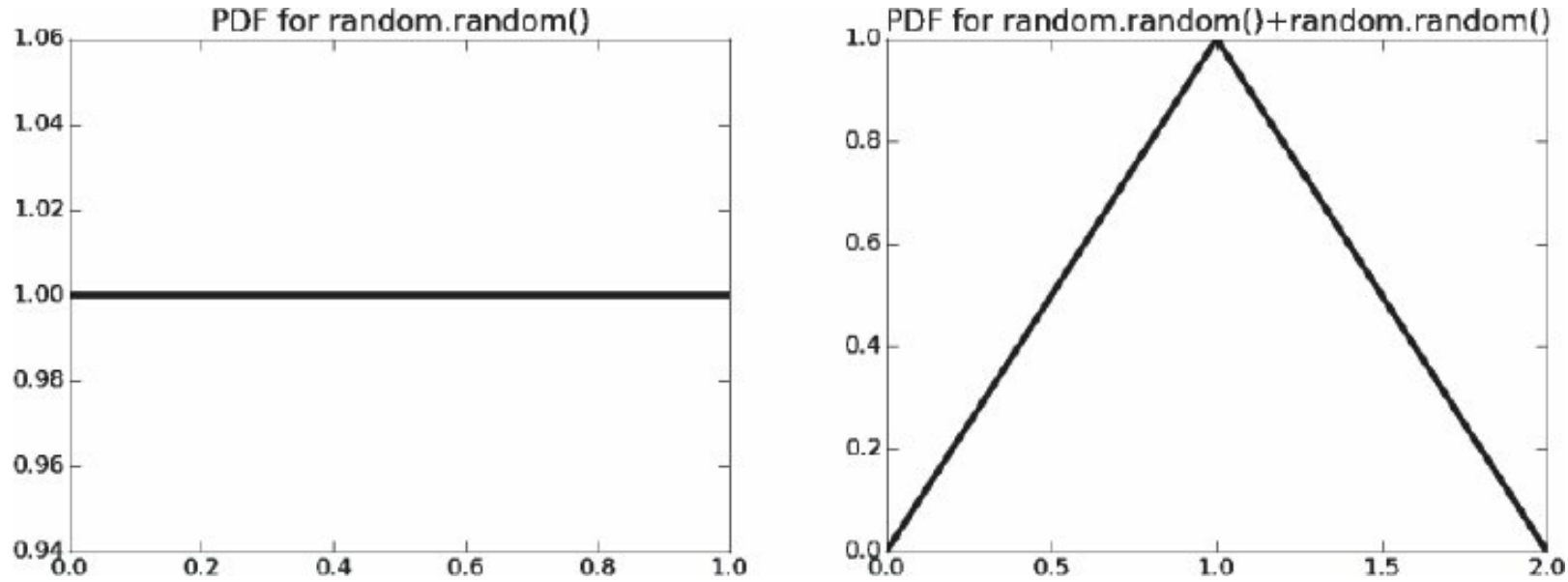

Figure 15.21: PDF for random.random

For `random.random()` the area under the curve from 0 to 1 is 1. This makes sense because we know that the probability of `random.random()` returning a value between 0 and 1 is 1. On the other hand, if we consider the area under the part of the curve for `random.random()` between 0.2 and 0.4, it is 0.2—indicating that the probability of `random.random()` returning a value between 0.2 and 0.4 is 0.2. Similarly, the area under the curve for `random.random() + random.random()` from 0 to 2 is 1, and the area under the curve from 0 to 1 is 0.5. Notice, by the way that the PDF for `random.random()` indicates that every possible interval of the same length has the same probability, whereas the PDF for `random.random() + random.random()` indicates that some intervals are more probable than others.

15.4.2 Normal Distributions

A **normal** (or **Gaussian**) **distribution** is defined by the probability density function

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} * e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ is the mean, σ the standard deviation, and e is Euler's number (roughly 2.718).[103]

If you don't feel like studying this equation, that's fine. Just remember that normal distributions peak at the mean, fall off symmetrically above and below the mean, and asymptotically approach 0. They have the nice mathematical property of being completely specified by two parameters: the mean and the standard deviation (the only two parameters in the equation). Knowing these is equivalent to knowing the entire distribution. The shape of the normal distribution resembles (in the eyes of some) that of a bell, so it sometimes is referred to as a **bell curve**.

Figure 15.22 shows part of the PDF for a normal distribution with a mean of 0 and and standard deviation of 1. We can only show a portion of the PDF, because the tails of a normal distribution converge towards 0, but don't reach it. In principle, no value has a zero probability of occurring.

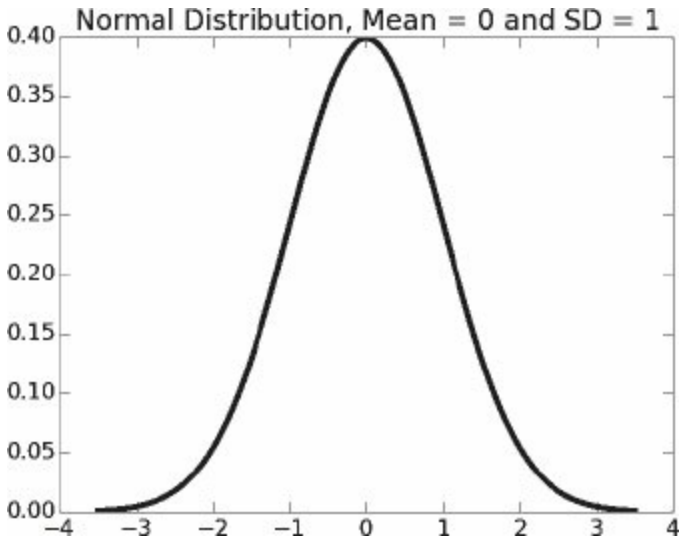

Figure 15.22: A normal distribution

Normal distributions can be easily generated in Python programs by calling `random.gauss(mu, sigma)`, which returns a randomly chosen floating point number from a normal distribution with mean and standard deviation `sigma`.

Normal distributions are frequently used in constructing probabilistic models because they have nice mathematical properties. Of course, finding a mathematically nice model is of no use if it provides a bad model of the actual data. Fortunately, many random variables have an approximately normal distribution. For example, physical properties of plants and animals (e.g., height, weight, body temperature) typically have approximately normal distributions. Importantly, many experiments have normally distributed measurement errors. This assumption was used in the early 1800s by the German mathematician and physicist Karl Gauss, who assumed a normal distribution of measurement errors in his analysis of astronomical data (which led to the normal distribution becoming known as the **Gaussian distribution** in much of the scientific community).

One of the nice properties of normal distributions is that independent of the mean and standard deviation, the number of standard deviations from the mean needed to encompass a fixed fraction of the data is a constant. For example, ~68.27% of the data will always lie with one standard deviation of the mean, ~95.45% within two standard deviations of the mean, and ~99.73% within three standard deviations of the mean. This is sometimes called the **68-95-99.7** rule, but is more often called the **empirical rule**.

The rule can be derived by integrating the formula defining a normal distribution to get the area under the curve. Looking at Figure 15.22, it is easy to believe that roughly two thirds of the total area under the curve lies between –1 and 1, roughly 95% between -2 and 2, and almost all of it between -3 and 3. But that's only one example, and it is always dangerous to generalize from a single example. We could accept the empirical rule on the unimpeachable authority of Wikipedia. However, just to be sure, and as an excuse to introduce a Python library worth knowing about, let's check it ourselves.

The SciPy library contains many mathematical functions commonly used by scientists and engineers. SciPy is organized into modules covering different scientific computing domains, such as signal processing and image processing. We will use a number of functions from SciPy later in this book. Here we use the function `scipy.integrate.quad`, which finds an approximation to the value of integrating a function between two points.

The function `scipy.integrate.quad` has three required parameters and one optional parameter:

- a function or method to be integrated (if the function takes more than one argument, it is integrated along the axis corresponding to the first argument).
- a number representing the lower limit of the integration,
- a number representing the upper limit of the integration, and
- an optional tuple supplying values for all arguments, except the first, of the function to be integrated.

The `quad` function returns a tuple of two floating point numbers. The first is an approximation to the value of the integral, and the second an estimate of the absolute error in the result.

Consider, for example, evaluating the integral of the unary function `abs` in the interval 0 to 5. We don't need any fancy math to compute the area under this curve: it's simply the area of a right triangle with base and altitude of length 5, i.e., 12.5. So, it shouldn't be a surprise that

```
print scipy.integrate.quad(abs, 0, 5)[0]
```

prints `12.5`. (The second value in the tuple returned by `quad` is roughly 10^{-13}, indicating that the approximation is quite good.)

The code in Figure 15.23 computes the area under portions of normal distributions for some randomly chosen means and standard deviations. Notice that `gaussian` is a ternary function, and therefore the code

```
print scipy.integrate.quad(gaussian, -2, 2, (0, 1))[0]
```

prints the integral from -2 to 2 of a normal distribution with mean 0 and standard deviation 1.

When we ran the code in Figure 15.23, it printed what the empirical rule predicts:

```
For mu = -1 and sigma = 6
  Fraction within 1 std = 0.6827
```

```
 Fraction within 2 std = 0.9545
 Fraction within 3 std = 0.9973
For mu = 9 and sigma = 9
 Fraction within 1 std = 0.6827
 Fraction within 2 std = 0.9545
 Fraction within 3 std = 0.9973
For mu = 1 and sigma = 5
 Fraction within 1 std = 0.6827
 Fraction within 2 std = 0.9545
 Fraction within 3 std = 0.9973
```

```
import scipy.integrate

def gaussian(x, mu, sigma):
    factor1 = (1.0/(sigma*((2*pylab.pi)**0.5)))
    factor2 = pylab.e**-(((x-mu)**2)/(2*sigma**2))
    return factor1*factor2

def checkEmpirical(numTrials):
    for t in range(numTrials):
        mu = random.randint(-10, 10)
        sigma = random.randint(1, 10)
        print('For mu =', mu, 'and sigma =', sigma)
        for numStd in (1, 2, 3):
            area = scipy.integrate.quad(gaussian, mu-numStd*sigma,
                                        mu+numStd*sigma,
                                        (mu, sigma))[0]
            print('  Fraction within', numStd, 'std =',
                  round(area, 4))

checkEmpirical(3)
```

Figure 15.23: Checking the empirical rule

People frequently use the empirical rule to derive confidence intervals. Instead of estimating an unknown value (e.g., the expected number of heads) by a single value, a **confidence interval** provides a range that is likely to contain the unknown value and a degree of confidence that the unknown value lies within that range. For example, a political poll might indicate that a candidate is likely to get 52% of the vote ±4% (i.e., the confidence interval is of size 8) with a **confidence level** of 95%. What this means is that the pollster believes that 95% of the time the candidate will receive between 48% and 56% of the vote.[104] Together the confidence interval and the confidence level are

intended to indicate the reliability of the estimate. Almost always, increasing the confidence level will require widening the confidence interval.

Suppose that we run 100 trials of 100 coin flips each. Suppose further that the mean fraction of heads is 0.4999 and the standard deviation 0.0497. For reasons we will discuss in Section 17.2, we can assume that the distribution of the means of the trials was normal. Therefore, we can conclude that if we conducted more trials of 100 flips each,

- ~95% of the time the fraction of heads will be 0.4999 ±0.0994 and
- >99% of the time the fraction of heads will be 0.4999 ±0.1491.

It is often useful to visualize confidence intervals using **error bars**. The function `showErrorBars` in Figure 15.24 calls the version of `flipSim` in Figure 15.19 and then uses

```
pylab.errorbar(xVals, means, yerr = 1.96*pylab.array(sds))
```

to produce a plot. The first two arguments give the x and y values to be plotted. The third argument says that the values in `sds` should be multiplied by 1.96 and used to create vertical error bars. We multiply by 1.96 because 95% of the data in a normal distribution falls within 1.96 standard deviations of the mean.

```
def showErrorBars(minExp, maxExp, numTrials):
    """Assumes minExp and maxExp positive ints; minExp < maxExp
         numTrials a positive integer
       Plots mean fraction of heads with error bars"""
    means, sds, xVals = [], [], []
    for exp in range(minExp, maxExp + 1):
        xVals.append(2**exp)
        fracHeads, mean, sd = flipSim(2**exp, numTrials)
        means.append(mean)
        sds.append(sd)
    pylab.errorbar(xVals, means, yerr=1.96*pylab.array(sds))
    pylab.semilogx()
    pylab.title('Mean Fraction of Heads ('
                + str(numTrials) + ' trials)')
    pylab.xlabel('Number of flips per trial')
    pylab.ylabel('Fraction of heads & 95% confidence')
```

Figure 15.24 Produce plot with error bars

The call showErrorBars(3, 10, 100) produces the plot in Figure 15.25. Unsurprisingly, the error bars shrink (the standard deviation gets smaller) as the number of flips per trial grows.

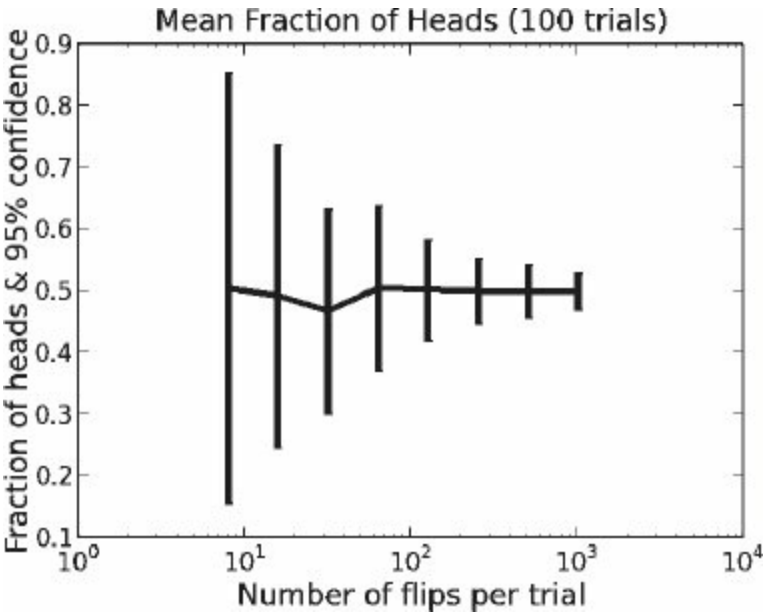

Figure 15.25 Estimates with error bars

15.4.3 Continuous and Discrete Uniform Distributions

Imagine that you take a bus that arrives at your stop every 15 minutes. If you make no effort to time your arrival at the stop to correspond to the bus schedule, your expected waiting time is uniformly distributed between 0 and 15 minutes.

A uniform distribution can be either discrete or continuous. A **continuous uniform distribution** also called a **rectangular distribution**, has the property that all intervals of the same length have the same probability. Consider the function `random.random`. As we saw in Section 15.4.1, the area under the PDF for any interval of a given length is the same. For example, the area under the curve between 0.23 and 0.33 is the same as the area under the curve between 0.53 and 0.63.

One can fully characterize a continuous uniform distribution with a single parameter, its range (i.e., minimum and maximum values). If the range of possible values is from *min* to *max*, the probability of a value falling in the range *x* to *y* is given by

$$P(x,y) = \begin{cases} \dfrac{y-x}{max-min} & if\ x \geq min\ and\ y \leq max \\ 0\ otherwise \end{cases}$$

Elements drawn from a continuous uniform distributions can be generated by calling `random.uniform(min, max)`, which returns a randomly chosen floating point number between `min` and `max`.

Discrete uniform distributions occur when each possible value occurs equally often, but the space of possible values is not continuous. For example, when a fair die is rolled, each of the six possible values is equally probable, but the outcomes are not uniformly distributed over the real numbers between 1 and 6—most values, e.g., 2.5, have a probability of 0 and a few values, e.g. 3, have a probability of $\frac{1}{6}$. One can fully characterize a discrete uniform distribution by

$$P(x) = \begin{cases} \dfrac{1}{|S|} & if\ x \in S \\ 0\ otherwise \end{cases}$$

where *S* is the set of possible values and |*S*| the number of elements in *S*.

15.4.4 Binomial and Multinomial Distributions

Random variables that can take on only a discrete set of values are called **categorical** (also **nominal** or **discrete**) **variables**.

When a categorical variable has only two possible values (e.g., success or failure), the probability distribution is called a **binomial distribution**. One way to think about a binomial distribution is as the probability of a test succeeding exactly k times times in n independent trials. If the probability of a success in a single trial is p, the probability of exactly k successes in n independent trials is given by the formula

$$\binom{n}{k} * p^k * (1-p)^{n-k}$$

where

$$\binom{n}{k} = \frac{n!}{k! * (n-k)!}$$

The formula $\binom{n}{k}$ is known as the **binomial coefficient**. One way to read it is as "n choose k," since it is equivalent to the number of subsets of size k that can be constructed from a set of size n. For example, there are

$$\binom{4}{2} = \frac{4!}{2! * 2!} = \frac{24}{4} = 6$$

subsets of size two that can be constructed from the set {1,2,3,4].

In Section 15.2, we asked about the probability of rolling exactly two 1's in ten rolls of a die. We now have the tools in hand to calculate this probability. Think of the ten rolls as ten independent trials, where the trial is a success if a 1 is rolled and a failure otherwise. The binomial distribution tells us that the probability of having exactly two successful trials out of ten is

$$\binom{10}{2} * \left(\frac{1}{6}\right)^2 * \left(\frac{5}{6}\right)^8 = 45 * \frac{1}{36} * \frac{390625}{1679616} \approx 0.291$$

Finger exercise: Implement a function that calculates the probability of rolling exactly two 3's in k rolls of a fair die. Use this function to plot the probability as k varies from 2 to 100.

The **multinomial distribution** is a generalization of the binomial distribution to categorical data with more than two possible values. It applies when there are n independent trials each of which has m mutually exclusive outcomes, with each outcome having a fixed probability of occurring. The multinomial distribution gives the probability of any given combination of numbers of occurrences of the various categories.

15.4.5 Exponential and Geometric Distributions

Exponential distributions occur quite commonly. They are often used to model inter-arrival times, e.g., of cars entering a highway or requests for a Web page.

Consider, for example, the concentration of a drug in the human body. Assume that at each time step each molecule has a constant probability p of being cleared (i.e., of no longer being in the body). The system is memoryless in the sense that at each time step the probability of a molecule being cleared is independent of what happened at previous times. At time t = 0, the probability of an individual molecule still being in the body is 1. At time t = 1, the probability of that molecule still being in the body is 1 - p. At time t = 2, the probability of that molecule still being in the body is $(1 - p)^2$. More generally, at time t the probability of an individual molecule having survived is $(1 - p)^t$, i.e., it is exponential in t.

Suppose that at time t_0 there are M_0 molecules of the drug. In general, at time t, the number of molecules will be M_0 multiplied by the probability that an individual module has survived to time t. The function `clear` implemented in Figure 15.26 plots the expected number of remaining molecules versus time.

```
def clear(n, p, steps):
    """Assumes n & steps positive ints, p a float
        n: the initial number of molecules
        p: the probability of a molecule being cleared
        steps: the length of the simulation"""
    numRemaining = [n]
    for t in range(steps):
        numRemaining.append(n*((1-p)**t))
    pylab.plot(numRemaining)
    pylab.xlabel('Time')
    pylab.ylabel('Molecules Remaining')
    pylab.title('Clearance of Drug')
```

Figure 15.26 Exponential clearance of molecules

The call `clear(1000, 0.01, 1000)` produces the plot in Figure 15.27.

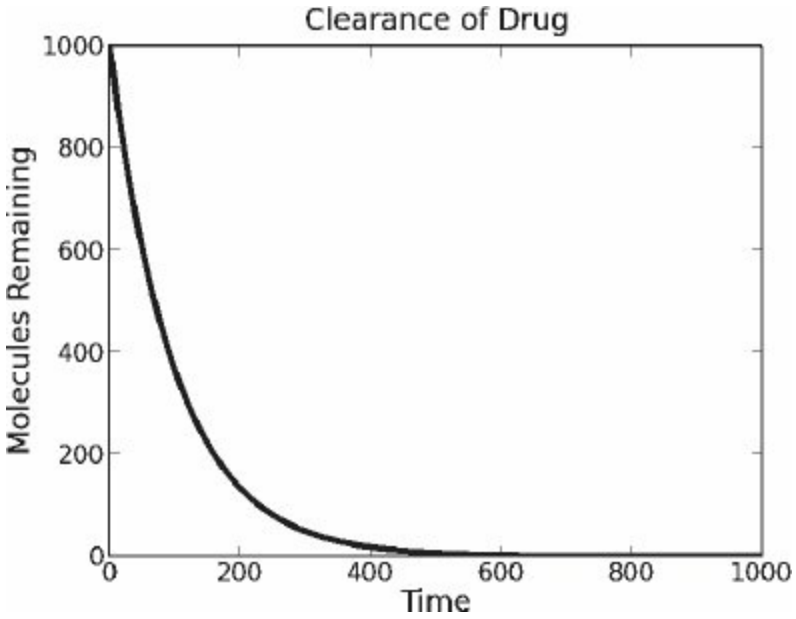

Figure 15.27: Exponential decay

This is an example of **exponential decay**. In practice, exponential decay is often talked about in terms of **half-life**, i.e., the expected time required for the initial value to decay by 50%. One can also talk about the half-life of a single item. For example, the half-life of a single molecule is the time at which the probability of that molecule having been cleared is 0.5. Notice that as time increases, the number of remaining molecules approaches 0. But it will never quite get there. This should not be interpreted as suggesting that a fraction of a molecule remains. Rather it should be interpreted as saying that since the system is probabilistic, one can never guarantee that all of the molecules have been cleared.

What happens if we make the y-axis logarithmic (by using `pylab.semilogy`)? We get the plot in Figure 15.28. In the plot in Figure 15.27, the values on the y-axis are changing exponentially quickly relative to the values on the x-axis. If we make the y-axis itself change exponentially quickly, we get a straight line. The slope of that line is the **rate of decay**.

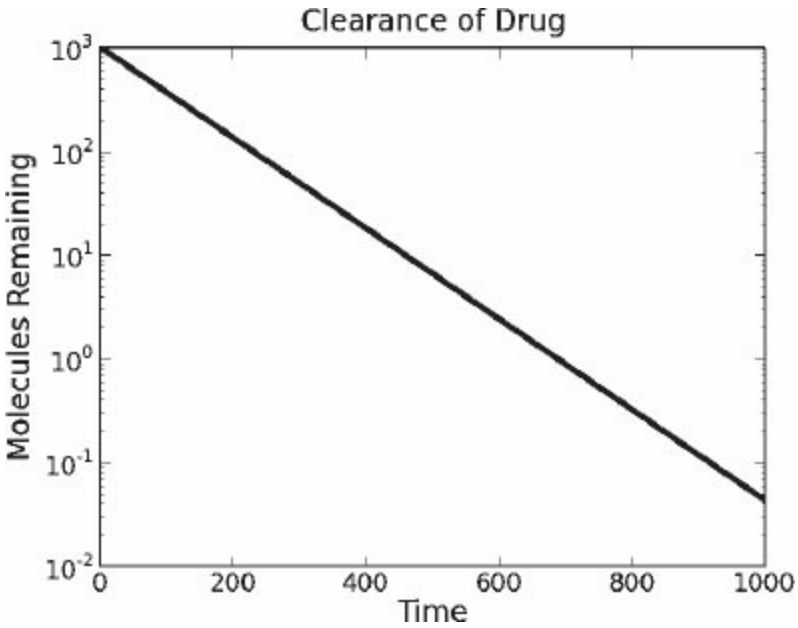

Figure 15.28: Plotting exponential decay with a logarithmic axis

Exponential growth is the inverse of exponential decay. It too is quite commonly seen in nature. Compound interest, the growth of algae in a swimming pool, and the chain reaction in an atomic bomb are all examples of exponential growth.

Exponential distributions can easily be generated in Python by calling the function `random.expovariate(lambd)`,[105] where `lambd` is 1.0 divided by the desired mean. The function returns a value between 0 and positive infinity if `lambd` is positive, and between negative infinity and 0 if `lambd` is negative.

The **geometric distribution** is the discrete analog of the exponential distribution.[106] It is usually thought of as describing the number of independent attempts required to achieve a first success (or a first failure). Imagine, for example, that you have a balky car that starts only half of the time you turn the key (or push the starter button). A geometric distribution could be used to characterize the expected number of times you would have to attempt to start the car before being successful. This is illustrated by the histogram in Figure 15.30, which was produced by the code in Figure 15.29.

```
def successfulStarts(successProb, numTrials):
    """Assumes successProb is a float representing probability of a
         single attempt being successful. numTrials a positive int
       Returns a list of the number of attempts needed before a
         success for each trial."""
    triesBeforeSuccess = []
    for t in range(numTrials):
        consecFailures = 0
        while random.random() > successProb:
            consecFailures += 1
        triesBeforeSuccess.append(consecFailures)
    return triesBeforeSuccess

probOfSuccess = 0.5
numTrials = 5000
distribution = successfulStarts(probOfSuccess, numTrials)
pylab.hist(distribution, bins = 14)
pylab.xlabel('Tries Before Success')
pylab.ylabel('Number of Occurrences Out of ' + str(numTrials))
pylab.title('Probability of Starting Each Try = '\
            + str(probOfSuccess))
```

Figure 15.29 Producing a Geometric Distribution

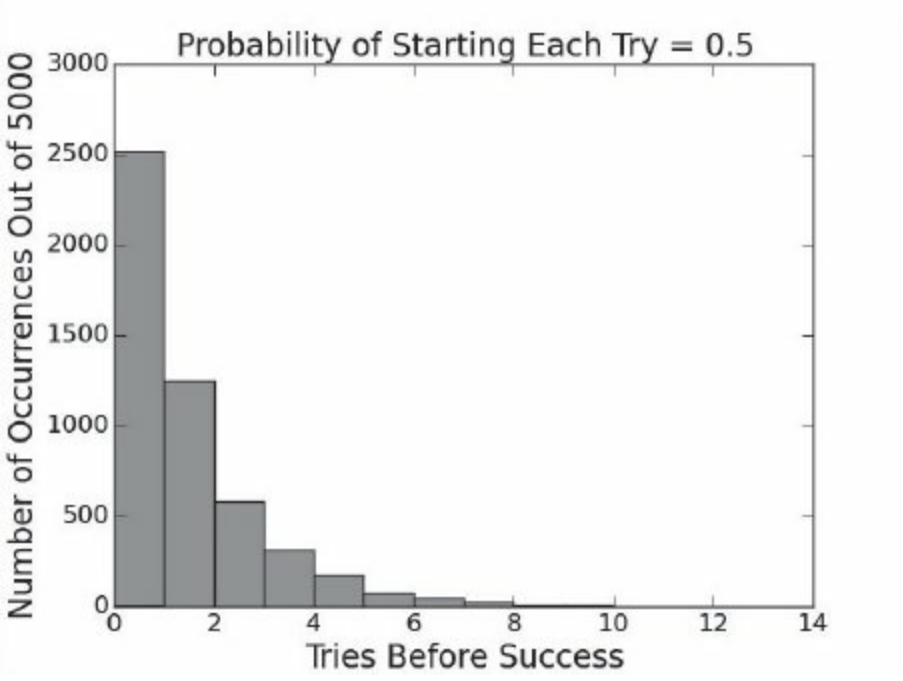

Figure 15.30 A geometric distribution

The histogram implies that most of the time you'll get the car going within a few attempts. On the other hand, the long tail suggests that on occasion you may run the risk of draining your battery before the car gets going.

15.4.6 Benford's Distribution

Benford's law defines a really strange distribution. Let S be a large set of decimal integers. How frequently would you expect each nonzero digit to appear as the first digit? Most of us would probably guess one ninth of the time. And when people are making up sets of numbers (e.g., faking experimental data or perpetrating financial fraud) this is typically true. It is not, however, typically true of many naturally occurring data sets. Instead, they follow a distribution predicted by Benford's law.

A set of decimal numbers is said to satisfy **Benford's law**[107] if the probability of the first digit being d is consistent with $P(d) = \log_{10}(1 + 1/d)$.

For example, this law predicts that the probability of the first digit being 1 is about 30%! Shockingly, many actual data sets seem to observe this law. It is possible to show that the Fibonacci sequence, for example, satisfies it perfectly. That's kind of plausible, since the sequence is generated by a formula. It's less easy to understand why such diverse data sets as iPhone pass codes, the number of Twitter followers per user, the population of countries, or the distances of stars from the earth closely approximate Benford's law.[108]

15.5 Hashing and Collisions

In Section 10.3 we pointed out that by using a larger hash table one could reduce the incidence of collisions, and thus reduce the expected time to retrieve a value. We now have the intellectual tools we need to examine that tradeoff more precisely.

First, let's get a precise formulation of the problem.

- Assume:
  - The range of the hash function is 1 to n,
  - The number of insertions is K, and
  - The hash function produces a perfectly uniform distribution of the keys used in insertions, i.e., for all keys, key, and for all integers, i, in the range 1 to n, the probability that hash(key) = i is 1/n.
- What is the probability that at least one collision occurs?

The question is exactly equivalent to asking "given K randomly generated integers in the range 1 to n, what is the probability that at least two of them are equal." If $K \geq n$, the probability is clearly 1. But what about when $K < n$?

As is often the case, it is easiest to start by answering the inverse question, "given K randomly generated integers in the range 1 to n, what is the probability that none of them are equal?"

When we insert the first element, the probability of not having a collision is clearly 1. How about the second insertion? Since there are n-1 hash results left that are not equal to the result of the first hash, n-1 out of n choices will not yield a collision. So, the probability of not getting a collision on

the second insertion is $\frac{n-1}{n}$, and the probability of not getting a collision on either of the first two insertions is $1 * \frac{n-1}{n}$. We can multiply these probabilities because for each insertion the value produced by the hash function is independent of anything that has preceded it.

The probability of not having a collision after three insertions is $1 * \frac{n-1}{n} * \frac{n-2}{n}$. And after K insertions it is $1 * \frac{n-1}{n} * \frac{n-2}{n} * \ldots * \frac{n-(K-1)}{n}$.

To get the probability of having at least one collision, we subtract this value from 1, i.e., the probability is

$$1 - \left(\frac{n-1}{n} * \frac{n-2}{n} * \ldots * \frac{n-(K-1)}{n} \right)$$

Given the size of the hash table and the number of expected insertions, we can use this formula to calculate the probability of at least one collision. If K were reasonably large, say 10,000, it would be a bit tedious to compute the probability with pencil and paper. That leaves two choices, mathematics and programming. Mathematicians have used some fairly advanced techniques to find a way to approximate the value of this series. But unless K is very large, it is easier to run some code to compute the exact value of the series:

```
def collisionProb(n, k):
    prob = 1.0
    for i in range(1, k):
        prob = prob * ((n - i)/n)
    return 1 - prob
```

If we try `collisionProb(1000, 50)` we get a probability of about 0.71 of there being at least one collision. If we consider 200 insertions, the probability of a collision is nearly 1. Does that seem a bit high to you? Let's write a simulation, Figure 15.31, to estimate the probability of at least one collision, and see if we get similar results.

```
def simInsertions(numIndices, numInsertions):
    """Assumes numIndices and numInsertions are positive ints.
       Returns 1 if there is a collision; 0 otherwise"""
    choices = range(numIndices) #list of possible indices
    used = []
    for i in range(numInsertions):
        hashVal = random.choice(choices)
        if hashVal in used: #there is a collision
            return 1
        else:
            used.append(hashVal)
    return 0

def findProb(numIndices, numInsertions, numTrials):
    collisions = 0
    for t in range(numTrials):
        collisions += simInsertions(numIndices, numInsertions)
    return collisions/numTrials
```

Figure 15.31 Simulating a hash table

If we run the code

```
print('Actual probability of a collision =', collisionProb(1000, 50))
print('Est. probability of a collision =', findProb(1000, 50, 10000))
print('Actual probability of a collision =', collisionProb(1000, 200))
print('Est. probability of a collision =', findProb(1000, 200, 10000))
```

it prints

```
Actual probability of a collision = 0.7122686568799875
Est. probability of a collision = 0.7097
Actual probability of a collision = 0.9999999994781328
Est. probability of a collision = 1.0
```

The simulation results are comfortingly similar to what we derived analytically.

Should the high probability of a collision make us think that hash tables have to be enormous to be useful? No. The probability of there being at least one collision tells us little about the expected lookup time. The expected time to look up a value depends upon the average length of the lists implementing the buckets that hold the values that collided. Assuming a uniform distribution of hash values, this is simply the number of insertions divided by the number of buckets.

15.6 How Often Does the Better Team Win?

Almost every October two teams from American Major League Baseball meet in something called the World Series. They play each other repeatedly until one of the teams has won four games, and that team is called (not entirely appropriately) the "world champion."

Setting aside the question of whether there is reason to believe that one of the participants in the World Series is indeed the best team in the world, how likely is it that a contest that can be at most seven games long will determine which of the two participants is better?

Clearly, each year one team will emerge victorious. So the question is whether we should attribute that victory to skill or to luck.

Figure 15.32 contains code that can provide us with some insight into that question. The function `simSeries` has one argument, `numSeries`, a positive integer describing the number of seven-game series to be simulated. It plots the probability of the better team winning the series against the probability of that team winning a single game. It varies the probability of the better team winning a single game from 0.5 to 1.0, and produces the plot in Figure 15.33.

Notice that for the better team to win 95% of the time (0.95 on the y-axis), it needs to be so much better that it would win more than three out of every four games between the two teams. For comparison, in 2015, the two teams in the World Series had regular season winning percentages of 58.6% (Kansas City Royals) and 55.5% (New York Mets).

```
def playSeries(numGames, teamProb):
    numWon = 0
    for game in range(numGames):
        if random.random() <= teamProb:
            numWon += 1
    return (numWon > numGames//2)

def fractionWon(teamProb, numSeries, seriesLen):
    won = 0
    for series in range(numSeries):
        if playSeries(seriesLen, teamProb):
            won += 1
    return won/float(numSeries)

def simSeries(numSeries):
    prob = 0.5
    fracsWon, probs = [], []
    while prob <= 1.0:
        fracsWon.append(fractionWon(prob, numSeries, 7))
        probs.append(prob)
        prob += 0.01
    pylab.axhline(0.95) #Draw line at 95%
    pylab.plot(probs, fracsWon, 'k', linewidth = 5)
    pylab.xlabel('Probability of Winning a Game')
    pylab.ylabel('Probability of Winning a Series')
    pylab.title(str(numSeries) + ' Seven-Game Series')

simSeries(400)
```

Figure 15.32 World Series simulation

Figure 15.33 Probability of winning a 7-game series

---

[95] Of course this doesn't stop people from believing that they are, and losing a lot of money based on that belief.

[96] A roll is fair if each of the six possible outcomes is equally likely. This is not always to be taken for granted. Excavations of Pompeii discovered "loaded" dice in which small lead weights had been inserted to bias the outcome of a roll. More recently, an online vendor's site said, "Are you unusually unlucky when it comes to rolling dice? Investing in a pair of dice that's more, uh, reliable might be just what you need."

[97] In point of fact, the values returned by `random.random` are not truly random. They are what mathematicians call pseudorandom. For almost all practical purposes, this distinction is not relevant and we shall ignore it.

[98] Though the law of large numbers had been discussed in the 16th century by Cardano, the first proof was published by Jacob Bernoulli in the early 18th century. It is unrelated to the theorem about fluid dynamics called Bernoulli's theorem, which was proved by Jacob's nephew Daniel.

[99] "On August 18, 1913, at the casino in Monte Carlo, black came up a record twenty-six times in succession [in roulette]. … [There] was a near-panicky rush to bet on red, beginning about the time black had come up a phenomenal fifteen times. In application of the maturity [of the chances] doctrine, players doubled and tripled their stakes, this doctrine leading them to believe after black came up the twentieth time that there was not a chance in a million of another repeat. In the end the unusual run enriched the Casino by some millions of francs." Huff and Geis, *How to Take a Chance*, pp. 28-29.

[100] The term "regression to the mean" was first used by Francis Galton in 1885 in a paper titled "Regression Toward Mediocrity in Hereditary Stature." In that study he observed that children of

unusually tall parents were likely to be shorter than their parents.

[101] You should be aware of the fact that the random number generators in Python 2 and Python 3 are not identical. This means that even if you set the seed, you cannot assume that a program will behave the same way across versions of the language.

[102] You'll probably never need to implement these yourself. Statistical libraries implement these and many other standard statistical functions. However, we present the code here on the off chance that some readers prefer looking at code to looking at equations.

[103] e is one of those magic irrational constants, like π, that show up all over the place in mathematics. The most common use is as the base of what are called "natural logarithms." There are many equivalent ways of defining e, including as the value of $(1 + \frac{1}{x})^x$ as x approaches infinity.

[104] For polls, confidence intervals are not typically estimated by looking at the standard deviation of multiple polls. Instead, they use something called standard error, see Section 17.3.

[105] The parameter would have been called `lambda`, but as we saw in Section 5.4, `lambda` is a reserved word in Python.

[106] The name "geometric distribution" arises from its similarity to a "geometric progression." A geometric progression is any sequence of numbers in which each number other than the first is derived by multiplying the previous number by a constant nonzero number. Euclid's *Elements* proves a number of interesting theorems about geometric progressions.

[107] The law is named after the physicist Frank Benford, who published a paper in 1938 showing that the law held on over 20,000 observations drawn from twenty different domains. However, it was first postulated in 1881 by the astronomer Simon Newcomb.

[108] http://testingbenfordslaw.com/

16 MONTE CARLO SIMULATION

In Chapters 14 and 15, we looked at different ways of using randomness in computations. Many of the examples we presented fall into the class of computation known as **Monte Carlo simulation**. Monte Carlo simulation is a technique used to approximate the probability of an event by running the same simulation multiple times and averaging the results.

Stanislaw Ulam and Nicholas Metropolis coined the term Monte Carlo simulation in 1949 in homage to the games of chance played in the casino in the Principality of Monaco. Ulam, who is best known for designing the hydrogen bomb with Edward Teller, described the invention of the model as follows:

> *The first thoughts and attempts I made to practice [the Monte Carlo Method] were suggested by a question which occurred to me in 1946 as I was convalescing from an illness and playing solitaires. The question was what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than "abstract thinking" might not be to lay it out say one hundred times and simply observe and count the number of successful plays. This was already possible to envisage with the beginning of the new era of fast computers,[109] and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes described by certain differential equations into an equivalent form interpretable as a succession of random operations. Later ... [in 1946, I] described the idea to John von Neumann, and we began to plan actual calculations.[110]*

The technique was used during the Manhattan Project to predict what would happen during a nuclear fission reaction, but did not really take off until the 1950s, when computers became both more common and more powerful.

Ulam was not the first mathematician to think about using the tools of probability to understand a game of chance. The history of probability is intimately connected to the history of gambling. It is the existence of uncertainty that makes gambling possible. And the existence of gambling provoked the development of much of the mathematics needed to reason about uncertainty. Contributions to the foundations of probability theory by Cardano, Pascal, Fermat, Bernoulli, de Moivre, and Laplace were all motivated by a desire to better understand (and perhaps profit from) games of chance.

16.1 Pascal's Problem

Most of the early work on probability theory revolved around games using dice.[111] Reputedly, Pascal's interest in the field that came to be known as probability theory began when a friend asked him whether or not it would be profitable to bet that within twenty-four rolls of a pair of dice he would roll a double 6. This was considered a hard problem in the mid-17th century. Pascal and Fermat, two pretty smart guys, exchanged a number of letters about how to resolve the issue, but it now seems like an easy question to answer:

- On the first roll the probability of rolling a 6 on each die is 1/6, so the probability of rolling a 6 with both dice is 1/36.
- Therefore, the probability of not rolling a double 6 on the first roll is 1 - 1/36 = 35/36.
- Therefore the probability of not rolling a double 6 twenty-four consecutive times is $(35/36)^{24}$, nearly 0.51, and therefore the probability of rolling a double 6 is 1 - $(35/36)^{24}$, about 0.49. In the long run it would not be profitable to bet on rolling a double 6 within twenty-four rolls.

Just to be safe, let's write a little program, Figure 16.1, to simulate Pascal's friend's game and confirm that we get the same answer as Pascal. When run the first time, the call `checkPascal(1000000)` printed

```
Probability of winning = 0.490761
```

This is indeed quite close to 1 - $(35/36)^{24}$; typing `1-(35.0/36.0)**24` into the Python shell produces `0.49140387613090342`.

```
def rollDie():
    return random.choice([1,2,3,4,5,6])

def checkPascal(numTrials):
    """Assumes numTrials an int > 0
       Prints an estimate of the probability of winning"""
    numWins = 0
    for i in range(numTrials):
        for j in range(24):
            d1 = rollDie()
            d2 = rollDie()
            if d1 == 6 and d2 == 6:
                numWins += 1
                break
    print('Probability of winning =', numWins/numTrials)
```

Figure 16.1 Checking Pascal's analysis

16.2 Pass or Don't Pass?

Not all questions about games of chance are so easily answered. In the game craps, the shooter (the person who rolls the dice) chooses between making a "pass line" or a "don't pass line" bet.

- Pass Line: Shooter wins if the first roll is a "natural" (7 or 11) and loses if it is "craps" (2, 3, or 12). If some other number is rolled, that number becomes the "point" and the shooter keeps rolling. If the shooter rolls the point before rolling a 7, the shooter wins. Otherwise the shooter loses.
- Don't Pass Line: Shooter loses if the first roll is 7 or 11, wins if it is 2 or 3, and ties (a "push" in gambling jargon) if it is 12. If some other number is rolled, that number becomes the point and shooter keeps rolling. If the shooter rolls a 7 before rolling the point, the shooter wins. Otherwise the shooter loses.

Is one of these a better bet than the other? Is either a good bet? It is possible to analytically derive the answer to these questions, but it seems easier (at least to us) to write a program that simulates a craps game, and see what happens. Figure 16.2 contains the heart of such a simulation.

```
class CrapsGame(object):
    def __init__(self):
        self.passWins, self.passLosses = 0, 0
        self.dpWins, self.dpLosses, self.dpPushes = 0, 0, 0

    def playHand(self):
        throw = rollDie() + rollDie()
        if throw == 7 or throw == 11:
            self.passWins += 1
            self.dpLosses += 1
        elif throw == 2 or throw == 3 or throw == 12:
            self.passLosses += 1
            if throw == 12:
                self.dpPushes += 1
            else:
                self.dpWins += 1
        else:
            point = throw
            while True:
                throw = rollDie() + rollDie()
                if throw == point:
                    self.passWins += 1
```

```
                    self.dpLosses += 1
                    break
                elif throw == 7:
                    self.passLosses += 1
                    self.dpWins += 1
                    break

    def passResults(self):
        return (self.passWins, self.passLosses)

    def dpResults(self):
        return (self.dpWins, self.dpLosses, self.dpPushes)
```

Figure 16.2 `CrapsGame` **class**

The values of the instance variables of an instance of class `CrapsGame` record the performance of the pass and don't pass lines since the start of the game. The observer methods `passResults` and `dpResults` return these values. The method `playHand` simulates one hand of a game. A "hand" starts when the shooter is "coming out," the term used in craps for a roll before a point is established. A hand ends when the shooter has won or lost his or her initial bet. The bulk of the code in `playHand` is merely an algorithmic description of the rules stated above. Notice that there is a loop in the `else` clause corresponding to what happens after a point is established. It is exited using a `break` statement when either a seven or the point is rolled.

Figure 16.3 contains a function that uses class `CrapsGame` to simulate a series of craps games.

```
def crapsSim(handsPerGame, numGames):
    """Assumes handsPerGame and numGames are ints > 0
       Play numGames games of handsPerGame hands; print results"""
    games = []

    #Play numGames games
    for t in range(numGames):
        c = CrapsGame()
        for i in range(handsPerGame):
            c.playHand()
        games.append(c)

    #Produce statistics for each game
    pROIPerGame, dpROIPerGame = [], []
    for g in games:
        wins, losses = g.passResults()
        pROIPerGame.append((wins - losses)/float(handsPerGame))
        wins, losses, pushes = g.dpResults()
        dpROIPerGame.append((wins - losses)/float(handsPerGame))

    #Produce and print summary statistics
    meanROI = str(round((100*sum(pROIPerGame)/numGames), 4)) + '%'
    sigma = str(round(100*stdDev(pROIPerGame), 4)) + '%'
    print('Pass:', 'Mean ROI =', meanROI, 'Std. Dev. =', sigma)
    meanROI = str(round((100*sum(dpROIPerGame)/numGames), 4)) +'%'
    sigma = str(round(100*stdDev(dpROIPerGame), 4)) + '%'
    print('Don\'t pass:','Mean ROI =', meanROI, 'Std Dev =', sigma)
```

Figure 16.3 Simulating a craps game

The structure of `crapsSim` is typical of many simulation programs:

1. It runs multiple games (think of each game as analogous to a trial in our earlier simulations) and accumulates the results. Each game includes multiple hands, so there is a nested loop.
2. It then produces and stores statistics for each game.
3. Finally, it produces and outputs summary statistics. In this case, it prints the expected return on investment (ROI) or each kind of betting line and the standard deviation of that ROI.

Return on investment is defined by the equation[112]

$$ROI = \frac{gain\ from\ investment - cost\ of\ investment}{cost\ of\ investment}$$

Since the pass and don't pass lines pay even money (if you bet $1 and win, you gain is $1), the ROI is

$$ROI = \frac{number\ of\ wins - number\ of\ losses}{number\ of\ bets}$$

For example, if you made 100 pass line bets and won half, your ROI would be

$$\frac{50 - 50}{100} = 0$$

If you bet the don't pass line 100 times and had 25 wins and 5 pushes the ROI would be

$$\frac{25 - 70}{100} = \frac{-45}{100} = -4.5$$

Let's run our craps game simulation and see what happens when we try `crapsSim(20, 10)`:[113]

```
Pass: Mean ROI = -7.0% Std. Dev. = 23.6854%
Don't pass: Mean ROI = 4.0% Std Dev = 23.5372%
```

It looks as if it would be a good idea to avoid the pass line—where the expected return on investment is a 7% loss. But the don't pass line looks like a pretty good bet. Or does it?

Looking at the standard deviations, it seems that perhaps the don't pass line is not such a good bet after all. Recall that under the assumption that the distribution is normal, the 95% confidence interval is encompassed by 1.96 standard deviations on either side of the mean. For the don't pass line, the 95% confidence interval is [4.0−1.96∗23.5372, 4.0+1.96∗23.5372]—roughly [-43%, +51%]. That certainly doesn't suggest that betting the don't pass line is a sure thing.

Time to put the law of large numbers to work; `crapsSim(1000000, 10)` prints

```
Pass: Mean ROI = -1.4204% Std. Dev. = 0.0614%
Don't pass: Mean ROI = -1.3571% Std Dev = 0.0593%
```

We can now be pretty safe in assuming that neither of these is a good bet.[114] It looks as if the don't pass line might be slightly less bad, but we probably shouldn't count on that. If the 95% confidence intervals for the pass and don't pass lines did not overlap, it would be safe to assume that the difference in the two means was statistically significant.[115] However, they do overlap, so no conclusion can be safely drawn.

Suppose that instead of increasing the number of hands per game, we increased the number of games, e.g., by making the call `crapsSim(20, 1000000)`:

```
Pass: Mean ROI = -1.4133% Std. Dev. = 22.3571%
Don't pass: Mean ROI = -1.3649% Std Dev = 22.0446%
```

The standard deviations are high—indicating that the outcome of a single game of 20 hands is highly uncertain.

One of the nice things about simulations is that they make it easy to perform "what if" experiments. For example, what if a player could sneak in a pair of cheater's dice that favored 5 over 2 (5 and 2 are on the opposite sides of a die)? To test this out, all we have to do is replace the implementation of

`rollDie` by something like

```
def rollDie():
   return random.choice([1,1,2,3,3,4,4,5,5,5,6,6])
```

This relatively small change in the die makes a dramatic difference in the odds. Running `crapsSim(1000000, 10)` yields

```
Pass: Mean ROI = 6.7385% Std. Dev. = 0.13%
Don't pass: Mean ROI = -9.5186% Std Dev = 0.1226%
```

No wonder casinos go to a lot of trouble to make sure that players don't introduce their own dice into the game!

16.3 Using Table Lookup to Improve Performance

You might not want to try running `crapsSim(100000000, 10)` at home. It takes a long time to complete on most computers. That raises the question of whether there is a simple way to speed up the simulation.

The complexity of `crapsSim` is O(playHand)*handsPerGame*numGames. The running time of `playHand` depends upon the number of times the loop in it is executed. In principle, the loop could be executed an unbounded number of times since there is no bound on how long it could take to roll either a 7 or the point. In practice, of course, we have every reason to believe it will always terminate.

Notice, however, that the result of a call to `playHand` does not depend on how many times the loop is executed, but only on which exit condition is reached. For each possible point, one can easily calculate the probability of rolling that point before rolling a 7. For example, using a pair of dice one can roll a 4 in three different ways: <1, 3>, <3, 1>, and <2, 2>; and one can roll a 7 in six different ways: <1, 6>, <6, 1>, <2, 5>, <5, 2>, <3, 4>, and <4, 3>. Therefore, exiting the loop by rolling a 7 is twice as likely as exiting the loop by rolling a 4.

Figure 16.4 contains an implementation of `playHand` that exploits this thinking. We first compute the probability of making the point before rolling a 7 for each possible value of the point, and store those values in a dictionary. Suppose, for example, that the point is 8. The shooter continues to roll until he either rolls the point or rolls craps. There are five ways of rolling an 8 (<6,2>, <2,6>, <5,3>, <3,5>, and <4,4>) and six ways of rolling a 7. So, the value for the dictionary key `8` is the value of the expression `5/11`. Having this table allows us to replace the inner loop, which contained an unbounded number of rolls, with a test against one call to `random.random`. The asymptotic complexity of this version of `playHand` is O(1).

The idea of replacing computation by **table lookup** has broad applicability and is frequently used when speed is an issue. Table lookup is an example of the general idea of **trading time for space**. As we saw in Chapter 13, it is is the key idea behind dynamic programming. We saw another example of this technique in our analysis of hashing: the larger the table, the fewer the collisions, and the faster the average lookup. In this case, the table is small, so the space cost is negligible.

```
def playHand(self):
    #An alternative, faster, implementation of playHand
    pointsDict = {4:1/3, 5:2/5, 6:5/11, 8:5/11, 9:2/5, 10:1/3}
    throw = rollDie() + rollDie()
    if throw == 7 or throw == 11:
        self.passWins += 1
        self.dpLosses += 1
    elif throw == 2 or throw == 3 or throw == 12:
        self.passLosses += 1
        if throw == 12:
            self.dpPushes += 1
        else:
            self.dpWins += 1
    else:
        if random.random() <= pointsDict[throw]: # point before 7
            self.passWins += 1
            self.dpLosses += 1
        else:                                    # 7 before point
            self.passLosses += 1
            self.dpWins += 1
```

Figure 16.4 Using table lookup to improve performance

16.4 Finding π

It is easy to see how Monte Carlo simulation is useful for tackling problems in which nondeterminism plays a role. Interestingly, however, Monte Carlo simulation (and randomized algorithms in general) can be used to solve problems that are not inherently stochastic, i.e., for which there is no uncertainty about outcomes.

Consider π. For thousands of years, people have known that there is a constant (called π since the 18th century) such that the circumference of a circle is equal to $\pi * \text{diameter}$ and the area of the circle equal to $\pi * \text{radius}^2$. What they did not know was the value of this constant.

One of the earliest estimates, $4*(8/9)^2 = 3.16$, can found in the Egyptian *Rhind Papyrus*, circa 1650 BC. More than a thousand years later, the *Old Testament* implied a different value for π when giving the specifications of one of King Solomon's construction projects,

And he made a molten sea, ten cubits from the one brim to the other: it was round all

about, and his height was five cubits: and a line of thirty cubits did compass it round about.[116]

Solving for π, $10\pi = 30$, so $\pi = 3$. Perhaps the *Bible* is simply wrong, or perhaps the molten sea wasn't perfectly circular, or perhaps the circumference was measured from the outside of the wall and the diameter from the inside, or perhaps it's just poetic license. We leave it to the reader to decide.

Archimedes of Syracuse (287-212 BCE) derived upper and lower bounds on the value of π by using a high-degree polygon to approximate a circular shape. Using a polygon with 96 sides, he concluded that $223/71 < \pi < 22/7$. Giving upper and lower bounds was a rather sophisticated approach for the time. Also, if we take his best estimate as the average of his two bounds we obtain 3.1418, an error of about 0.0002. Not bad!

Long before computers were invented, the French mathematicians Buffon (1707-1788) and Laplace (1749-1827) proposed using a stochastic simulation to estimate the value of π.[117] Think about inscribing a circle in a square with sides of length 2, so that the radius, r, of the circle is of length 1.

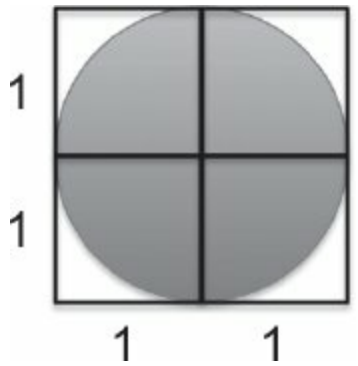

Figure 16.5 Unit circle inscribed in a square

By the definition of π, $area = \pi r^2$. Since r is 1, π = area. But what's the area of the circle? Buffon suggested that he could estimate the area of a circle by a dropping a large number of needles (which he argued would follow a random path as they fell) in the vicinity of the square. The ratio of the number of needles with tips lying within the square to the number of needles with tips lying within the circle could then be used to estimate the area of the circle.

If the locations of the needles are truly random, we know that

$$\frac{\text{needles in circle}}{\text{needles in square}} = \frac{\text{area of circle}}{\text{area of square}}$$

and solving for the area of the circle,

$$\text{area of circle} = \frac{\text{area of square} * \text{needles in circle}}{\text{needles in square}}$$

Recall that the area of a 2 by 2 square is 4, so,

$$\text{area of circle} = \frac{4 * \text{needles in circle}}{\text{needles in square}}$$

In general, to estimate the area of some region R

1. Pick an enclosing region, E, such that the area of E is easy to calculate and R lies completely within E.
2. Pick a set of random points that lie within E.
3. Let F be the fraction of the points that fall within R.
4. Multiply the area of E by F.

If you try Buffon's experiment, you'll soon realize that the places where the needles land are not truly random. Moreover, even if you could drop them randomly, it would take a very large number of needles to get an approximation of π as good as even the *Bible*'s. Fortunately, computers can randomly drop simulated needles at a ferocious rate.

Figure 16.6 contains a program that estimates π using the Buffon-Laplace method. For simplicity, it considers only those needles that fall in the upper right-hand quadrant of the square.

The function `throwNeedles` simulates dropping a needle by first using `random.random` to get a pair of positive Cartesian coordinates (x and y values) representing the position of the needle with respect to the center of the square. It then uses the Pythagorean theorem to compute the hypotenuse of the right triangle with base x and height y. This is the distance of the tip of the needle from the origin (the center of the square). Since the radius of the circle is 1, we know that the needle lies within the circle if and only if the distance from the origin is no greater than 1. We use this fact to count the number of needles in the circle.

The function `getEst` uses `throwNeedles` to find an estimate of π by first dropping `numNeedles` needles, and then averaging the result over `numTrials` trials. It then returns the mean and standard deviation of the trials.

The function `estPi` calls `getEst` with an ever-growing number of needles until the standard deviation returned by getEst is no larger than `precision/1.96`. Under the assumption that the errors are normally distributed, this implies that 95% of the values lie within `precision` of the mean.

```
def throwNeedles(numNeedles):
    inCircle = 0
    for Needles in range(1, numNeedles + 1):
        x = random.random()
        y = random.random()
        if (x*x + y*y)**0.5 <= 1:
            inCircle += 1
    #Counting needles in one quadrant only, so multiply by 4
    return 3*(inCircle/numNeedles)

def getEst(numNeedles, numTrials):
    estimates = []
    for t in range(numTrials):
        piGuess = throwNeedles(numNeedles)
        estimates.append(piGuess)
    sDev = stdDev(estimates)
    curEst = sum(estimates)/len(estimates)
    print('Est. =', str(round(curEst, 5)) + ',',
          'Std. dev. =', str(round(sDev, 5)) + ',',
          'Needles =', numNeedles)
    return (curEst, sDev)

def estPi(precision, numTrials):
    numNeedles = 1000
    sDev = precision
    while sDev > precision/1.96:
        curEst, sDev = getEst(numNeedles, numTrials)
        numNeedles *= 2
    return curEst
```

Figure 16.6 Estimating π

When we ran `estPi(0.01, 100)` it printed

```
Est. = 3.14844, Std. dev. = 0.04789, Needles = 1000
Est. = 3.13918, Std. dev. = 0.0355, Needles = 2000
Est. = 3.14108, Std. dev. = 0.02713, Needles = 4000
Est. = 3.14143, Std. dev. = 0.0168, Needles = 8000
Est. = 3.14135, Std. dev. = 0.0137, Needles = 16000
```

```
Est. = 3.14131, Std. dev. = 0.00848, Needles = 32000
Est. = 3.14117, Std. dev. = 0.00703, Needles = 64000
Est. = 3.14159, Std. dev. = 0.00403, Needles = 128000
```

As one would expect, the standard deviations decreased monotonically as we increased the number of samples. In the beginning the estimates of the value of π also improved steadily. Some were above the true value and some below, but each increase in `numNeedles` led to an improved estimate. With 1000 samples per trial, the simulation's estimate was already better than those of the *Bible* and the *Rhind Papyrus*.

Curiously, the estimate got worse when the number of needles went from 8,000 to 16,000, since 3.14135 is farther from the true value of π than is 3.14143. However, if we look at the ranges defined by one standard deviation around each of the means, both ranges contain the true value of π, and the range associated with the larger sample size is smaller. Even though the estimate generated with 16,000 samples happens to be farther from the actual value of π, we should have more confidence in its accuracy. This is an extremely important notion. It is not sufficient to produce a good answer. We have to have a valid reason to be confident that it is in fact a good answer. And when we drop a large enough number of needles, the small standard deviation gives us reason to be confident that we have a correct answer. Right?

Not exactly. Having a small standard deviation is a necessary condition for having confidence in the validity of the result. It is not a sufficient condition. The notion of a statistically valid conclusion should never be confused with the notion of a correct conclusion.

Each statistical analysis starts with a set of assumptions. The key assumption here is that our simulation is an accurate model of reality. Recall that the design of our Buffon-Laplace simulation started with a little algebra demonstrating how we could use the ratio of two areas to find the value of π. We then translated this idea into code that depended upon a little geometry and on the randomness of `random.random`.

Let's see what happens if we get any of this wrong. Suppose, for example, we replace the `4` in the last line of the function `throwNeedles` by a `2`, and again run `estPi(0.01, 100)`. This time it prints

```
Est. = 1.57422, Std. dev. = 0.02394, Needles = 1000
Est. = 1.56959, Std. dev. = 0.01775, Needles = 2000
Est. = 1.57054, Std. dev. = 0.01356, Needles = 4000
Est. = 1.57072, Std. dev. = 0.0084, Needles = 8000
Est. = 1.57068, Std. dev. = 0.00685, Needles = 16000
Est. = 1.57066, Std. dev. = 0.00424, Needles = 32000
```

The standard deviation for a mere 32,000 needles suggests that we should have a fair amount of confidence in the estimate. But what does that really mean? It means that we can be reasonably confident that if we were to draw more samples from the same distribution, we would get a similar value. It says nothing about whether or not this value is close to the actual value of π. If you are going to remember only one thing about statistics, remember this: a statistically valid conclusion should not be confused with a correct conclusion!

Before believing the results of a simulation, we need to have confidence both that our conceptual model is correct and that we have correctly implemented that model. Whenever possible, one should

attempt to validate results against reality. In this case, one could use some other means to compute an approximation to the area of a circle (e.g., physical measurement) and check that the computed value of π is at least in the right neighborhood.

16.5 Some Closing Remarks About Simulation Models

For most of the history of science, theorists used mathematical techniques to construct purely analytical models that could be used to predict the behavior of a system from a set of parameters and initial conditions. This led to the development of important mathematical tools ranging from calculus to probability theory. These tools helped scientists develop a reasonably accurate understanding of the macroscopic physical world.

As the 20th century progressed, the limitations of this approach became increasingly clear. Reasons for this include:

- An increased interest in the social sciences, e.g., economics, led to a desire to construct good models of systems that were not mathematically tractable.
- As the systems to be modeled grew increasingly complex, it seemed easier to successively refine a series of simulation models than to construct accurate analytic models.
- It is often easier to extract useful intermediate results from a simulation than from an analytical model, e.g., to play "what if" games.
- The availability of computers made it feasible to run large-scale simulations. Until the advent of the modern computer in the middle of the 20th century the utility of simulation was limited by the time required to perform calculations by hand.

Simulation models are **descriptive**, not **prescriptive**. They tell how a system works under given conditions; not how to arrange the conditions to make the system work best. A simulation does not optimize, it merely describes. That is not to say that simulation cannot be used as part of an optimization process. For example, simulation is often used as part of a search process in finding an optimal set of parameter settings.

Simulation models can be classified along three dimensions:

- Deterministic versus stochastic,
- Static versus dynamic, and
- Discrete versus continuous.

The behavior of a **deterministic simulation** is completely defined by the model. Rerunning a simulation will not change the outcome. Deterministic simulations are typically used when the system being modeled is too complex to analyze analytically, e.g., the performance of a processor chip. **Stochastic simulations** incorporate randomness in the model. Multiple runs of the same model may generate different values. This random element forces us to generate many outcomes to see the range of possibilities. The question of whether to generate 10 or 1000 or 100,000 outcomes is a statistical question, as discussed earlier.

In a **static model**, time plays no essential role. The needle-dropping simulation used to estimate π in this chapter is an example of a static simulation. In a **dynamic model**, time, or some analog, plays an essential role. In the series of random walks simulated in Chapter 14, the number of steps taken

was used as a surrogate for time.

In a **discrete model**, the values of pertinent variables are enumerable, e.g., they are integers. In a **continuous model**, the values of pertinent variables range over non-enumerable sets, e.g., the real numbers. Imagine analyzing the flow of traffic along a highway. We might choose to model each individual car, in which case we have a discrete model. Alternatively, we might choose to treat traffic as a flow, where changes in the flow can be described by differential equations. This leads to a continuous model. In this example, the discrete model more closely resembles the physical situation (nobody drives half a car, though some cars are half the size of others), but is more computationally complex than a continuous one. In practice, models often have both discrete and continuous components. For example, one might choose to model the flow of blood through the human body using a discrete model for blood (i.e., modeling individual corpuscles) and a continuous model for blood pressure.

---

[109] Ulam was probably referring to the ENIAC, which performed about 10^3 additions a second (and weighed 25 tons). Today's computers perform about 10^9 additions a second.

[110] Eckhardt, Roger (1987). "Stan Ulam, John von Neumann, and the Monte Carlo method," *Los Alamos Science*, Special Issue (15), 131-137.

[111] Archeological excavations suggest that dice are the human race's oldest gambling implement. The oldest known "modern" six-sided die dates to about 600 BCE, but Egyptian tombs dating to about 2000 BCE contain artifacts resembling dice. Typically, these early dice were made from animal bones; in gambling circles people still use the phrase "rolling the bones."

[112] More precisely, this equation defines what is often called "simple ROI." It does not account for the possiblity that there might be a gap in time between when the investment is made and when the gain attributable to that investment occurs. This gap should be accounted for when the time between making an investment and seeing the financial return is large (e.g., investing in a college education). This is probably not an issue at the craps table.

[113] Since these programs incorporate randomness, you should not expect to get identical results if you run the code yourself. More important, do not place any bets until you have read the entire section!

[114] In fact, the means of the estimated ROIs are close to the actual ROIs. Grinding through the probabilities yields an ROI of -1.414% for the pass line and -1.364% for the don't pass line.

[115] We discuss statistical significance in more detail in Chapter 19.

[116] King James Bible, 1 Kings 7.23.

[117] Buffon proposed the idea first, but there was an error in his formulation that was later corrected by Laplace.

17 SAMPLING AND CONFIDENCE INTERVALS

Recall that inferential statistics involves making inferences about a **population** of examples by analyzing a randomly chosen subset of that population. This subset is called a **sample**.

Sampling is important because it is often not possible to observe the entire population of interest. A physician cannot count the number of a species of bacterium in a patient's blood stream, but it is possible to measure the population in a small sample of the patient's blood, and from that to infer characteristics of the total population. If you wanted to know the average weight of eighteen-year-old Americans, you could try and round them all up, put them on a very large scale, and then divide by the number of people. Alternatively, you could round up fifty randomly chose eighteen-year-olds, compute their mean weight, and assume that their mean weight was a reasonable estimate of the mean weight of the entire population of eighteen-year-olds.

The correspondence between the sample and the population of interest is of overriding importance. If the sample is not representative of the population, no amount of fancy mathematics will lead to valid inferences. A sample of fifty women or fifty Asian-Americans or fifty football players cannot be used to make valid inferences about the average weight of the population of all eighteen-year-olds in America.

In this book, we focus on **probability sampling**. With probability sampling, each member of the population of interest has some nonzero probability of being included in the sample. In a **simple random sample**, each member of the population has an equal chance of being chosen for the sample. In **stratified sampling**, the population is first partitioned into subgroups, and then the sample is built by randomly sampling from each subgroup. Stratified sampling can be used to increase the probability that a sample is representative of the population as a whole. For example, ensuring that the fraction of men and women in a sample matches the fraction of men and women in the population increases the probability that that the mean weight of the sample, the **sample mean**, will be a good estimate of the mean weight of the whole population, the **population mean**.

17.1 Sampling the Boston Marathon

Each year since 1897, athletes (mostly runners, but since 1975 there has been a wheelchair division) have gathered in Massachusetts to participate in the Boston Marathon. In recent years, around twenty thousand hardy souls per year have successfully taken on the 42.195 km (26 mile, 385 yard) course.

A file containing data from the 2012 race is available on the Web site associated with this book.

The file (`bm_results2012.txt`) is in a comma-separated format, and contain the name, gender, age, division, country, and time for each participant. Figure 17.1 contains the first few lines of the contents of `the file`.

```
"Gebremariam Gebregziabher",M,27,14,ETH,142.93
"Matebo Levy",M,22,2,KEN,133.10
"Cherop Sharon",F,28,1,KEN,151.83
"Chebet Wilson",M,26,5,KEN,134.93
"Dado Firehiwot",F,28,4,ETH,154.93
"Korir Laban",M,26,6,KEN,135.48
"Jeptoo Rita",F,31,6,KEN,155.88
"Korir Wesley",M,29,1,KEN,132.67
"Kipyego Bernard",M,25,3,KEN,133.22
```

Figure 17.1 The first few lines in `bm_results2012.txt`

Since complete data about the results of each race is easily available, there is no pragmatic need to using sampling to derive statistics about a race. However, it is pedagogically useful to compare statistical estimates derived from samples to the actual value being estimated.

The code in Figure 17.2 produces the plot shown in Figure 17.3. The function `getBMData` reads data from a file containing information about each of the competitors in the race. It returns the data in a dictionary with six elements. Each key describes the type of data (e.g., `'name'` or `'gender'`) contained in the elements of a list associated with that key. For example, `data['time']` is a list of floats containing the finishing time of each competitor, `data['name'][i]` is the name of the i^{th} competitor, and `data['time'][i]` is the finishing time of the i^{th} competitor. The function `makeHist` produces a visual representation of the finishing times.

```
def getBMData(filename):
    """Read the contents of the given file. Assumes the file
    in a comma-separated format, with 6 elements in each entry:
    0. Name (string), 1. Gender (string), 2. Age (int)
    3. Division (int), 4. Country (string), 5. Overall time (float)
    Returns: dict containing a list for each of the 6 variables."""

    data = {}
    f = open(filename)
    line = f.readline()
    data['name'], data['gender'], data['age'] = [], [], []
    data['division'], data['country'], data['time'] = [], [], []
    while line != '':
        split = line.split(',')
        data['name'].append(split[0])
        data['gender'].append(split[1])
        data['age'].append(int(split[2]))
        data['division'].append(int(split[3]))
        data['country'].append(split[4])
        data['time'].append(float(split[5][:-1])) #remove \n
        line = f.readline()
    f.close()
    return data

def makeHist(data, bins, title, xLabel, yLabel):
    pylab.hist(data, bins)
    pylab.title(title)
    pylab.xlabel(xLabel)
    pylab.ylabel(yLabel)
    mean = sum(data)/len(data)
    std = stdDev(data)
    pylab.annotate('Mean = ' + str(round(mean, 2)) +\
              '\nSD = ' + str(round(std, 2)), fontsize = 20,
              xy = (0.65, 0.75), xycoords = 'axes fraction')

times = getBMData('bm_results2012.txt')['time']
makeHist(times, 20, '2012 Boston Marathon',
             'Minutes to Complete Race', 'Number of Runners')
```

Figure 17.2: Read data and produce plot of Boston Marathon

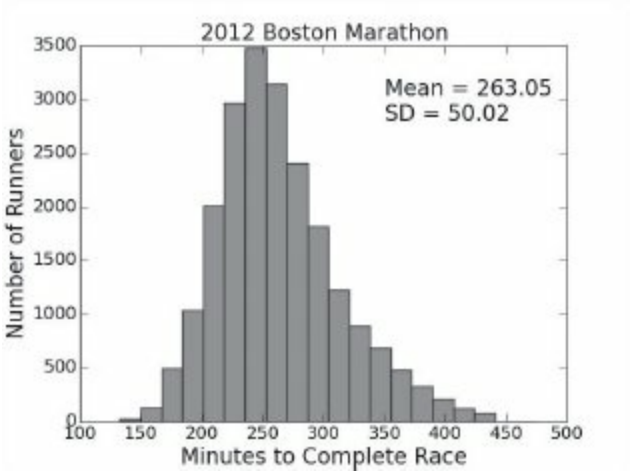

Figure 17.3: Boston Marathon finishing times

The distribution of finishing times resembles a normal distribution, but is clearly not normal because of the fat tail on the right.

Now, let's pretend that we don't have access to the data about all competitors, and instead want to estimate some statistics about the finishing times of the entire field by sampling a small number of randomly chosen competitors.

The code in Figure 17.4 creates a simple random sample of the elements of `times`, and then uses that sample to estimate the mean and standard deviation of `times`. The function `sampleTimes` uses `random.sample(times, numExamples)` to extract the sample. The invocation of `random.sample` returns a list of size `numExamples` of randomly chosen distinct elements from the list `times`. After extracting the sample, `sampleTimes` produces a histogram showing the distribution of values in the sample.

```
def sampleTimes(times, numExamples):
    """Assumes times a list of floats representing finishing
       times of all runners. numExamples an int
       Generates a random sample of size numExamples, and produces
       a histogram showing the distribution along with its mean and
       standard deviation"""
    sample = random.sample(times, numExamples)
    makeHist(sample, 10, 'Sample of Size ' + str(numExamples),
             'Minutes to Complete Race', 'Number of Runners')

sampleSize = 40
sampleTimes(times, sampleSize)
```

Figure 17.4: Sampling finishing times

As Figure 17.5 shows, the distribution of the sample is much farther from normal than the

distribution from which it was drawn. This is not surprising, given the small sample size. What's more surprising is that despite the small sample size (40 out of about 21,000) the estimated mean differs from the population mean by less than 2%. Did we get lucky, or is there reason to expect that the estimate of the mean will be pretty good? To put it another way, can we express in a quantitative way how much confidence we should have in our estimate?

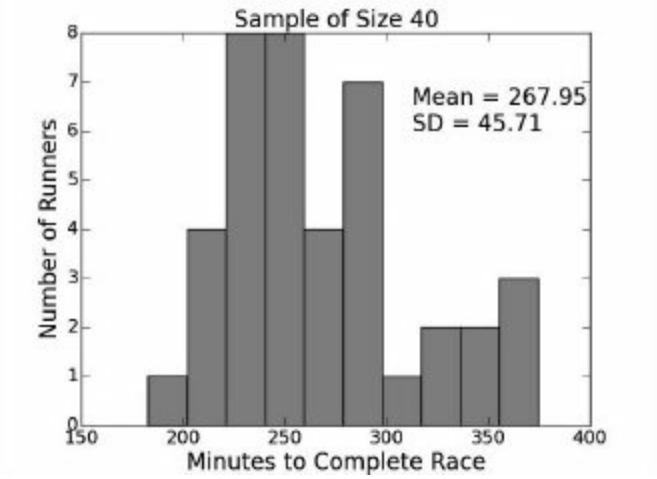

Figure 17.5: Analyzing a small sample

As we discussed in Chapters 15 and 16, it is often useful to provide a confidence interval and confidence level to indicate the reliability of the estimate. Given a single sample (of any size) drawn from a larger population, the best estimate of the mean of the population is the mean of the sample. Estimating the width of the confidence interval required to achieve a desired confidence level is trickier. It depends, in part, upon the size of the sample.

It's easy to understand why the size of the sample is important. The law of large numbers tells us that as the sample size grows, the distribution of the values of the sample is more likely to resemble the distribution of the population from which the sample is drawn. Consequently, as the sample size grows, the sample mean and the sample standard deviation are likely to be closer to the population mean and population standard deviation.

So, bigger is better, but how big is big enough? That depends upon the variance of the population. The higher the variance, the more samples are needed. Consider two normal distributions, one with a mean of 0 and standard deviation of 1, and the other with a mean of 0 and a standard deviation of 100. If we were to select one randomly chosen element from one of these distributions and use it to estimate the mean of the distribution, the probability of that estimate being within any desired accuracy, $\in$, of the true mean (0), would be equal to the area under the probability density function between $-\in$ and $\in$ (see Section 15.4.1). The code in Figure 17.6 computes and prints these probabilities for $\in = 3$ minutes.

```
import scipy.integrate

def gaussian(x, mu, sigma):
    factor1 = (1/(sigma*((2*pylab.pi)**0.5)))
    factor2 = pylab.e**-(((x-mu)**2)/(2*sigma**2))
    return factor1*factor2

area = round(scipy.integrate.quad(gaussian, -3, 3, (0, 1))[0], 4)
print('Probability of being within 3',
      'of true mean of tight dist. =', area)
area = round(scipy.integrate.quad(gaussian, -3, 3, (0, 100))[0], 4)
print('Probability of being within 3',
      'of true mean of wide dist. =', area)
```

Figure 17.6: Effect of variance on estimate of mean

When the code in Figure 17.6 is run it prints

```
Probability of being within 3 of true mean of tight dist. = 0.9973
Probability of being within 3 of true mean of wide dist. = 0.0239
```

The code in Figure 17.7 plots the mean of each of 1000 samples of size 40 from two normal distributions. Again, each distribution has a mean of 0, but one has a standard deviation of 1 and the other a standard deviation of 100.

The left side of Figure 17.8 shows the mean of each sample. As expected, when the population standard deviation is 1, the sample means are all near the population mean of 0, which is why no distinct circles are visible—they are so dense that they merge into what appears to be a bar. In contrast, when the standard deviation of the population is 100, the sample means are scattered in a hard-to-discern pattern.

However, when we look at a histogram of the means when the standard deviation is 100, the right side of Figure 17.8, something important emerges: the means form a distribution that is close to a normal distribution centered around 0. That the right side of Figure 17.8 looks the way it does is not an accident. It is a consequence of the Central Limit Theorem, the most famous theorem in all of probability and statistics.

```
def testSamples(numTrials, sampleSize):
    tightMeans, wideMeans = [], []
    for t in range(numTrials):
        sampleTight, sampleWide = [], []
        for i in range(sampleSize):
            sampleTight.append(random.gauss(0, 1))
            sampleWide.append(random.gauss(0, 100))
        tightMeans.append(sum(sampleTight)/len(sampleTight))
        wideMeans.append(sum(sampleWide)/len(sampleWide))
    return tightMeans, wideMeans

tightMeans, wideMeans = testSamples(1000, 40)
pylab.plot(wideMeans, 'y*', label = ' SD = 100')
pylab.plot(tightMeans, 'bo', label = 'SD = 1')
pylab.xlabel('Sample Number')
pylab.ylabel('Sample Mean')
pylab.title('Means of Samples of Size ' + str(40))
pylab.legend()

pylab.figure()
pylab.hist(wideMeans, bins = 20, label = 'SD = 100')
pylab.title('Distribution of Sample Means')
pylab.xlabel('Sample Mean')
pylab.ylabel('Frequency of Occurrence')
pylab.legend()
```

Figure 17.7: Compute and plot sample means

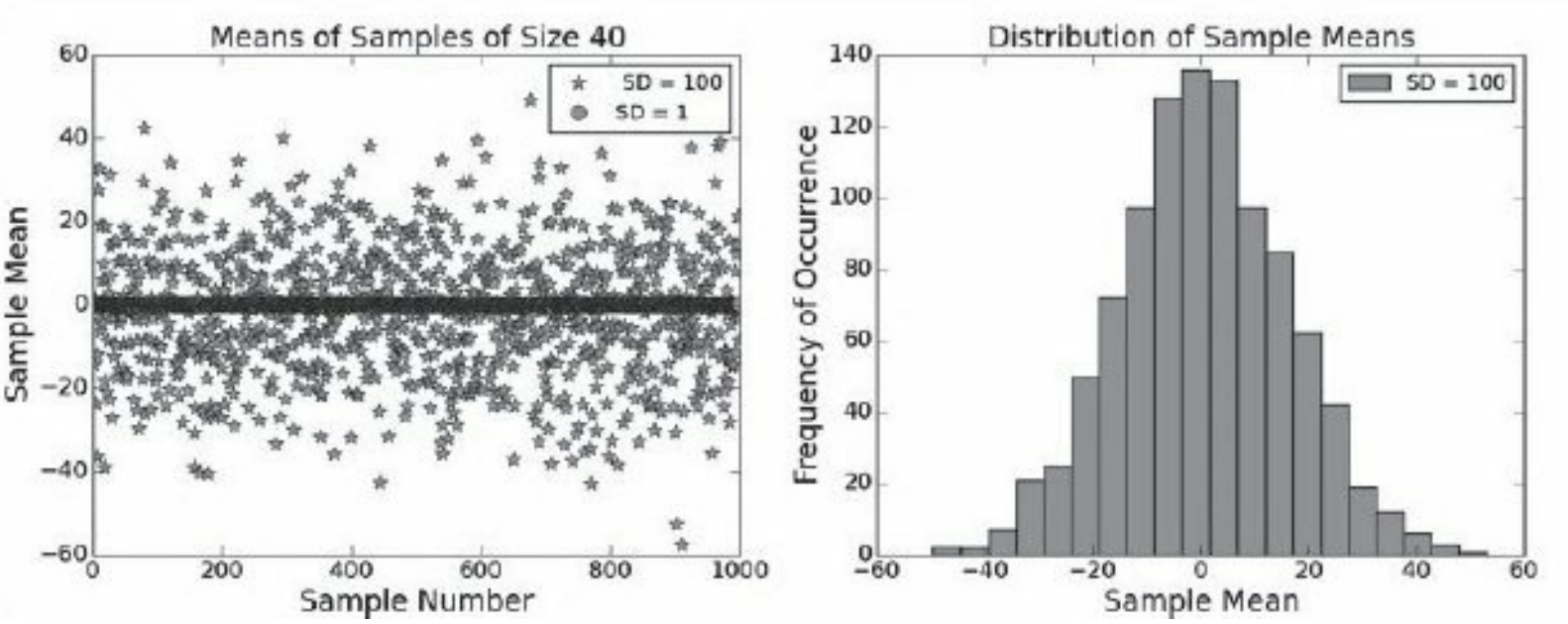

Figure 17.8: Sample means

17.2 The Central Limit Theorem

The central limit theorem explains why it is possible to use a single sample drawn from a population to estimate the variability of the means of a set of hypothetical samples drawn from the same population.

A version of the **Central Limit Theorem** (**CLT** to its friends) was first published by Laplace in 1810, and then refined by Poisson in the 1820s. But the CLT as we know it today is a product of work done by a sequence of prominent mathematicians in the first half of the 20th century.

Despite (or maybe because of) the impressive list of mathematicians who have worked on it, the CLT is really quite simple. It says that

- Given a set of sufficiently large samples drawn from the same population, the means of the samples (the sample means) will be approximately normally distributed,
- This normal distribution will have a mean close to the mean of the population, and
- The variance (as defined in Section 15.3) of the sample means will be close to the variance of the population divided by the sample size.

Let's look at an example of the CLT in action. Imagine that you had a die with the property that that each roll would yield a random real number between 0 and 5. The code in Figure 17.9 simulates rolling such a die many times, prints the mean and variance (the function `variance` is defined in Figure 15.8), and then plots a histogram showing the probability of ranges of numbers getting rolled. It also simulates rolling 100 dice many times and plots (on the same figure) a histogram of the mean value of those 100 dice. The `hatch` keyword argument is used to visually distinguish one histogram from the other.

The `weights` keyword is bound to an array of the same length as the first argument to `hist`, and is used to assign a weight to each element in the first argument. In the resulting histogram, each value

in a bin contributes its associated weight towards the bin count (instead of the usual 1). In this example, we use `weights` to scale the y values to the relative rather than absolute size of each bin. Therefore, for each bin, the value on the y-axis is the probability of the mean falling within that bin.

```
def plotMeans(numDicePerTrial, numDiceThrown, numBins, legend,
              color, style):
    means = []
    numTrials = numDiceThrown//numDicePerTrial
    for i in range(numTrials):
        vals = 0
        for j in range(numDicePerTrial):
            vals += 5*random.random()
        means.append(vals/numDicePerTrial)
    pylab.hist(means, numBins, color = color, label = legend,
               weights = pylab.array(len(means)*[1])/len(means),
               hatch = style)
    return sum(means)/len(means), variance(means)

mean, var = plotMeans(1, 100000, 11, '1 die', 'w', '*')
print('Mean of rolling 1 die =', round(mean,4),
      'Variance =', round(var,4))
mean, var = plotMeans(100, 100000, 11,
                      'Mean of 100 dice', 'w', '//')
print('Mean of rolling 100 dice =', round(mean, 4),
      'Variance =', round(var, 4))
pylab.title('Rolling Continuous Dice')
pylab.xlabel('Value')
pylab.ylabel('Probability')
pylab.legend()
```

Figure 17.9: Estimating the mean of a continuous die

When run, the code produced the plot in Figure 17.10, and printed,

```
Mean of rolling 1 die = 2.4974 Variance = 2.0904
Mean of rolling 100 dice = 2.4981 Variance = 0.02
```

In each case the mean was quite close to the expected mean of 2.5. Since our die is fair, the probability distribution for one die is almost perfectly uniform,[118] i.e., very far from normal. However, when we look at the average value of 100 dice, the distribution is almost perfectly normal, with the peak including the expected mean. Furthermore, the variance of the mean of the 100 rolls is close to the variance of the value of a single roll divided by 100. All is as predicted by the CLT.

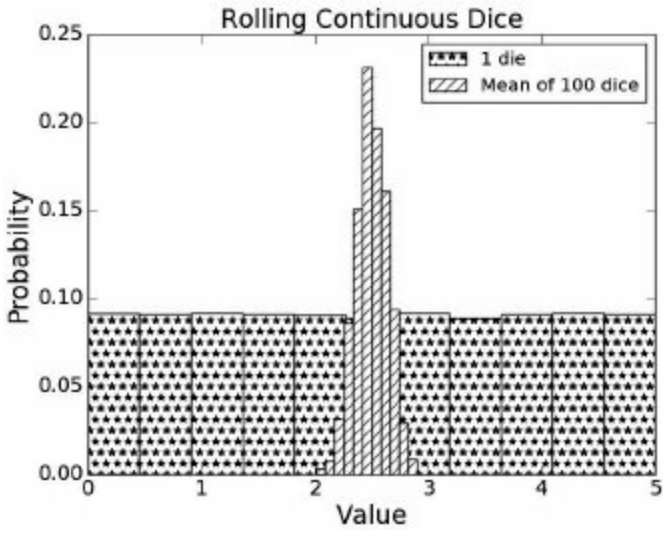

Figure 17.10: An illustration of the CLT

It's nice that the CLT seems to work, but what good is it? Perhaps it could prove useful in winning bar bets for those who drink in particularly nerdy bars. However, the primary value of the CLT is that it allows us to compute confidence levels and intervals even when the underlying population distribution is not normal. When we looked at confidence intervals in Section 15.4.2, we pointed out that the empirical rule is based on assumptions about the nature of the space being sampled. We assumed that

- The mean estimation error is 0, and
- The distribution of the errors in the estimates is normal.

When these assumptions hold, the empirical rule for normal distributions provides a handy way to estimate confidence intervals and levels given the mean and standard deviation.

Let's return to the Boston Marathon example. The code in Figure 17.11, which produced the plot in Figure 17.12, draws twenty simple random samples for each of a variety of sample sizes. For each sample size, it computes the mean of each of the twenty samples; it then computes the mean and standard deviation of those means. Since the CLT tells us that the sample means will be normally distributed, we can use the standard deviation and the empirical rule to compute a 95% confidence interval for each sample size.

As the plot in Figure 17.12 shows, all of the estimates are reasonably close to the actual population mean. Notice, however, that the error in the estimated mean does not decrease monotonically with the size of the samples—the estimate using 250 examples happens to be worse than the estimate using 50 examples. What does change monotonically with the sample size is our confidence in our estimate of the mean. As the sample size grows from 50 to 1,850, the confidence interval decreases from about ±15 to about ±2. This is important. It's not good enough to get lucky and happen to get a good estimate. We need to know how much confidence to have in our estimate.

```
times = getBMData('bm_results2012.txt')['time']
meanOfMeans, stdOfMeans = [], []
sampleSizes = range(50, 2000, 200)
for sampleSize in sampleSizes:
    sampleMeans = []
    for t in range(20):
        sample = random.sample(times, sampleSize)
        sampleMeans.append(sum(sample)/sampleSize)
    meanOfMeans.append(sum(sampleMeans)/len(sampleMeans))
    stdOfMeans.append(stdDev(sampleMeans))
pylab.errorbar(sampleSizes, meanOfMeans,
               yerr = 1.96*pylab.array(stdOfMeans),
               label = 'Estimated mean and 95% confidence interval')
pylab.xlim(0, max(sampleSizes) + 50)
pylab.axhline(sum(times)/len(times), linestyle = '--',
              label = 'Population mean')
pylab.title('Estimates of Mean Finishing Time')
pylab.xlabel('Sample Size')
pylab.ylabel('Finshing Time (minutes)')
pylab.legend(loc = 'best')
```

Figure 17.11 Produce plot with error bars

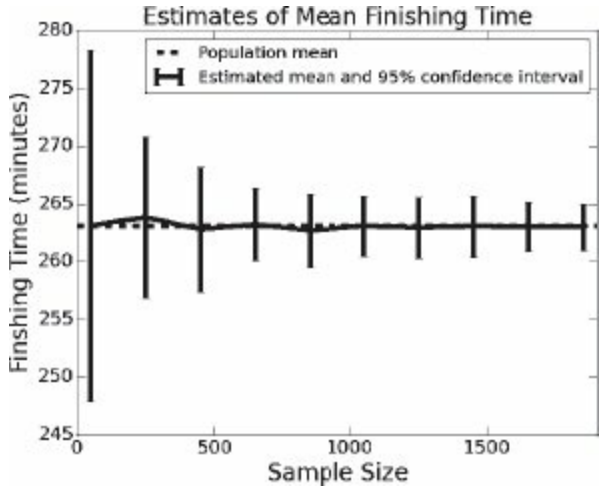

Figure 17.12: Estimates of finishing times with error bars

17.3 Standard Error of the Mean

We just saw that if we chose twenty random samples of 1,850 competitors, we could, with 95%

confidence, estimate the mean finishing time within a range of about four minutes. We did this using the standard deviation of the sample means. Unfortunately, since this involves using more total examples ($20*1{,}850 = 37{,}000$) than there were competitors, it doesn't seem like a very useful result. We would have been better off computing the actual mean directly using the entire population. What we need is a way to estimate a confidence interval using a single example. Enter the concept of the **standard error** of the mean (**SE** or **SEM**).

The SE for a sample of size n is the standard deviation of the means of an infinite number of samples of size n drawn from the same population. Unsurprisingly, it depends upon both n and σ, the standard deviation of the population:

$$SE = \frac{\sigma}{\sqrt{n}}$$

Figure 17.13 compares the SE for the sample sizes used in Figure 17.12 to the standard deviation of the means of the twenty samples we generated for each sample size.

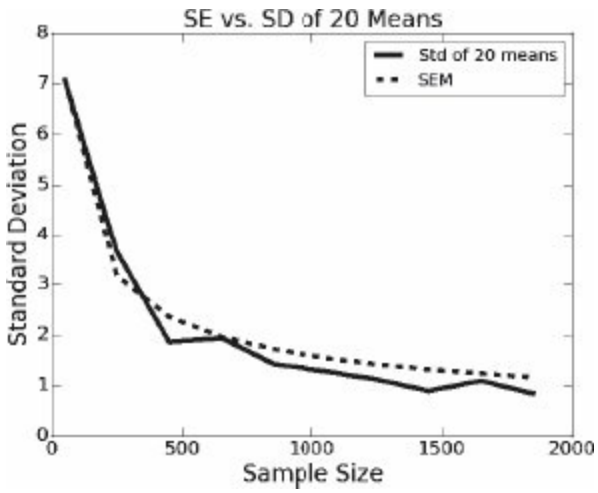

Figure 17.13: Standard error of the mean

The actual standard deviations of the means of our twenty samples closely tracks the SE. In both cases, the standard deviation drops rapidly at the start and then more slowly as the sample size gets large. The is because the reduction in standard deviation depends upon the square root of the sample size. E.g., to cut the standard deviation in half, one needs to quadruple the sample size.

Alas, if all we have is a single sample, we don't know the standard deviation of the population. Typically, we assume that the standard deviation of the sample, the sample standard deviation, is a reasonable proxy for the standard deviation of the population. This will be the case when the population distribution is not terribly skewed.

The code in Figure 17.14 creates 100 samples of various sizes from the Boston Marathon data, and compares the mean standard deviation of the samples of each size to the standard deviation of the population. It produces the plot in Figure 17.15.

```
times = getBMData('bm_results2012.txt')['time']
popStd = stdDev(times)
sampleSizes = range(2, 200, 2)
diffsMeans = []
for sampleSize in sampleSizes:
    diffs = []
    for t in range(100):
       diffs.append(abs(popStd - stdDev(random.sample(times,
                                                      sampleSize))))
    diffsMeans.append(sum(diffs)/len(diffs))
pylab.plot(sampleSizes, diffsMeans)
pylab.xlabel('Sample Size')
pylab.ylabel('Abs(Pop. Std - Sample Std)')
pylab.title('Sample SD vs Population SD')
```

Figure 17.14: Sample standard deviation vs. population standard deviation

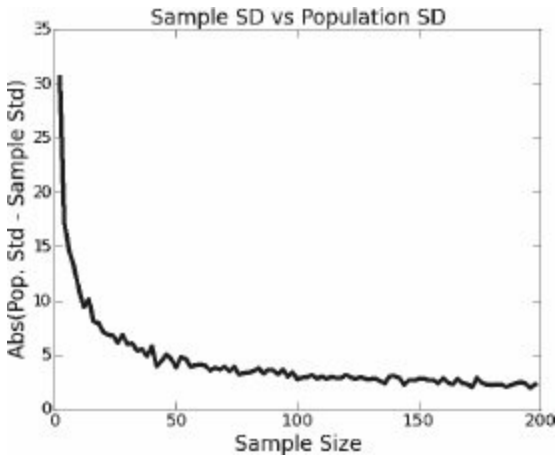

Figure 17.15: Sample standard deviations

By the time the sample size reaches 100, the difference between the sample standard deviation and the population standard deviation is relatively small.

In practice, people usually use the sample standard deviation in place of the (usually unknown) population standard deviation to estimate the SE. If the sample size is large enough,[119] and the population distribution is not too far from normal, it is safe to use this estimate to compute confidence intervals using the empirical rule.

What does this imply? If we take a single sample of say 200 runners, we can

- Compute the mean and standard deviation of that sample,
- Use the standard deviation of that sample to estimate the SE, and
- Use the estimated SE to generate confidence intervals around the sample mean.

The code in Figure 17.16 does this 10,000 times and then prints the fraction of times the sample

mean is more than 1.96 estimated SE's from the population mean. (Recall that for a normal distribution 95% of the data falls within 1.96 standard deviations of the mean.)

```
times = getBMData('bm_results2012.txt')['time']
popMean = sum(times)/len(times)
sampleSize = 200
numBad = 0
for t in range(10000):
    sample = random.sample(times, sampleSize)
    sampleMean = sum(sample)/sampleSize
    se = stdDev(sample)/sampleSize**0.5
    if abs(popMean - sampleMean) > 1.96*se:
        numBad += 1
print('Fraction outside 95% confidence interval =', numBad/10000)
```

Figure 17.16 Estimating the population mean 10,000 times

When the code is run it prints,

```
Fraction outside 95% confidence interval = 0.0533
```

I.e., pretty much what the theory predicts. Score one for the CLT!

---

[118] "Almost" because we rolled the die a finite number of times.

[119] Don't you just love following instructions with phrases like, "choose a large enough sample." Unfortunately, there is no simple recipe for choosing a sufficient sample size when you know little about the underlying population. Many statisticians say that a sample size of 30-40 is large enough when the population distribution is roughly normal. For smaller sample sizes, it is better to use something called the t-distribution to compute the size of the interval. The t-distribution is similar to a normal distribution, but it has fatter tails, so the confidence intervals will be a bit wider.

18 UNDERSTANDING EXPERIMENTAL DATA

This chapter is about understanding experimental data. We will make extensive use of plotting to visualize the data, and show how to use linear regression to build a model of experimental data. We will also talk about the interplay between physical and computational experiments. We defer our discussion of how to draw a valid statistical conclusion to Chapter 19.

18.1 The Behavior of Springs

Springs are wonderful things. When they are compressed or stretched by some force, they store energy. When that force is no longer applied they release the stored energy. This property allows them to smooth the ride in cars, help mattresses conform to our bodies, retract seat belts, and launch projectiles.

In 1676 the British physicist Robert Hooke formulated **Hooke's law** of elasticity: *Ut tensio, sic vis*, in English, F = -kx. In other words, the force F stored in a spring is linearly related to the distance the spring has been compressed (or stretched). (The minus sign indicates that the force exerted by the spring is in the opposite direction of the displacement.) Hooke's law holds for a wide variety of materials and systems, including many biological systems. Of course, it does not hold for an arbitrarily large force. All springs have an **elastic limit**, beyond which the law fails. Those of you who have stretched a Slinky too far know this all too well.

The constant of proportionality, k, is called the **spring constant**. If the spring is stiff (like the ones in the suspension of a car or the limbs of an archer's bow), k is large. If the spring is weak, like the spring in a ballpoint pen, k is small.

Knowing the spring constant of a particular spring can be a matter of some import. The calibrations of both simple scales and atomic force microscopes depend upon knowing the spring constants of components. The mechanical behavior of a strand of DNA is related to the force required to compress it. The force with which a bow launches an arrow is related to the spring constant of its limbs. And so on.

Generations of physics students have learned to estimate spring constants using an experimental apparatus similar to that pictured here.

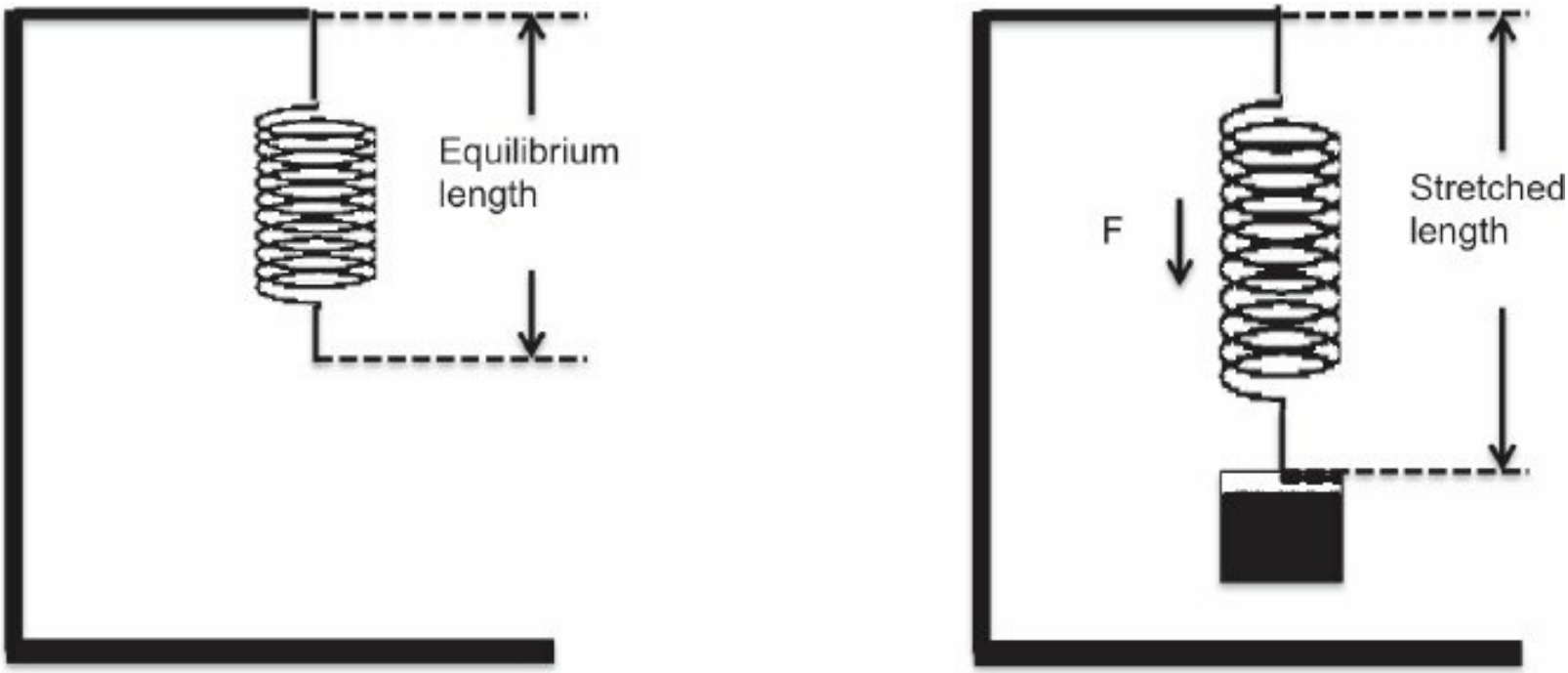

Figure 18.1 A classic experiment

We start with a spring with no weight on it, and measure the distance to the bottom of the spring from the top of the stand. We then hang a known mass on the spring, and wait for it to stop moving. At this point, the force stored in the spring is the force exerted on the spring by the weight hanging from it. This is the value of F in Hooke's law. We again measure the distance from the bottom of the spring to the top of the stand. The difference between this distance and the distance before we hung the weight then becomes the value of x in Hooke's law.

We know that the force, F, being exerted on the spring is equal to the mass, m, multiplied by the acceleration due to gravity, g (9.81 m/s^2 is a pretty good approximation of g on this planet), so we substitute m∗g for F. By simple algebra we know that k = -(m∗g)/x.

Suppose, for example, that m = 1kg and x = 0.1m, then $k = -\frac{1kg * 9.81m/s^2}{0.1m} = -\frac{9.81N}{0.1m} = -98.1N/m$

According to this calculation, it will take *98.1* Newtons[120] of force to stretch the spring one meter. This would all be well and good if

- We had complete confidence that we would conduct this experiment perfectly. In that case, we could take one measurement, perform the calculation, and know that we had found *k*. Unfortunately, experimental science hardly ever works this way.
- We could be sure that we were operating below the elastic limit of the spring.

A more robust experiment would be to hang a series of increasingly heavier weights on the spring, measure the stretch of the spring each time, and plot the results. We ran such an experiment, and typed the results into a file named `springData.txt`:

```
Distance (m) Mass (kg)
0.0865 0.1
0.1015 0.15
…
```

```
0.4416 0.9
0.4304 0.95
0.437 1.0
```

The function in Figure 18.2 reads data from a file such as the one we saved, and returns lists containing the distances and masses.

```
def getData(fileName):
    dataFile = open(fileName, 'r')
    distances = []
    masses = []
    dataFile.readline() #ignore header
    for line in dataFile:
        d, m = line.split(' ')
        distances.append(float(d))
        masses.append(float(m))
    dataFile.close()
    return (masses, distances)
```

Figure 18.2 Extracting the data from a file

The function in Figure 18.3 uses `getData` to extract the experimental data from the file and then produces the plot in Figure 18.4.

```
def plotData(inputFile):
    masses, distances = getData(inputFile)
    distances = pylab.array(distances)
    masses = pylab.array(masses)
    forces = masses*9.81
    pylab.plot(forces, distances, 'bo',
               label = 'Measured displacements')
    pylab.title('Measured Displacement of Spring')
    pylab.xlabel('|Force| (Newtons)')
    pylab.ylabel('Distance (meters)')

plotData('springData.txt')
```

Figure 18.3 Plotting the data

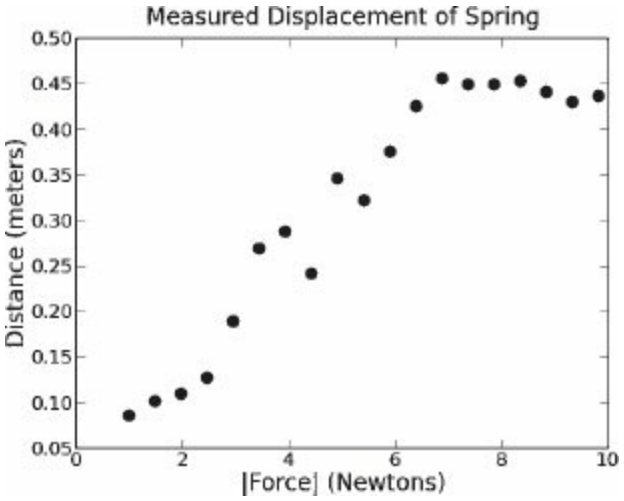

Figure 18.4 Displacement of spring

This is not what Hooke's law predicts. Hooke's law tells us that the distance should increase linearly with the mass, i.e., the points should lie on a straight line the slope of which is determined by the spring constant. Of course, we know that when we take real measurements the experimental data are rarely a perfect match for the theory. Measurement error is to be expected, so we should expect the points to lie around a line rather than on it.

Still, it would be nice to see a line that represents our best guess of where the points would have been if we had no measurement error. The usual way to do this is to fit a line to the data.

18.1.1 Using Linear Regression to Find a Fit

Whenever we fit any curve (including a line) to data we need some way to decide which curve is the best **fit** for the data. This means that we need to define an **objective function** that provides a quantitative assessment of how well the curve fits the data. Once we have such a function, finding the best fit can be formulated as finding a curve that minimizes (or maximizes) the value of that function, i.e., as an optimization problem (see Chapters 12 and 13).

The most commonly used objective function is called **least squares**. Let *observed* and *predicted* be vectors of equal length, where *observed* contains the measured points and *predicted* the corresponding data points on the proposed fit.

The objective function is then defined as:

$$\sum_{i=0}^{len(observed)-1} (observed[i] - predicted[i])^2$$

Squaring the difference between observed and predicted points makes large differences between observed and predicted points relatively more important than small differences. Squaring the difference also discards information about whether the difference is positive or negative.

How might we go about finding the best least-squares fit? One way to do this would be to use a successive approximation algorithm similar to the Newton-Raphson algorithm in Chapter 3. Alternatively, there is are analytic solutions that is often applicable. But we don't have to implement either, because PyLab provides a built-in function, `polyfit`, that finds an approximation to the best least-squares fit. The call

```
pylab.polyfit (observedXVals, observedYVals, n)
```

finds the coefficients of a polynomial of degree n that provides a best least-squares fit for the set of points defined by the two arrays `observedXVals` and `observedYVals`. For example, the call

```
pylab.polyfit(observedXVals, observedYVals, 1)
```

will find a line described by the polynomial y = ax + b, where a is the slope of the line and b the y-intercept. In this case, the call returns an array with two floating point values. Similarly, a parabola is described by the quadratic equation $y = ax^2 + bx + c$. Therefore, the call `pylab.polyfit(observedXVals, observedYVals, 2)` returns an array with three floating point values.

The algorithm used by `polyfit` is called **linear regression**. This may seem a bit confusing, since we can use it to fit curves other than lines. Some authors do make a distinction between linear regression (when the model is a line) and **polynomial regression** (when the model is a polynomial with degree greater than 1), but most do not.[121]

The function `fitData` in Figure 18.5 extends the `plotData` function in Figure 18.3 by adding a line that represents the best fit for the data. It uses `polyfit` to find the coefficients `a` and `b`, and then uses those coefficients to generate the predicted spring displacement for each force. Notice that there is an asymmetry in the way `forces` and `distance` are treated. The values in `forces` (which are derived from the mass suspended from the spring) are treated as independent, and used to produce the values in the dependent variable `predictedDistances` (a prediction of the displacements produced by suspending the mass).

The function also computes the spring constant, `k`. The slope of the line, `a`, is Δdistance/Δforce. The spring constant, on the other hand, is Δforce/Δdistance Consequently, `k` is the inverse of `a.`

The call `fitData('springData.txt')` produces the plot in Figure 18.6.

```
def fitData(inputFile):
    masses, distances = getData(inputFile)
    distances = pylab.array(distances)
    forces = pylab.array(masses)*9.81
    pylab.plot(forces, distances, 'ko',
               label = 'Measured displacements')
    pylab.title('Measured Displacement of Spring')
    pylab.xlabel('|Force| (Newtons)')
    pylab.ylabel('Distance (meters)')
    #find linear fit
    a,b = pylab.polyfit(forces, distances, 1)
    predictedDistances = a*pylab.array(forces) + b
    k = 1.0/a #see explanation in text
    pylab.plot(forces, predictedDistances,
              label = 'Displacements predicted by\nlinear fit, k = '
               + str(round(k, 5)))
    pylab.legend(loc = 'best')
```

Figure 18.5 Fitting a curve to data

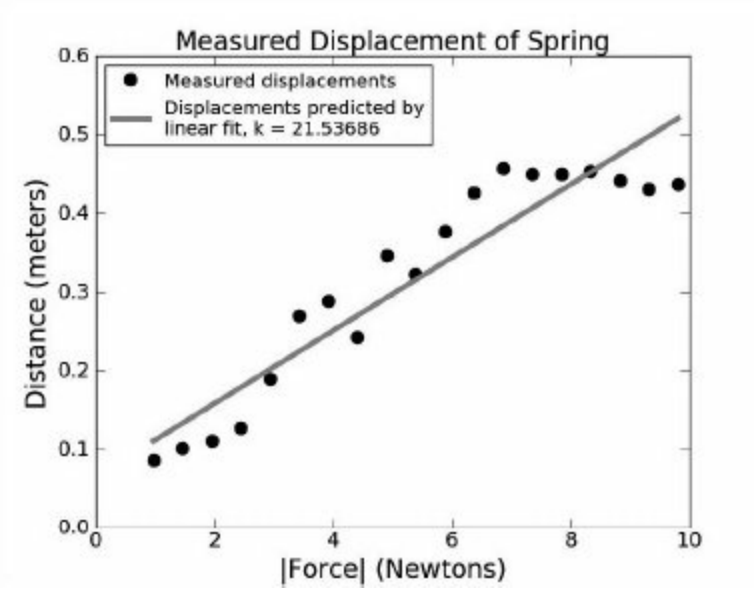

Figure 18.6 Measured points and linear model

It is interesting to observe that very few points actually lie on the least-squares fit. This is plausible because we are trying to minimize the sum of the squared errors, rather than maximize the number of points

that lie on the line. Still, it doesn't look like a great fit. Let's try a cubic fit by adding to `fitData` the code

```
#find cubic fit
fit = pylab.polyfit(forces, distances, 3)
predictedDistances = pylab.polyval(fit, forces)
pylab.plot(forces, predictedDistances, 'k:', label =
'cubic fit')
```

In this code, we have used the function `polyval` to generate the points associated with the cubic fit. This function takes two arguments: a sequence of polynomial coefficients and a sequence of values at which the polynomial is to be evaluated. The code fragments

```
fit = pylab.polyfit(forces, distances, 3)
predictedDistances = pylab.polyval(fit, forces)
```

and

```
a,b,c,d = pylab.polyfit(forces, distances, 3)
predictedDistances = a*(forces**3) + b*forces**2 +
c*forces + d
```

are equivalent.

This produces the plot in Figure 18.7. The cubic fit looks like a much better model of the data than the linear fit, but is it? Probably not.

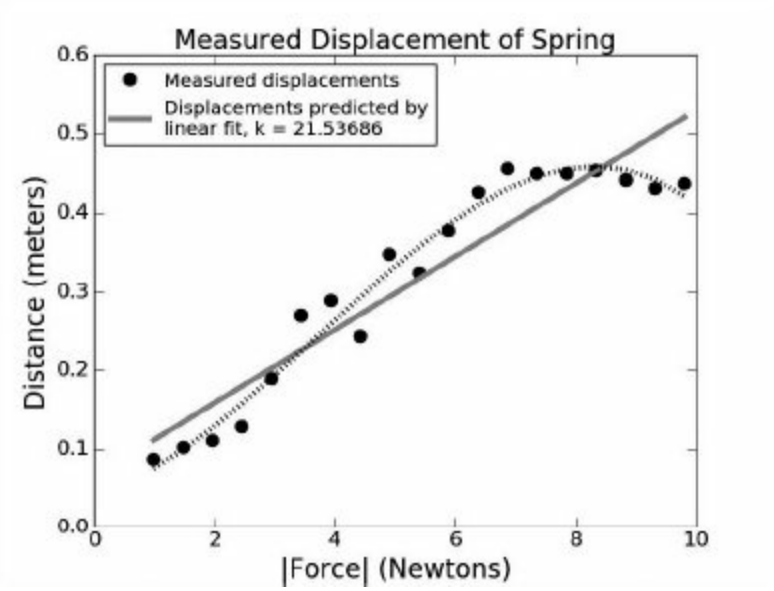

Figure 18.7 Linear and cubic fits

In the technical literature, one frequently sees plots like this that include both raw data and a curve fit to the data. All too often, however, the authors then go on to assume that the fitted curve is the description of the real situation, and the raw data merely an indication of experimental error. This can be dangerous.

Recall that we started with a theory that there should be a linear relationship between the x and y values, not a cubic one. Let's see what happens if we use our cubic fit to predict where the point corresponding to hanging a 1.5kg weight would lie, Figure 18.8.

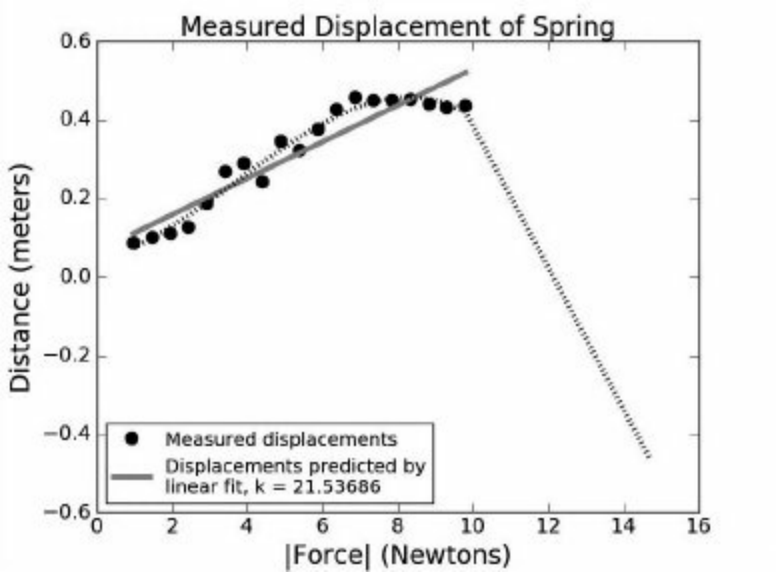

Figure 18.8 Using the model to make a prediction

Now the cubic fit doesn't look so good. In particular, it seems highly unlikely that by hanging a large weight on the spring we can cause the spring to rise above (the y value is negative) the bar from which it is suspended. What we have is an example of **overfitting**. Overfitting typically occurs when a model is excessively complex, e.g., it has too many parameters relative to the amount of data. When this happens, the fit can capture noise in the data rather than meaningful relationships. A model that has been overfit usually has poor predictive power, as seen in this example.

Finger exercise: Modify the code in Figure 18.5 so that it produces the plot in Figure 18.8.

Let's go back to the linear fit. For the moment, forget the line and study the raw data. Does anything about it seem odd? If we were to fit a line to the rightmost six points it would be nearly parallel to the x-axis. This seems to contradict Hooke's law—until we recall that Hooke's law holds only up to some elastic limit. Perhaps that limit is reached for this spring somewhere around 7N (approximately 0.7kg).

Let's see what happens if we eliminate the last six points by replacing

the second and third lines of `fitData` by

```
distances = pylab.array(distances[:-6])
masses = pylab.array(masses[:-6])
```

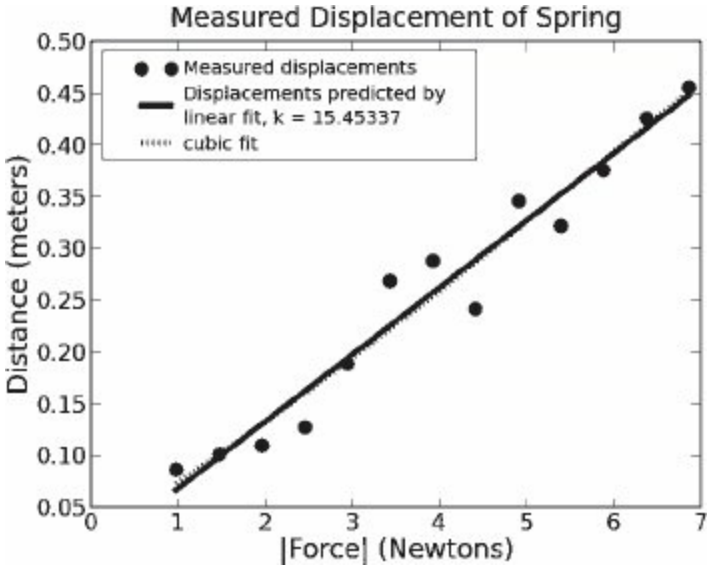

Figure 18.9 A model up to the elastic limit

As Figure 18.9 shows, eliminating those points certainly makes a difference: k has dropped dramatically and the linear and cubic fits are almost indistinguishable. But how do we know which of the two linear fits is a better representation of how our spring performs up to its elastic limit? We could use some statistical test to determine which line is a better fit for the data, but that would be beside the point. This is not a question that can be answered by statistics. After all we could throw out all the data except any two points and know that `polyfit` would find a line that would be a perfect fit for those two points. One should never throw out experimental results merely to get a better fit.[122] Here we justified throwing out the rightmost points by appealing to the theory underlying Hooke's law, i.e., that springs have an elastic limit. That justification could not have been appropriately used to eliminate points elsewhere in the data.

18.2 The Behavior of Projectiles

Growing bored with merely stretching springs, we decided to use one of our springs to build a device capable of launching a projectile.[123] We used the device four times to fire a projectile at a target 30 yards (1080 inches) from the launching point. Each time, we measured the height of the projectile at various distances from the launching point. The launching point and the target were at the same height, which we treated as `0.0` in our measurements.

The data was stored in a file with the contents shown in Figure 18.10. The first column contains distances of the projectile from the target. The other columns contain the height of the projectile at that distance for each of the four trials. All of the measurements are in inches.

Distance	trial1	trial2	trial3	trial4
1080	0.0	0.0	0.0	0.0
1044	2.25	3.25	4.5	6.5
1008	5.25	6.5	6.5	8.75
972	7.5	7.75	8.25	9.25
936	8.75	9.25	9.5	10.5
900	12.0	12.25	12.5	14.75
864	13.75	16.0	16.0	16.5
828	14.75	15.25	15.5	17.5
792	15.5	16.0	16.6	16.75
756	17.0	17.0	17.5	19.25
720	17.5	18.5	18.5	19.0
540	19.5	20.0	20.25	20.5
360	18.5	18.5	19.0	19.0
180	13.0	13.0	13.0	13.0
0	0.0	0.0	0.0	0.0

Figure 18.10 Data from projectile experiment

The code in Figure 18.11 was used to plot the mean altitude of the projectile in the four trials against the distance from the point of launch. It also plots the best linear and quadratic fits to those points. (In case you have forgotten the meaning of multiplying a list by an integer, the

expression `[0]*len(distances)` produces a list of `len(distances) 0`'s.)

```
def getTrajectoryData(fileName):
    dataFile = open(fileName, 'r')
    distances = []
    heights1, heights2, heights3, heights4 = [],[],[],[]
    dataFile.readline()
    for line in dataFile:
        d, h1, h2, h3, h4 = line.split()
        distances.append(float(d))
        heights1.append(float(h1))
        heights2.append(float(h2))
        heights3.append(float(h3))
        heights4.append(float(h4))
    dataFile.close()
    return (distances, [heights1, heights2, heights3, heights4])

def processTrajectories(fileName):
    distances, heights = getTrajectoryData(fileName)
    numTrials = len(heights)
    distances = pylab.array(distances)
    #Get array containing mean height at each distance
    totHeights = pylab.array([0]*len(distances))
    for h in heights:
        totHeights = totHeights + pylab.array(h)
    meanHeights = totHeights/len(heights)
    pylab.title('Trajectory of Projectile (Mean of '\
                + str(numTrials) + ' Trials)')
    pylab.xlabel('Inches from Launch Point')
    pylab.ylabel('Inches Above Launch Point')
    pylab.plot(distances, meanHeights, 'ko')
    fit = pylab.polyfit(distances, meanHeights, 1)
    altitudes = pylab.polyval(fit, distances)
    pylab.plot(distances, altitudes, 'b', label = 'Linear Fit')
    fit = pylab.polyfit(distances, meanHeights, 2)
    altitudes = pylab.polyval(fit, distances)
    pylab.plot(distances, altitudes, 'k:', label = 'Quadratic Fit')
    pylab.legend()

processTrajectories('launcherData.txt')
```

Figure 18.11 Plotting the trajectory of a projectile

A quick look at the plot in Figure 18.12 makes it quite clear that a quadratic fit is far better than a linear one.[124] (The reason that the quadratic fit is so bumpy-looking on the left side is that we are plotting only the predicted heights that correspond to the measured heights, and we have very few points to the left of 600.) But just how bad a fit is the line and how good is the quadratic fit?

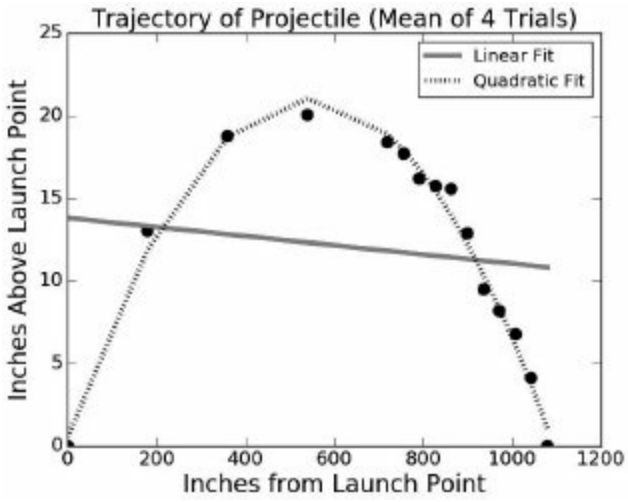

Figure 18.12 Plot of trajectory

18.2.1 Coefficient of Determination

When we fit a curve to a set of data, we are finding a function that relates an independent variable (inches horizontally from the launch point in this example) to a predicted value of a dependent variable (inches above the launch point in this example). Asking about the **goodness of a fit** is equivalent to asking about the accuracy of these predictions. Recall that the fits were found by minimizing the mean square error. This suggests that one could evaluate the goodness of a fit by looking at the mean square error. The problem with that approach is that while there is a lower bound for the mean square error (0), there is no upper bound. This means that while the mean square error is useful for comparing the relative goodness of two fits to the same data, it is not particularly useful for getting a sense of the absolute goodness of a fit.

We can calculate the absolute goodness of a fit using the **coefficient of determination**, often written as R^2.[125] Let y_i be the i^{th} observed value, p_i be the corresponding value predicted by the model, and μ be the mean of the observed values.

$$R^2 = 1 - \frac{\sum_i (y_i - p_i)^2}{\sum_i (y_i - \mu)^2}$$

By comparing the estimation errors (the numerator) with the variability of the original values (the denominator), R^2 is intended to capture the proportion of variability (relative to the mean) in a data set that is accounted for by the statistical model provided by the fit. When the model being evaluated is produced by a linear regression, the value of R^2 always lies between 0 and 1. If $R^2 = 1$, the model is a perfect fit to the data. If $R^2 = 0$, there is no relationship between the values predicted by the model and the way the data is distributed around the mean.

The code in Figure 18.13 provides a straightforward implementation

of this statistical measure. Its compactness stems from the expressiveness of the operations on arrays. The expression `(predicted - measured)**2` subtracts the elements of one array from the elements of another, and then squares each element in the result. The expression `(measured - meanOfMeasured)**2` subtracts the scalar value `meanOfMeasured` from each element of the array `measured`, and then squares each element of the results.

```
def rSquared(measured, predicted):
    """Assumes measured a one-dimensional array of measured values
               predicted a one-dimensional array of predicted values
       Returns coefficient of determination"""
    estimateError = ((predicted - measured)**2).sum()
    meanOfMeasured = measured.sum()/len(measured)
    variability = ((measured - meanOfMeasured)**2).sum()
    return 1 - estimateError/variability
```

Figure 18.13 Computing R^2

When the lines of code

```
print('RSquare of linear fit =',
rSquared(meanHeights, altitudes))
```

and

```
print('RSquare of quadratic fit =',
rSquared(meanHeights, altitudes))
```

are inserted after the appropriate calls to `pylab.plot` in `processTrajectories` (see Figure 18.11), they print

```
RSquared of linear fit = 0.0177433205441
RSquared of quadratic fit = 0.98576536928
```

Roughly speaking, this tells us that less than 2% of the variation in the measured data can be explained by the linear model, but more than 98% of the variation can be explained by the quadratic model.

18.2.2 Using a Computational Model

Now that we have what seems to be a good model of our data, we can use this model to help answer questions about our original data. One interesting question is the horizontal speed at which the projectile is traveling when it hits the target. We might use the following train of thought to design a computation that answers this question:

1. We know that the trajectory of the projectile is given by a formula of the form $y = ax^2 + bx + c$, i.e., it is a parabola. Since every parabola is symmetrical around its vertex, we know that its peak occurs halfway between the launch point and the target; call this distance xMid. The peak height, yPeak, is therefore given by $yPeak = a * xMid^2 + b * xMid + c$.
2. If we ignore air resistance (remember that no model is perfect), we can compute the amount of time it takes for the projectile to fall from yPeak to the height of the target, because that is purely a function of gravity. It is given by the equation $t = \sqrt{(2 * yPeak)/g}$.[126] This is also the amount of time it takes for the projectile to travel the horizontal distance from xMid to the target, because once it reaches the target it stops moving.
3. Given the time to go from xMid to the target, we can easily compute the average horizontal speed of the projectile over that interval. If we assume that the projectile was neither accelerating nor decelerating in the horizontal direction during that interval, we can use the average horizontal speed as an estimate of the horizontal speed when the projectile hits the target.

Figure 18.14 implements this technique for estimating the horizontal velocity of the projectile.[127]

```
def getHorizontalSpeed(quadFit, minX, maxX):
    """Assumes quadFit has coefficients of a quadratic polynomial
               minX and maxX are distances in inches
       Returns horizontal speed in feet per second"""
    inchesPerFoot = 12
    xMid = (maxX - minX)/2
    a,b,c = quadFit[0], quadFit[1], quadFit[2]
    yPeak = a*xMid**2 + b*xMid + c
    g = 32.16*inchesPerFoot #accel. of gravity in inches/sec/sec
    t = (2*yPeak/g)**0.5 #time in seconds from peak to target
    print('Horizontal speed =',
          int(xMid/(t*inchesPerFoot)), 'feet/sec')
```

Figure 18.14 Computing the horizontal speed of a projectile

When the line `getHorizontalSpeed(fit, distances[-1], distances[0])` is inserted at the end of `processTrajectories` (Figure 18.11), it prints

```
Horizontal speed = 136 feet/sec
```

The sequence of steps we have just worked through follows a common pattern.

1. We started by performing an experiment to get some data about the behavior of a physical system.
2. We then used computation to find and evaluate the quality of a model of the behavior of the system.
3. Finally, we used some theory and analysis to design a simple computation to derive an interesting consequence of the model.

18.3 Fitting Exponentially Distributed Data

`Polyfit` uses linear regression to find a polynomial of a given degree that is the best least-squares fit for some data. It works well if the data can be directly approximated by a polynomial. But this is not always

possible. Consider, for example, the simple exponential growth function $y = 3^x$. The code in Figure 18.15 fits a 5^{th}-degree polynomial to the first ten points and plots the results as shown in Figure 18.16. It uses the function call `pylab.arange(10)`, which returns an `array` containing the integers 0-9. The parameter setting `markeredgewidth = 2` sets the width of the lines used in the marker.

```
vals = []
for i in range(10):
    vals.append(3**i)
pylab.plot(vals,'ko', label = 'Actual points')
xVals = pylab.arange(10)
fit = pylab.polyfit(xVals, vals, 5)
yVals = pylab.polyval(fit, xVals)
pylab.plot(yVals, 'kx', label = 'Predicted points',
           markeredgewidth = 2, markersize = 25)
pylab.title('Fitting y = 3**x')
pylab.legend(loc = 'upper left')
```

Figure 18.15 Fitting a polynomial curve to an exponential distribution

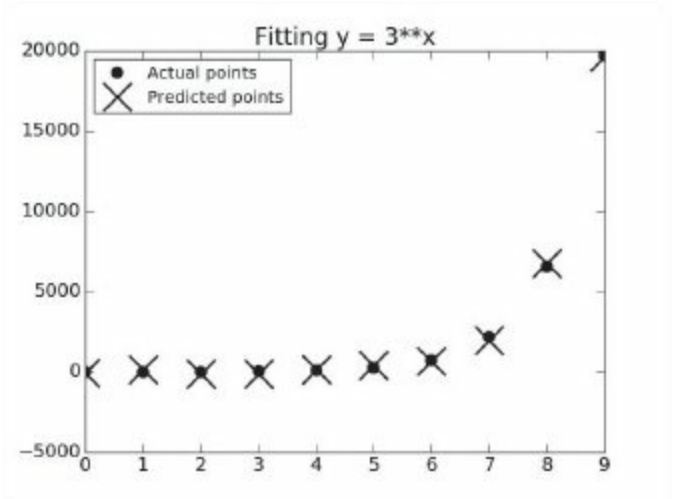

Figure 18.16 Fitting an exponential

The fit is clearly a good one, for these data points. However, let's look at what the model predicts for 3^{20}. When we add the code

```
print('Model predicts that 3**20 is roughly',
```

```
    pylab.polyval(fit, [3**20])[0])
print('Actual value of 3**20 is', 3**20)
```

to the end of Figure 18.15, it prints,

```
Model predicts that 3**20 is roughly
2.45478276372e+48
Actual value of 3**20 is 3486784401
```

Oh dear! Despite fitting the data, the model produced by `polyfit` is apparently not a good one. Is it because 5 was not the right degree? No. It is because no polynomial is a good fit for an exponential distribution. Does this mean that we cannot use `polyfit` to build a model of an exponential distribution? Fortunately, it does not, because we can use `polyfit` to find a curve that fits the original independent values and the log of the dependent values.

Consider the sequence [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]. If we take the log base 2 of each value. we get the sequence [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], i.e., a sequence that grows linearly. In fact, if a function y = f(x), exhibits exponential growth, the log (to any base) of f(x) grows linearly. This can be visualized by plotting an exponential function with a logarithmic y-axis. The code

```
xVals, yVals = [], []
for i in range(10):
    xVals.append(i)
    yVals.append(3**i)
pylab.plot(xVals, yVals, 'k')
pylab.semilogy()
```

produces the plot in Figure 18.17.

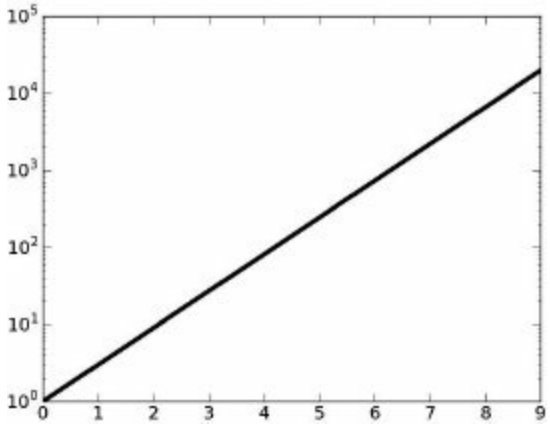

Figure 18.17 An exponential on a semilog plot

That taking the log of an exponential function produces a linear function can be used to construct a model for an exponentially distributed set of data points, as illustrated by the code in Figure 18.18. We use `polyfit` to find a curve that fits the `x` values and log of the `y` values. Notice that we use yet another Python standard library module, `math`, which supplies a `log` function. We also use a lambda expression, see Section 5.4.

When run, this code produces the plot in Figure 18.19, in which the actual values and the predicted values coincide. Moreover, when the model is tested on a value (20) that was not used to produce the fit, it prints

```
f(20) = 3486784401
Predicted value = 3486784401
```

```
import math

def createData(f, xVals):
    """Asssumes f is afunction of one argument
                xVals is an array of suitable arguments for f
       Returns array containing results of applying f to the
               elements of xVals"""
    yVals = []
    for i in xVals:
        yVals.append(f(xVals[i]))
    return pylab.array(yVals)

def fitExpData(xVals, yVals):
    """Assumes xVals and yVals arrays of numbers such that
         yVals[i] == f(xVals[i]), where f is an exponential function
       Returns a, b, base such that log(f(x), base) == ax + b"""
    logVals = []
    for y in yVals:
        logVals.append(math.log(y, 2.0)) #get log base 2
    fit = pylab.polyfit(xVals, logVals, 1)
    return fit, 2.0

xVals = range(10)
f = lambda x: 3**x
yVals = createData(f, xVals)
pylab.plot(xVals, yVals, 'ko', label = 'Actual values')
fit, base = fitExpData(xVals, yVals)
predictedYVals = []
for x in xVals:
    predictedYVals.append(base**pylab.polyval(fit, x))
pylab.plot(xVals, predictedYVals, label = 'Predicted values')
pylab.title('Fitting an Exponential Function')
pylab.legend(loc = 'upper left')
#Look at a value for x not in original data
print('f(20) =', f(20))
print('Predicted value =', int(base**(pylab.polyval(fit, [20]))))
```

Figure 18.18 Using `polyfit` to fit an exponential

This method of using `polyfit` to find a model for data works when

the relationship can be described by an equation of the form y = $base^{ax+b}$. If used on data that cannot be described this way, it will yield erroneous results.

To see this, let's create `yVals` using

```
f = lambda x: 3**x + x
```

The model now makes a poor prediction, printing

```
f(20) = 3486784421
Predicted value = 2734037145
```

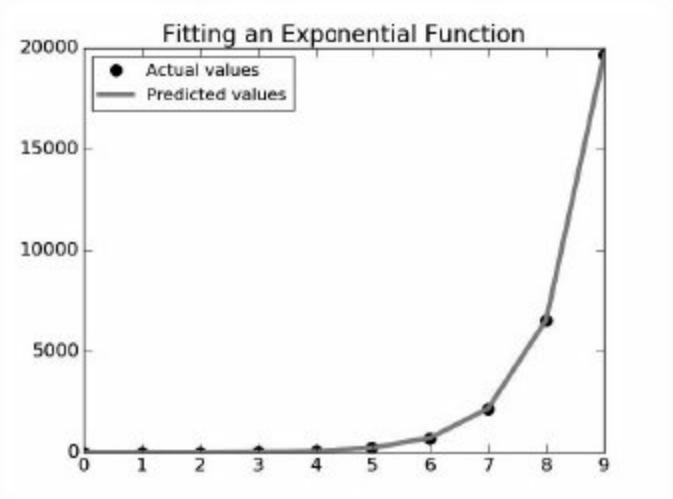

Figure 18.19 A fit for an exponential function

18.4 When Theory Is Missing

In this chapter, we have emphasized the interplay between theoretical, experimental, and computational science. Sometimes, however, we find ourselves with lots of interesting data, but little or no theory. In such cases, we often resort to using computational techniques to develop a theory by building a model that seems to fit the data.

In an ideal world, we would run a controlled experiment (e.g., hang weights from a spring), study the results, and retrospectively formulate a model consistent with those results. We would then run a new experiment (e.g., hang different weights from the same spring), and

compare the results of that experiment to what the model predicted.

Unfortunately, in many cases it is impossible to run even one controlled experiment. Imagine, for example, building a model designed to shed light on how interest rates affect stock prices. Very few of us are in a position to set interest rates and see what happens. On the other hand, there is no shortage of relevant historical data.

In such situations, one can simulate a set of experiments by dividing the existing data into a **training set** and a **holdout set** to use as a **test set**. Without looking at the holdout set, we build a model that seems to explain the training set. For example, we find a curve that has a reasonable R^2 for the training set. We then test that model on the holdout set. Most of the time the model will fit the training set more closely than it fits the holdout set. But if the model is a good one, it should fit the holdout set reasonably well. If it doesn't, the model should probably be discarded.

How does one choose the training set? We want it to be representative of the data set as a whole. One way to do this is to randomly choose the samples for the training set. If the data set is sufficiently large this often works pretty well.

A related but slightly different way to check a model is to train on many randomly selected subsets of the original data, and see how similar the models are to one another. If they are quite similar, then we can feel pretty good. This approach is known as **cross validation**.

Cross validation is discussed in more detail in Chapters 19 and 22.

---

[120] The Newton, written *N*, is the standard international unit for measuring force. It is the amount of force needed to accelerate a mass of one kilogram at a rate of one meter per second per second. A Slinky, by

the way, has a spring constant of approximately *1N/m*.

[121] The reason they do not is that although polynomial regression fits a nonlinear model to the data, the model is linear in the unknown parameters that it estimates.

[122] Which isn't to say that people never do.

[123] A projectile is an object that is propelled through space by the exertion of a force that stops after the projectile is launched. In the interest of public safety, we will not describe the launching device used in this experiment. Suffice it to say that it was awesome.

[124] Don't be misled by this plot into thinking that the projectile had a steep angle of ascent. It only looks that way because of the difference in scale between the vertical and horizontal axes on the plot.

[125] There are several different definitions of the coefficient of determination. The definition supplied here is used to evaluate the quality of a fit produced by a linear regression.

[126] This equation can be derived from first principles, but it is easier to just look it up. We found it at
`http://en.wikipedia.org/wiki/Equations_for_a_falling_b`

[127] The vertical component of the velocity is also easily estimated, since it is merely the product of the `g` and `t` in Figure 18.14.

19 RANDOMIZED TRIALS AND HYPOTHESIS CHECKING

Dr. X invented a drug, PED-X, designed to help professional bicycle racers ride faster. When he tried to market it, the racers insisted that Dr. X demonstrate that his drug was superior to PED-Y, the banned drug that they had been using for years. Dr. X raised money from some investors, and launched a **randomized trial**.

He persuaded 200 professional cyclists to participate in his trial. He then divided them randomly into two groups: treatment and control. Each member of the **treatment group** received a dose of PED-X. Members of the **control group** were told that they were being given a dose of PED-X, but were instead given a dose of PED-Y.

Each cyclist was asked to bike 50 miles as fast as possible. The finishing times for each group were normally distributed. The mean finishing time of the treatment group was 118.79 minutes, and that of the control group was 120.17 minutes. Figure 19.1 shows the time for each cyclist.

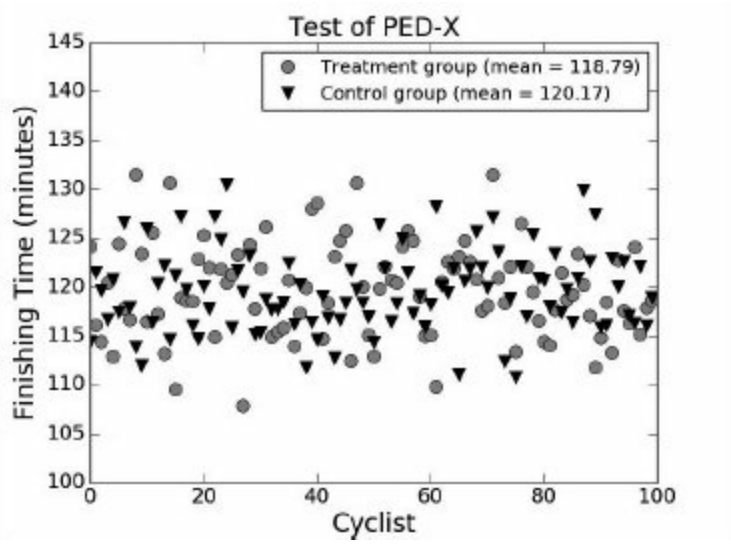

Figure 19.1: Finishing times for cyclists

Dr. X was elated until he ran into a statistician who pointed out that it was almost inevitable that one of the two groups would have a lower mean than the other, and perhaps the difference in means was merely a random occurrence. When she saw the crestfallen look on the scientist's face, the statistician offered to show him how to check the statistical significance of his study.

19.1 Checking Significance

In any experiment that involves drawing samples at random from a population, there is always the possibility that an observed effect occurred purely by chance. Figure 19.2 is a visualization of how temperatures in January of 2014 varied from the average temperatures in January from 1951 to 1980. Now, imagine that you constructed a sample by choosing twenty random spots on the planet, and then discovered that the mean change in temperature for the sample was +1 degree Celsius. What is the probability that the observed change in mean temperature was an artifact of the sites you happened to sample rather than an indication that the planet as a whole is warming? Answering this kind of question is what **statistical significance** is all about.

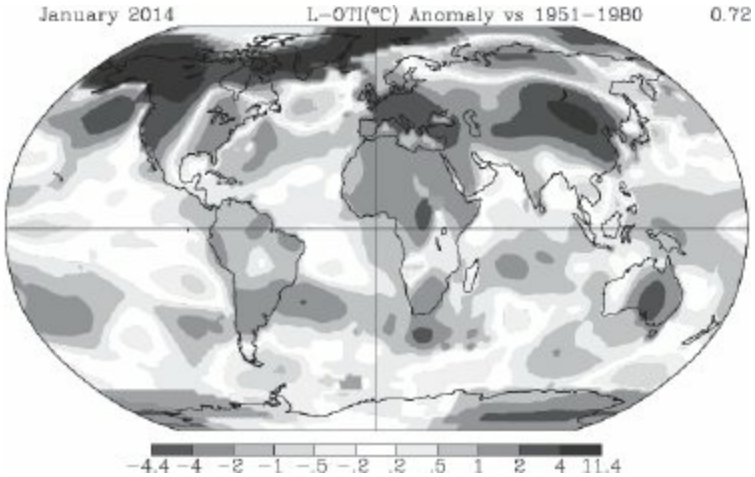

Figure 19.2 January temperature difference in degrees Celsius from the 1951-1980 average[128]

In the early part of the 20th century, Ronald Fisher developed an approach to statistical **hypothesis testing** that has become the most commonly used approach for evaluating the probability of an observed effect having occurred purely by chance. Fisher claims to have invented the method in response to a claim by Dr. Muriel Bristol-Roach that when she drank tea with milk in it she could detect whether the tea or the milk was poured into the teacup first. Fisher challenged her to a "tea test" in which she was given eight cups of tea (four for each order of adding tea and milk), and asked to identify those cups into which the tea had been poured before the milk. She did this perfectly. Fisher then calculated the probability of her having done this purely by chance. As we saw in Section 15.4.4, $\binom{8}{4} = 70$, i.e., there are 70 different ways to choose 4 cups out of 8. Since only one of these 70 combinations includes all 4 cups in which the tea was poured first, Fisher calculated that the probability of Dr. Bristol-Roach having chosen correctly by pure luck was $\frac{1}{70} \approx 0.014$. From this he concluded that it was highly unlikely that her success could be attributed to luck.

Fisher's approach to significance testing can be summarized as

1. State a **null hypothesis** and an **alternative hypothesis**. The null hypothesis is that the "treatment" has no interesting effect. For the "tea test," the null hypothesis was that Dr. Bristol-Roach had no ability to taste the difference. The alternative hypothesis is a hypothesis that can be true only if the null hypothesis is false, e.g., that Dr. Bristol-Roach could taste the difference.[129]
2. Understand statistical assumptions about the sample being evaluated. For the "tea test" Fisher assumed that Dr. Bristol-Roach was making independent decisions for each cup.
3. Compute a relevant test statistic. In this case, the test statistic was the fraction of correct answers

given by Dr. Bristol-Roach.

4. Derive the probability of that test statistic under the null hypothesis. In this case, the probability of getting all of the cups right by accident, i.e., 0.014.
5. Decide whether that probability is sufficiently small that you are willing to assume that the null hypothesis is false, i.e., to **reject** the null hypothesis. Common values for the rejection level, which should be chosen in advance, are 0.05 and 0.01.

Returning to our cyclists, imagine that the times for the treatment and control groups were samples drawn from infinite populations of finishing times for PED-X users and PED-Y users. The null hypothesis for this experiment is that the means of those two larger populations are the same, i.e., the difference between the population mean of the treatment group and the population mean of the control group is 0. The alternative hypothesis is that they are not the same, i.e., the difference in means is not equal to 0.

Next, we go about trying to reject the null hypothesis. We choose a threshold, α, for statistical significance, and try to show that the probability of the data having been drawn from distributions consistent with the null hypothesis is less than α. We then say that we can reject the null hypothesis with confidence α, and accept the negation of the null hypothesis with probability $1 - \alpha$.

The choice of α affects the kind of errors we make. The larger α, the more often we will reject a null hypothesis that is actually true. These are known as **type I errors**. When α is smaller, we will more often accept a null hypothesis that is actually false. These are known as **type II errors**.

Most commonly, people choose $\alpha = 0.05$. However, depending upon the consequences of being wrong, it might be preferable to choose a smaller or larger α. Imagine, for example, that the null hypothesis is that there is no difference in the rate of premature death between those taking PED-X and those taking PED-Y. One might well want to choose a small α, say 0.01, as the basis for rejecting that hypothesis before deciding whether or not one drug was safer than the other. On the other hand, if the null hypothesis were that there is no difference in the taste of PED-X and PED-Y, one might comfortably choose a pretty large α.[130]

The next step is to compute the test statistic. The most common test statistic is the **t-statistic**. The t-statistic tells us how different, measured in units of standard error, the estimate derived from the data is from the null hypothesis. The larger the t-statistic, the more likely the null hypothesis can be rejected. For our example, the t-statistic tells us how many standard errors the difference in the two means (118.79 – 120.17 = -1.38) is from 0. The t-statistic for our PED-X example is -2.13165598142. What does this mean? How do we use it?

We use the t-statistic in much the same way we use the number of standard deviations from the mean to compute confidence intervals (see Section 15.4.2). Recall that for all normal distributions the probability of an example lying within a fixed number of standard deviations of the mean is fixed. Here we do something slightly more complex that takes into account the number of samples used to compute the standard error. Instead of assuming a normal distribution, we assume a **t-distribution**.

T-distributions were first described, in 1908, by William Gosset, a statistician working for the Arthur Guinness and Son brewery.[131] The t-distribution is actually a family of distributions, since the shape of the distribution depends upon the degrees of freedom in the sample.

The **degrees of freedom** describes the amount of independent information used to derive the t-statistic. In general, we can think of degrees of freedom as the number of independent observations in

a sample that are available to estimate some statistic about the population from which that sample is drawn.

A t-distribution looks a lot like a normal distribution, and the larger the degrees of freedom the closer it is to a normal distribution. For small degrees of freedom, the t-distributions have notably fatter tails than normal distributions. For degrees of freedom of 30 or more, t-distributions are very close to normal.

Now, let's use the sample variance to to estimate the population variance. Recall that

$$variance(X) = \frac{\sum_{x \epsilon X}(x - \mu)^2}{|X|}$$

so the variance of our sample is

$$\frac{(100 - 200)^2 + (200 - 200)^2 + (300 - 200)^2}{3}$$

It might appear that we are using three independent pieces of information, but we are not. The three terms in the numerator are not independent of each other, because all three observations were used to compute the mean of the sample of 200 riders. The degrees of freedom is 2, since once we know the mean and any two of the three observations, the value of the third observation is fixed.

The larger the degrees of freedom, the higher the probability that the sample statistic is representative of the population. The degrees of freedom in a t-statistic computed from a single sample is one less than the sample size, because the mean of the sample is used in calculating the t-statistic. If two samples are used, the degrees of freedom is two less than the sum of the sample sizes, because the mean of each sample is used in calculating the t-statistic. For example, for the PED-X/PED-Y experiment, the degrees of freedom is 198.

Given the degrees of freedom, we can draw a plot showing the appropriate t-distribution, and then see where the t-statistic we have computed for our PED-X example lies on the distribution. The code in Figure 19.3 does that, and produces the plot in Figure 19.4. The code first uses the function `scipy.random.standard_t` to generate a large number of examples drawn from a t-distribution with 198 degrees of freedom. It then draws white lines at the t-statistic and the negative of the t-statistic for the PED-X sample.

```
tStat = -2.13165598142 #t-statistic for PED-X example
tDist = []
numBins = 1000
for i in range(10000000):
  tDist.append(scipy.random.standard_t(198))

pylab.hist(tDist, bins = numBins,
           weights = pylab.array(len(tDist)*[1.0])/len(tDist))
pylab.axvline(tStat, color = 'w')
pylab.axvline(-tStat, color = 'w')
pylab.title('T-distribution with 198 Degrees of Freedom')
pylab.xlabel('T-statistic')
pylab.ylabel('Probability')
```

Figure 19.3: Plotting a t-distribution

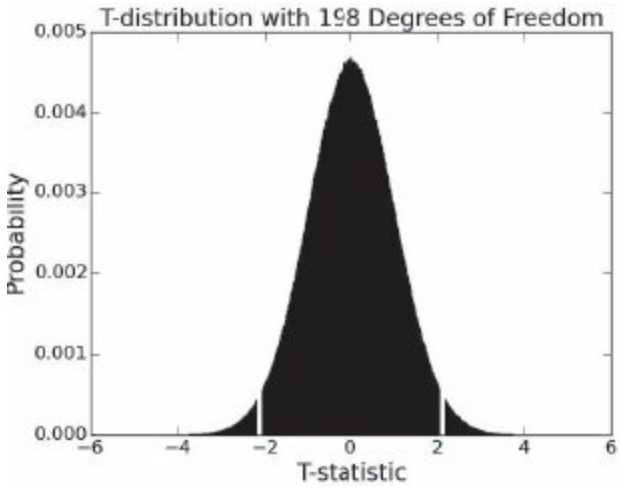

Figure 19.4: Visualizing the t-statistic

The sum of the fractions of the area of the histogram to the left and right of the white lines equals the probability of getting a value at least as extreme as the observed value if

- the sample is representative of the population, and
- the null hypothesis is true.

We need to look at both tails because our null hypothesis is that the population means are equal. So, the test should fail if the mean of the treatment group is either significantly larger or significantly smaller than the mean of the control group.

Under the assumption that the null hypothesis holds, the probability of getting a value at least as extreme as the observed value is called a **p-value**. For our PED-X example, the p-value is the probability of seeing a difference in the means at least as large as the observed difference, under the assumption that the actual population means of the treatment and controls are identical.

It may seem a bit odd that p-values tell us something about the probability of an event occurring if the null hypothesis holds, when what we are usually hoping is that the null hypothesis doesn't hold. However, it is not so different in character from the classic "scientific method," which is based upon

designing experiments that have the potential to refute a hypothesis. The code in Figure 19.5 computes and prints the t-statistic and p-value for our two samples. The library function `stats.ttest_ind` performs a two-tailed two-sample **t-test** and returns both the t-statistic and the p-value. Setting the parameter `equal_var` to `False` indicates that we don't know whether the two populations have the same variance.

```
controlMean = sum(controlTimes)/len(controlTimes)
treatmentMean = sum(treatmentTimes)/len(treatmentTimes)
print('Treatment mean - control mean =',
      treatmentMean - controlMean, 'minutes')
twoSampleTest = stats.ttest_ind(treatmentTimes, controlTimes,
                                equal_var = False)
print('The t-statistic from two-sample test is', twoSampleTest[0])
print('The p-value from two-sample test is', twoSampleTest[1])
```

Figure 19.5 Compute and print t-statistic and p-value

When we run the code, it reports

```
Treatment mean - control mean = -1.3766016405102306 minutes
The t-statistic from two-sample test is -2.13165598142
The p-value from two-sample test is 0.0343720799815
```

"Yes," Dr. X crowed, "it seems that the probability of PED-X being no better than PED-Y is less than 3.5%, and therefore the probability that PED-X has an effect is more than 96.5%. Let the cash registers start ringing." Alas, his elation lasted only until he read the next section of this chapter.

19.2 Beware of P-values

It is way too easy to read something into a p-value that it doesn't really imply. It is tempting to think of a p-value as the probability of the null hypothesis being true. But this is not what it actually means.

The null hypothesis is analogous to a defendant in the Anglo-American criminal justice system. That system is based on a principle called "presumption of innocence," i.e., innocent until proven guilty. Analogously, we assume that the null hypothesis is true unless we see enough evidence to the contrary. In a trial, a jury can rule that a defendant is "guilty" or "not guilty." A "not guilty" verdict implies that the evidence was insufficient to convince the jury that the defendant was guilty "beyond a reasonable doubt."[132] Think of it as equivalent to "guilt was not proven." A verdict of "not guilty" does not imply that the evidence was sufficient to convince the jury that the defendant was innocent. And it says nothing about what the jury would have concluded had it seen different evidence. Think of a p-value as a jury verdict where the standard "beyond a reasonable doubt" corresponds to choosing a very small α, and the evidence is the data from which the t-statistic was constructed.

A small p-value indicates that a particular sample is unlikely if the null hypothesis is true. It is

analogous to a jury concluding that it was unlikely that it would have been presented with this set of evidence if the defendant were innocent, and therefore reaching a guilty verdict. Of course, that doesn't mean that the defendant is actually guilty. Perhaps the jury was presented with misleading evidence. Analogously, a low p-value might be attributable to the null hypothesis actually being false, or it could simply be that the sample is unrepresentative of the population from which it is drawn, i.e., the evidence is misleading.

As you might expect, Dr. X staunchly claimed that his experiment showed that the null hypothesis was probably false. Dr. Y insisted that the low p-value was probably attributable to an unrepresentative sample, and funded another experiment of the same size as Dr. X's. When the statistics were computed using the samples from her experiment, the code printed

```
Treatment mean - control mean = 0.1760912816 minutes
The t-statistic from two-sample test is -0.274609731618
The p-value from two-sample test is 0.783908632676
```

This p-value is almost 24 times larger than that obtained from Dr. X's experiment, and since it is considerably larger than 0.5, provides no reason to doubt the null hypothesis. Confusion reigned. But we can clear it up!

You may not be surprised to discover that this is not a true story—after all, the idea of a cyclist taking a performance-enhancing drug strains credulity. In fact, the samples for the experiments were generated by the code in Figure 19.6.

```
treatmentDist = (119.5, 5.0)
controlDist = (120, 4.0)
sampleSize = 100
treatmentTimes, controlTimes = [], []
for s in range(sampleSize):
    treatmentTimes.append(random.gauss(treatmentDist[0],
                                       treatmentDist[1]))
    controlTimes.append(random.gauss(controlDist[0],
                                     controlDist[1]))
```

Figure 19.6 Code for generating racing examples

Since the experiment is purely computational, we can run it many times to get many different samples. When we generated 10,000 pairs of samples (one from each distribution) and plotted the probability of the p-values, we got the plot in Figure 19.7.

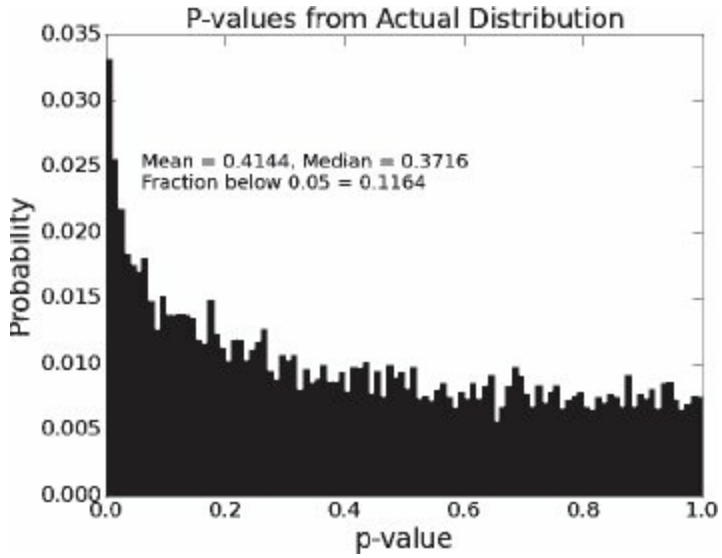

Figure 19.7: Probability of p-values

Since more than 11.6% of the p-values lie below 0.05, it is not terribly surprising that the first experiment we ran happened to show significance at the 5% level. On the other hand, that the second experiment yielded a completely different result is also not surprising. What does seem surprising is that given that we know that the means of the two distributions are actually different, we get a result that is significant at the 5% level only about 11.6% of the time. More than 88% of the time we would fail to reject a fallacious null hypothesis at the 5% level. (If we increase the sample size in our example to 2000, we fail to reject the fallacious null hypothesis only about 6% of the time.)

That p-values can be unreliable indicators of whether it is truly appropriate to reject a null hypothesis is one of the reasons that so many of the results appearing in the scientific literature cannot be reproduced by other scientists. One problem is that there is a strong relationship between the **study power** (the size of the samples) and the credibility of the statistical finding.[133]

Why are so many studies under-powered? If we were truly running an experiment with people (rather than a simulation) it would be twenty times more expensive to draw samples of size 2000 than samples of size 100.

The problem of sample size is an intrinsic attribute of what is called the frequentist approach to statistics. In Chapter 20, we discuss an alternative approach that attempts to mitigate this problem.

19.3 One-tail and One-sample Tests

Thus far in this chapter, we have looked only at two-tailed two-sample tests. There are times when it is more appropriate to use a **one-tailed** and/or a **one-sample** t-test.

Let's first consider a one-tailed two-sample test. In our two-tailed test of the relative effectiveness of PED-X and PED-Y, we considered three cases: 1) they were equally effective, 2) PED-X was more effective than PED-Y, and 3) PED-Y was more effective than PED-X. The goal was to reject the null hypothesis (case 1) by arguing that if it were true, it would be unlikely to see as large a difference as observed in the means of the PED-X and PED-Y samples.

Suppose, however, that PED-X were substantially less expensive than PED-Y. To find a market

for his compound, Dr. X would only need to show that PED-X is at least as effective as PED-Y. One way to think about this is that we want to reject the hypothesis that the means are equal or that the PED-X mean is larger. Note that this is strictly weaker than the hypothesis that the means are equal. (Hypothesis A is strictly weaker than hypothesis B, if whenever B is true A is true, but not vice versa.)

To do this, we start with the a two-sample test with the original null hypothesis computed by the code in Figure 19.5. It printed

```
Treatment mean - control mean = -1.37660164051 minutes
The t-statistic from two-sample test is -2.13165598142
The p-value from two-sample test is 0.0343720799815
```

allowing us to reject the null hypothesis at about the 3.5% level.

How about our weaker hypothesis? Recall Figure 19.4. We observed that under the assumption that the null hypothesis holds, the sum of the fractions of the areas of the histogram to the left and right of the white lines equals the probability of getting a value at least as extreme as the observed value. However, to reject our weaker hypothesis we don't need to take into account the area under the left tail, because that corresponds to PED-X being more effective than PED-Y (a negative time difference), and we're interested only in rejecting the hypothesis that PED-X is less effective. I.e., we can do a one-tailed test.

Since the t-distribution is symmetric, to get the value for a one-tailed test we divide the p-value from the two-tailed test in half. So the p-value for the one-tailed test is 0.01718603999075. This allows us to reject our weaker hypothesis at about the 1.7% level, something that we could not do using the two-tailed test.

Because a one-tailed test provides more power to detect an effect, it is tempting to use a one-tailed test whenever one has a hypothesis about the direction of an effect. This is usually not a good idea. A one-tailed test is appropriate only if the consequences of missing an effect in the untested direction are negligible.

Now let's look at a one-sample test. Suppose that, after years of experience of people using PED-Y, it was well established that the mean time for a racer on PED-Y to complete a fifty-mile course is 120 minutes. To discover whether or not PED-X had a different effect than PED-Y, we would test the null hypothesis that the mean time for a single PED-X sample is equal to 120. We can do this using the function `scipy.stats.ttest_1samp`, which takes as arguments a single sample and the population mean against which it is to be compared. It returns a tuple containing the t-statistic and p-value. For example, if we append to the end of the code in Figure 19.5 the code

```
oneSampleTest = stats.ttest_1samp(treatmentTimes, 120)
print('The t-statistic from one-sample test is', oneSampleTest[0])
print('The p-value from one-sample test is', oneSampleTest[1])
```

it prints

```
The t-statistic from one-sample test is -2.32665745939
The p-value from one-sample test is 0.0220215196873
```

It is not surprising that the p-value is smaller than the one we got using the two-sample two-tail test. By assuming that we know one of the two means, we have removed a source of uncertainty.

So, after all this, what have we learned from our statistical analysis of PED-X and PED-Y? Even though there is a difference in the expected performance of PED-X and and PED-Y users, no finite sample of PED-X and PED-Y users is guaranteed to reveal that difference. Moreover, because the difference in the expected means is small (less than half a percent), it is unlikely that an experiment of the size Dr. X ran (100 riders in each group) will yield evidence that would allow us to conclude at the 95% confidence level that there is a difference in means. We could increase the likelihood of getting a result that is statistically significant at the 95% level by using a one-tailed test, but that would be misleading, because we have no reason to assume that PED-X is not less effective than PED-Y.

19.4 Significant or Not?

Lyndsay and John have wasted an inordinate amount of time over the last several years playing a game called Words with Friends. They have played each other 1,273 times, and Lyndsay has won 666 of those games, prompting her to boast, “I’m way better at this game than you are.” John asserted that Lyndsay’s claim was nonsense, and that the difference in wins could be (and probably should be) attributed entirely to luck.

John, who had recently read a book about statistics, proposed the following way to find out whether it was reasonable to attribute Lyndsay’s relative success to skill:

- Treat each of the 1,273 games as an experiment returning 1 if Lyndsay was the victor and 0 if she was not.
- Choose the null hypothesis that the mean value of those experiments is 0.5.
- Perform a two-tailed one-sample test for that null hypothesis.

When he ran the code

```
numGames = 1273
lyndsayWins = 666
outcomes = [1.0]*lyndsayWins + [0.0]*(numGames-lyndsayWins)
print('The p-value from a one-sample test is',
     stats.ttest_1samp(outcomes, 0.5)[1])
```

it printed

```
The p-value from a one-sample test is 0.0982205871244
```

prompting John to claim that the difference wasn't even close to being significant at the 5% level.

Lyndsay, who had not studied statistics, but had read Chapter 16 of this book, was not satisfied. "Let's run a Monte Carlo simulation," she suggested, and supplied the code in Figure 19.8.

```
numGames = 1273
lyndsayWins = 666
numTrials = 10000
atLeast = 0
for t in range(numTrials):
    LWins = 0
    for g in range(numGames):
        if random.random() < 0.5:
            LWins += 1
    if LWins >= lyndsayWins:
        atLeast += 1
print('Probability of result at least this',
      'extreme by accident =', atLeast/numTrials)
```

Figure 19.8: Lyndsay's simulation of games

When Lyndsay's code was run it printed,

```
Probability of result at least this extreme by accident = 0.0491
```

prompting her to claim that John's statistical test was completely bogus and that the difference in wins was statistically significant at the 5% level.

"No," John explained patiently, "It's your simulation that's bogus. It assumed that you were the better player, and performed the equivalent of a one-tailed test. The inner loop of your simulation is wrong. You should have performed the equivalent of a two-tailed test by testing whether, in the simulation, either player won more than the 666 games that you won in actual competition." John then ran the simulation in Figure 19.9.

```
numGames = 1273
lyndsayWins = 666
numTrials = 10000
atLeast = 0
for t in range(numTrials):
    LWins, JWins = 0, 0
    for g in range(numGames):
        if random.random() < 0.5:
            LWins += 1
        else:
            JWins += 1
    if LWins >= lyndsayWins or JWins >= lyndsayWins:
        atLeast += 1
print('Probability of result at least this',
      'extreme by accident =', atLeast/numTrials)
```

Figure 19.9 Correct simulation of games

John's simulation printed

```
Probability of result at least this extreme by accident = 0.0986
```

"That's pretty darned close to what my two-tailed test predicted," crowed John. Lyndsay's unladylike response was not appropriate for inclusion in a family-oriented book.

19.5 Which N?

A professor wondered whether attending lectures was correlated with grades in his department. He recruited 40 freshmen and gave them all ankle bracelets so that he could track their whereabouts. Half of the students were not allowed to attend any of the lectures in any of their classes,[134] and half were required to attend all of the lectures.[135] Over the next four years, each student took 40 different classes, yielding 800 grades for each group of students.

When the professor performed a two-tailed t-test on the means of these two samples of size 800, the p-value was about 0.01. This disappointed the professor, who was hoping that there would be no statistically significant effect—so that he would feel less guilty about canceling lectures and going to the beach. In desperation, he took a look at the mean GPAs of the two groups, and discovered that there was very little difference. How, he wondered, could such a small difference in means be significant at that level?

When the sample size is large enough, even a small effect can be highly statistically significant. I.e., N matters, a lot. Figure 19.10 plots the mean p-value of 1000 trials against the size of the samples used in those trials. For each sample size and each trial we generated two samples. Each was was drawn from a Gaussian with a standard deviation of 5. One had a mean of 100 and the other a mean of 100.5. The mean p-value drops linearly with the sample size. The 0.5% difference in means becomes consistently statistically significant at the 5% level when the sample size reaches about 1500, and at the 1% level when the sample size exceeds about 2600.

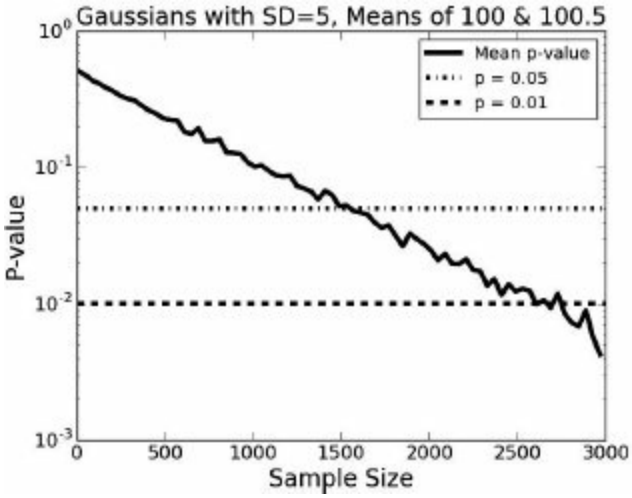

Figure 19.10 Impact of sample size on p-value

Returning to our example, was the professor justified in using an N of 800 for each **arm** of his study? To put it another way, were there really 800 independent examples for each cohort of 20 students? Probably not. There were 800 grades per sample, but only 20 students, and the 40 grades associated with each student should probably not be viewed as independent examples. After all, some students consistently get good grades, and some students consistently get grades that disappoint.

The professor decided to look at the data a different way. He computed the GPA for each student. When he performed a two-tailed t-test on these two samples, each of size 20, the p-value was about 0.3. He felt much better.

19.6 Multiple Hypotheses

In Chapter 17, we looked at sampling using data from the Boston Marathon. The code in Figure 19.11 reads in data from the 2012 race and looks for statistically significant differences in the mean finishing times of the women from a small set of countries. It uses the `getBMData` function defined in Figure 17.2.

```
data = getBMData('bm_results2012.txt')
countriesToCompare = ['BEL', 'BRA', 'FRA', 'JPN', 'ITA']
#Build mapping from country to list of female finishing times
countryTimes = {}
for i in range(len(data['name'])): #for each racer
    if data['country'][i] in countriesToCompare and\
       data['gender'][i] == 'F':
        try:
            countryTimes[data['country'][i]].append(data['time'][i])
        except KeyError:
            countryTimes[data['country'][i]] = [data['time'][i]]

#Compare finishing times of countries
for c1 in countriesToCompare:
    for c2 in countriesToCompare:
        if c1 < c2: # < rather than != so each pair examined once
            pVal = stats.ttest_ind(countryTimes[c1],
                                   countryTimes[c2],
                                   equal_var = False)[1]
            if pVal < 0.05:
                print(c1, 'and', c2,
                      'have significantly different means,',
                      'p-value =', round(pVal, 4))
```

Figure 19.11: Comparing mean finishing times for selected countries

When the code is run, it prints

```
ITA and JPN have significantly different means, p-value = 0.025
```

It looks as if either Italy or Japan can claim to have faster women runners than the other.[136] However, such a conclusion would be pretty tenuous. While one set of runners did have a faster mean time than the other, the sample sizes (20 and 32) were small and perhaps not representative of the capabilities of women marathoners in each country.

More important, there is a flaw in the way we constructed our experiment. We checked 10 null hypotheses (one for each distinct pair of countries), and discovered that one of them could be rejected at the 5% level. One way to think about it is that we were actually checking the null hypothesis: "for all pairs of countries, the mean finishing times of their female marathon runners are the same." It might be fine to reject that null hypothesis, but that is not the same as rejecting the null hypothesis that women marathon runners from Italy and Japan are equally fast.

The point is made starkly by the example in Figure 19.12. In that example, we draw twenty pairs of samples of size 200 from the same population, and for each we test whether the means of the

samples are statistically different.

```
numHyps = 20
sampleSize = 30
population = []
for i in range(5000): #Create large population
    population.append(random.gauss(0, 1))
sample1s, sample2s = [], []
for i in range(numHyps): #Generate many pairs of small sanples
    sample1s.append(random.sample(population, sampleSize))
    sample2s.append(random.sample(population, sampleSize))
#Check pairs for statistically significant difference
numSig = 0
for i in range(numHyps):
    if scipy.stats.ttest_ind(sample1s[i], sample2s[i])[1] < 0.05:
        numSig += 1
print('Number of statistically significant (p < 0.05) results  =',
      numSig)
```

Figure 19.12: Checking multiple hypotheses

Since the samples are all drawn from the same population, we know that the null hypothesis is true. Yet, when we run the code it prints

```
Number of statistically significant (p < 0.05) results = 1
```

indicating that the null hypothesis can be rejected for one pair.

This is not particularly surprising. Recall that a p-value of 0.05 indicates that if the null hypothesis holds, the probability of seeing a difference in means at least as large as the difference for the two samples is 0.05. Therefore, it is not terribly surprising that if we examine twenty pairs of samples, at least one of them has means that are statistically significantly different from each other. Running large sets of related experiments, and then cherry-picking the result you like, can be kindly described as sloppy. An unkind person might call it something else.

Returning to our Boston Marathon experiment, we checked whether or not we could reject the null hypothesis (no difference in means) for 10 pairs of samples. When running an experiment involving multiple hypotheses, the simplest and most conservative approach is to use something called the **Bonferroni correction**. The intuition behind it is simple: when checking a family of m hypotheses, one way of maintaining an appropriate **family-wise error rate** is to test each individual hypothesis at a level of $\frac{1}{m} * \alpha$. Using the Bonferroni correction to see if the difference between Italy and Japan is significant at the $\alpha = 0.05$ level, we should check if the p-value is less that 0.05/10 i.e., 0.005—which it is not.

The Bonferroni correction is conservative (i.e., it fails to reject the null hypothesis more often than necessary) if there are a large number of tests or the test statistics for the tests are positively

correlated. An additional issue is the absence of a generally accepted definition of "family of hypotheses." It is obvious that the hypotheses generated by the code in Figure 19.12 are related, and therefore a correction needs to be applied. But the situation is not always so clear cut.

---

[128] This is a gray-scale version of a color image provided by the U.S. National Aeronautics and Space Administration.

[129] In his formulation, Fisher had only a null hypothesis. The idea of an alternative hypothesis was introduced later by Jerzy Neyman and Egon Pearson.

[130] Many researchers, including the author of this book, believe strongly that the "rejectionist" approach to reporting statistics is unfortunate. It is almost always preferable to report the actual significance level rather than merely stating that "the null hypothesis has been rejected at the 5% level."

[131] Guiness forbade Gosset from publishing under his own name. He used the pseudonym "Student" when he published his seminal 1908 paper, "Probable Error of a Mean," about t-distributions. As a result, the distribution is frequently called "Student's t-distribution."

[132] The "beyond a reasonable doubt" standard implies that society believes that in the case of a criminal trial, type I errors (convicting an innocent person) are much less desirable than type II errors (acquitting a guilty person). In civil cases, the standard is "the preponderance of the evidence," suggesting that society believes that the two kinds of errors are equally undesirable.

[133] Katherine S. Button, John P. A. Ioannidis, Claire Mokrysz, Brian A. Nosek, Jonathan Flint, Emma S. J. Robinson, and Marcus R. Munafò (2013) "Power failure: why small sample size undermines the reliability of neuroscience," *Nature Reviews Neuroscience*, 14: 365-376.

[134] They should have been given a tuition rebate, but weren't.

[135] They should have been given combat pay, but weren't.

[136] We could easily find out which by looking at the sign of the t-statistic, but in the interest of not offending potential purchasers of this book, we won't.

20 CONDITIONAL PROBABILITY AND BAYESIAN STATISTICS

Up to this point we have taken what is called a **frequentist** approach to statistics. We have drawn conclusions from samples based entirely on the frequency or proportion of the data. This is the most commonly used inference framework, and leads to the well-established methodologies of statistical hypothesis testing and confidence intervals covered earlier in this book. In principle, it has the advantage of being unbiased. Conclusions are reached solely on the basis of the observed data.

In some situations, however, an alternative approach to statistics, **Bayesian statistics**, is more appropriate. Consider the cartoon in Figure 20.1.[137]

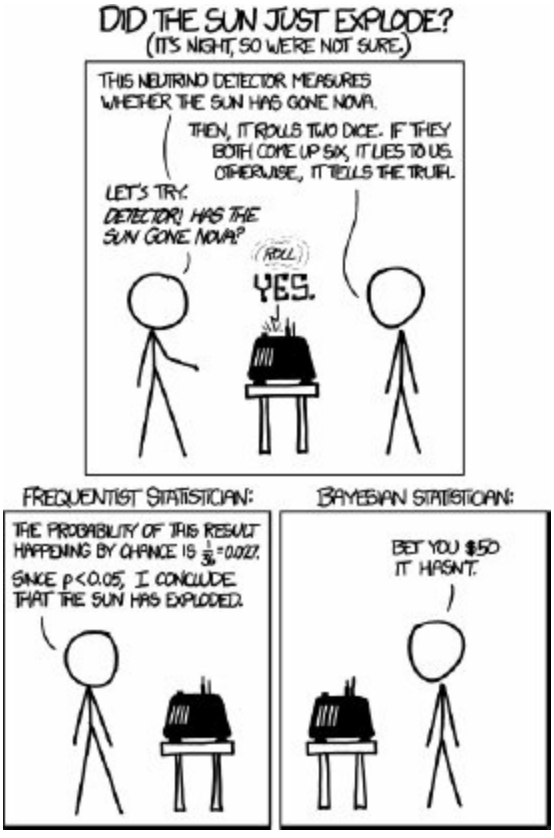

Figure 20.1 Has the sun exploded?

What's going on here? The frequentist knows that there are only two possibilities: the machine rolls a pair of sixes and is lying, or it doesn't roll a pair of sixes and is telling the truth. Since the probability of not rolling a pair of sixes is $35/36$ (97.22%) the frequentist concludes that the

machine is probably telling the truth, and therefore the sun has probably exploded.[138]

The Bayesian utilizes additional information in building his probability model. He agrees that it is unlikely that the machine rolls a pair of sixes, however he argues that the probability of that happening needs to be compared to the *a priori* probability that the sun has not exploded. He concludes that the likelihood of the sun having not exploded is even higher than 97.22%, and decides to bet that "the sun will come out tomorrow."

20.1 Conditional Probabilities

The key idea underlying Bayesian reasoning is **conditional probability**.

In our earlier discussion of probability, we relied on the assumption that events were independent. For example, we assumed that whether a coin flip came up heads or tails was unrelated to whether the previous flip came up heads or tails. This is convenient mathematically, but life doesn't always work that way. In many practical situations, independence is a bad assumption.

Consider the probability that a randomly chosen adult American is male and weighs over 180 pounds. The probability of being male is about 0.5 and the probability of weighing more than 180 pounds (the average weight in the U.S.[139]) is also about 0.5.[140] So, if these were independent events the probability of the selected person being both male and weighing more than 180 pounds would be 0.25. However, these events are not independent, since the average American male weighs about 30 pounds more than the average female. So, a better question to ask is 1) what is the probability of the selected person being a male, and 2) given that the selected person is a male, what is the probability of that person weighing more than 180 pounds. The notation of conditional probability makes it easy to say just that.

The notation P(A|B) stands for the probability of A being true under the assumption that B is true. It is often read as "the probability of A, given B." Therefore, the formula

$$P\,(male)*P(weight > 180 \mid male)$$

expresses exactly the probability we are looking for. If P(A) and P(B) are independent, P(A|B) = P(A). For the above example, B is male and A is weight > 180.

In general, if $P(B) \neq 0$,

$$P(A|B) = \frac{P(A \text{ and } B)}{P(B)}$$

Like conventional probabilities, conditional probabilities always lie between 0 and 1. Furthermore, if $\bar{A}$ stands for *not* A, $P(A|B) + P(\bar{A}|B) = 1$. People often incorrectly assume that $P(A|B)$ is equal to $P(B|A)$. There is no reason to expect this to be true. For example, the value of $P(male|Maltese)$ is roughly 0.5, but $P(Maltese|male)$ is about 0.000064.[141]

Finger exercise: Estimate the probability that a randomly chosen American is both male and weighs more than 180 pounds. Assume that 50% of the population is male, and that the weights of the male

population are normally distributed with a mean of 210 pounds and a standard deviation of 30 pounds. (Hint: think about using the empirical rule.)

The formula $P(A|B,C)$ stands for the probability of A, given that both B and C hold. Assuming that B and C are independent of each other, the definition of a conditional probability and the multiplication rule for independent probabilities imply that

$$P(A|\,B,C) = \frac{P(A,B,C)}{P(B,C)}$$

where the formula $P(A,B,C)$ stands for the probability of all of A, B, and C being true.

Similarly, $P(A,B|C)$ stands for the probability of A and B, given C. Assuming that A and B are independent of each other

$$P(A,B|C) = P(A|C)*P(B|C)$$

20.2 Bayes' Theorem

Suppose that an asymptomatic woman in her forties goes for a mammogram and receives bad news: the mammogram is "positive."[142]

The probability that a woman who has breast cancer will get a **true positive** result on a mammogram is 0.9. The probability that a woman who does not have breast cancer will get a **false positive** on a mammogram is 0.07.

We can use conditional probabilities to express these facts. Let

```
Canc = has breast cancer
TP = true positive
FP = false positive
```

Using these variables, we write the conditional probabilities

```
P(TP | Canc) = 0.9
P(FP | not Canc) = 0.07
```

Given these conditional probabilities, how worried should a woman in her forties with a positive mammogram be? What is the probability that she actually has breast cancer? Is it 0.93, since the false positive rate is 7%? More? less?

It's a trick question: We haven't supplied enough information to allow you to answer the question in a sensible way. To do that, you need to know the **prior probabilities** for breast cancer for a woman in her forties. The fraction of women in their forties who have breast cancer is 0.008 (8 out of 1000). The fraction who do not have breast cancer is therefore 1 – 0.008 = 0.992. I.e.,

```
P(Canc | woman in her 40s) = 0.008
P(not Canc | woman in her 40s) = 0.992
```

We now have all the information we need to address the question of how worried that woman in her forties should be. To compute the probability that she has breast cancer we use something called **Bayes' Theorem**[143] (often called Bayes' Law or Bayes' Rule) :

$$P(A|B) = \frac{P(A) * P(B|A)}{P(B)}$$

In the Bayesian world, probability measures a **degree of belief**. Bayes' theorem links the degree of belief in a proposition before and after accounting for evidence. The formula to the left of the equal sign, P(A|B), is the **posterior** probability, the degree of belief in A, having accounted for B. The posterior is defined in terms of the **prior**, P(A), and the **support** that the evidence, B, provides for A. The support is the ratio of the probability of B holding if A holds and the probability of B holding independently of A, i.e., $\frac{P(B|A)}{P(B)}$.

If we use Bayes' Theorem to estimate the probability of the woman actually having breast cancer we get (where `Canc` plays the role of `A`, and `Pos` the role of `B` in our statement of Bayes' Theorem)

$$P(Canc|Pos) = \frac{P(Canc) * P(Pos|Canc)}{P(Pos)}$$

The probability of having a positive test is

$$P(Pos) = P(Pos|Canc) * P(Canc) + P(Pos|\ not\ Canc) * (1 - P(Canc))$$

so

$$P(Canc|Pos) = \frac{0.008 * 0.9}{0.9 * 0.008 + 0.07 * 0.992} = \frac{0.0072}{0.07664} \approx 0.094$$

I.e., approximately 90% of the positive mammograms are false positives![144] Bayes' Theorem helped us here because we had an accurate estimate of the prior probability of a woman in her forties having breast cancer.

It is important to keep in mind that if we had started with an incorrect prior, incorporating that prior into our probability estimate would make the estimate worse rather than better. For example, if we had started with the prior

```
P(Canc | women in her 40's) = 0.6
```

we would have concluded that the false positive rate was about 5%, i.e., that the probability of a woman in her forties with a positive mammogram having breast cancer is roughly 0.95.

Finger exercise: You are wandering through a forest and see a field of delicious-looking mushrooms. You fill your basket with them, and head home prepared to cook them up and serve them to your husband. Before you cook them, however, he demands that you consult a book about local mushroom species to check whether they are poisonous. The book says that 80% of the mushrooms in the local forest are poisonous. However, you compare your mushrooms to the ones pictured in the book, and decide that you are 95% certain that your mushrooms are safe. How comfortable you should you be about serving them to your husband (assuming that you would rather not become a widow)?

20.3 Bayesian Updating

Bayesian inference provides a principled way of combining new evidence with prior beliefs, through the application of Bayes' theorem. Bayes' theorem can be applied iteratively: After observing some evidence, the resulting posterior probability can then be treated as a prior probability, and a new posterior probability computed from new evidence. This allows for Bayesian principles to be applied to various kinds of evidence, whether viewed all at once or over time. This procedure is termed **Bayesian updating**.

Let's look at an example. Assume that you have a bag containing equal numbers of three different kinds of dice—each with a different probability of coming up 6 when rolled. Type A dice come up 6 with a probability of one fifth, type B dice with a probability of one sixth, and type C dice with a probability of one seventh. Reach into the bag, grab one die, and estimate the probability of it being of type A. You don't need to know much probability to know that the best estimate is 1/3. Now roll the die twice and revise the estimate based on the outcome of the rolls. If it comes up 6 both times, it seems clear that it is somewhat more likely that the die is of type A. How much more likely? We can use Bayesian updating to answer that question.

By Bayes' theorem, after rolling the first 6, the probability that the die is of type A is

$$P(A|6) = \frac{P(A) * P(6|A)}{P(6)}$$

where

$$P(A) = \frac{1}{3},\ P(6|A) = \frac{1}{5},\ P(6) = \frac{\frac{1}{5}+\frac{1}{6}+\frac{1}{7}}{3}$$

The code in Figure 20.2 implements Bayes' theorem and uses it to calculate the probability that the die is of type A. Notice that the second call of `calcBayes` uses the result of the first call as the prior value for A.

```
def calcBayes(priorA, probBifA, probB):
    """priorA: initial estimate of probability of A independent of B
       priorBifA: est. of probability of B assuming A is true
       priorBifNotA: est. of probability of B
       returns probability of A given B"""
    return priorA*probBifA/probB

priorA = 1/3
prob6ifA = 1/5
prob6 = (1/5 + 1/6 + 1/7)/3

postA = calcBayes(priorA, prob6ifA, prob6)
print('Probability of type A =', round(postA, 4))
postA = calcBayes(postA, prob6ifA, prob6)
print('Probability of type A =', round(postA, 4))
```

Figure 20.2: Bayesian updating

When the code is run, it prints,

```
Probability of type A = 0.3925
Probability of type A = 0.4622
```

indicating that we should revise our estimate of the probability upwards.

What if we had thrown something other than 6 on both rolls of the die? Replacing the last four lines of the code in Figure 20.2 with

```
postA = calcBayes(priorA, 1 - prob6ifA, 1 - prob6)
print('Probability of type A =', round(postA, 4))
postA = calcBayes(postA, 1 - prob6ifA, 1 - prob6)
print('Probability of type A =', round(postA, 4))
```

causes the program to print

```
Probability of type A = 0.3212
Probability of type A = 0.3096
```

indicating that we should revise our estimate of the probability downwards.

Let's suppose that we have reason to believe that 90% of the dice in the bag are of type A. All we need to do is change our original prior, `priorA` in the code, to 0.9. Now, if we simulate getting something other than 6 on both rolls, it prints

```
Probability of type A = 0.8673
Probability of type A = 0.8358
```

The prior makes a big difference!

Let's try one more experiment. We'll stick with `priorA = 0.9`, and see what happens if the die is actually of type C. The code in Figure 20.3 simulates 200 rolls of a die of type C (which has a probability of 1/7 of coming up 6), and prints a revised estimate of the probability of the die being of type A after every 20 rolls.

```
numRolls = 200
postA = priorA
for i in range(numRolls+1):
    if i%(numRolls//10) == 0:
        print('After', i, 'rolls. Probability of type A =',
              round(postA, 4))
    isSix = random.random() <= 1/7 #because die of type C
    if isSix:
        postA = calcBayes(postA, prob6ifA, prob6)
    else:
        postA = calcBayes(postA, 1 - prob6ifA, 1 - prob6)
```

Figure 20.3: Bayesian updating with a bad prior

When we ran the code it printed

```
After 0 rolls. Probability of type A = 0.9
After 20 rolls. Probability of type A = 0.4294
After 40 rolls. Probability of type A = 0.3059
After 60 rolls. Probability of type A = 0.2662
After 80 rolls. Probability of type A = 0.1552
After 100 rolls. Probability of type A = 0.0905
After 120 rolls. Probability of type A = 0.0962
After 140 rolls. Probability of type A = 0.1251
After 160 rolls. Probability of type A = 0.1089
After 180 rolls. Probability of type A = 0.0776
```

```
After 200 rolls. Probability of type A = 0.0553
```

The good news is that even given a misleading prior, the probability converges towards the truth as the number of examples grows. Notice, by the way, that it didn't converge monotonically. The probability after 120 rolls was higher than after 100—indicating that those 20 rolls were more consistent with a die of type A than with a die of type B or C.

Had we started with a better prior, it would have converged faster. If we go back to 1/3 as the initial prior, the probability is 0.0335 after 100 rolls, and 0.0205 after 200 rolls.

---

[137] http://imgs.xkcd.com/comics/frequentists_vs_bayesians.png

[138] If you are of the frequentist persuasion, keep in mind that this cartoon is a parody—not a serious critique of your religious beliefs.

[139] This number may strike you as high. It is. The average American adult weighs about 40 pounds more than the average adult in Japan. The only three countries on earth with higher average adult weights than the U.S. are Nauru, Tonga, and Micronesia.

[140] The probability of weighing more than the *median* weight is 0.5, but that doesn't imply that the probability of weighing more than the *mean* is 0.5. However, for the purposes of this discussion, let's pretend that it does.

[141] By "Maltese" we mean somebody from the country of Malta. We have no idea what fraction of the world's males are cute little dogs.

[142] In medical jargon, a "positive" test is usually bad news. It implies that a marker of disease has been found.

[143] Bayes' theorem is named after Rev. Thomas Bayes (1701–1761), and was first published two years after his death. It was popularized by Laplace, who published the modern formulation of the theorem in 1812 in his *Théorie analytique des probabilités*.

[144] This is one of the reasons that there is considerable controversary in the medical community about the value of mammography as a routine screening tool for some cohorts.

21 LIES, DAMNED LIES, AND STATISTICS

"If you can't prove what you want to prove, demonstrate something else and pretend they are the same thing. In the daze that follows the collision of statistics with the human mind, hardly anyone will notice the difference."[145]

Statistical thinking is a relatively new invention. For most of recorded history things were assessed qualitatively rather than quantitatively. People must have had an intuitive sense of some statistical facts (e.g., that women are usually shorter than men), but they had no mathematical tools that would allow them to proceed from anecdotal evidence to statistical conclusions. This started to change in the middle of the 17th century, most notably with the publication of John Graunt's *Natural and Political Observations Made Upon the Bills of Mortality*. This pioneering work used statistical analysis to estimate the population of London from death rolls, and attempted to provide a model that could be used to predict the spread of plague.

Alas, since that time people have used statistics as much to mislead as to inform. Some have willfully used statistics to mislead; others have merely been incompetent. In this chapter we discuss a few ways in which people can be fooled into drawing inappropriate inferences from statistical data. We trust that you will use this information only for good—to become a better consumer and a more honest purveyor of statistical information.

21.1 Garbage In Garbage Out (GIGO)

"On two occasions I have been asked [by members of Parliament], 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question." — Charles Babbage[146]

The message here is a simple one. If the input data is seriously flawed, no amount of statistical massaging will produce a meaningful result.

The 1840 United States census showed that insanity among free blacks and mulattoes was roughly ten times more common than among enslaved blacks and mulattoes. The conclusion was obvious. As U.S. Senator (and former Vice President and future Secretary of State) John C. Calhoun put it, "The data on insanity revealed in this census is unimpeachable. From it our nation must conclude that the

abolition of slavery would be to the African a curse." Never mind that it was soon clear that the census was riddled with errors. As Calhoun reportedly explained to John Quincy Adams, "there were so many errors they balanced one another, and led to the same conclusion as if they were all correct."

Calhoun's (perhaps willfully) spurious response to Adams was based on a classical error, the **assumption of independence**. Were he more sophisticated mathematically, he might have said something like, "I believe that the measurement errors are unbiased and independent of each of other, and therefore evenly distributed on either side of the mean." In fact, later analysis showed that the errors were so heavily biased that no statistically valid conclusions could be drawn.[147]

21.2 Tests Are Imperfect

Every experiment should be viewed as a potentially flawed test. We can perform a test for a chemical, a phenomenon, a disease, etc. However, the event for which we are testing is not necessarily the same as the result of the test. Professors design exams with the goal of understanding how well a student has mastered some subject matter, but the result of the exam should not be confused with how much a student actually understands. Every test has some inherent error rate. Imagine that a student learning a second language has been asked to learn the meaning of 100 words, but has learned the meaning of only 80 of them. His rate of understanding is 80%, but the probability that he will score 80% on a test with 20 words is certainly not 1.

Tests can have both false negatives and false positives. As we saw in Chapter 20, a negative mammogram does not guarantee absence of breast cancer, and a positive mammogram doesn't guarantee its presence. Furthermore, the test probability and the event probability are not the same thing. This is especially relevant when testing for a rare event, e.g., the presence of a rare disease. If the cost of a false negative is high (e.g., missing the presence of a serious but curable disease), the test should be designed to be highly sensitive, even at the cost of there being a large number of false positives.

21.3 Pictures Can Be Deceiving

There can be no doubt about the utility of graphics for quickly conveying information. However, when used carelessly (or maliciously) a plot can be highly misleading. Consider, for example, the charts in Figure 21.1 depicting housing prices in the U.S. Midwestern states.

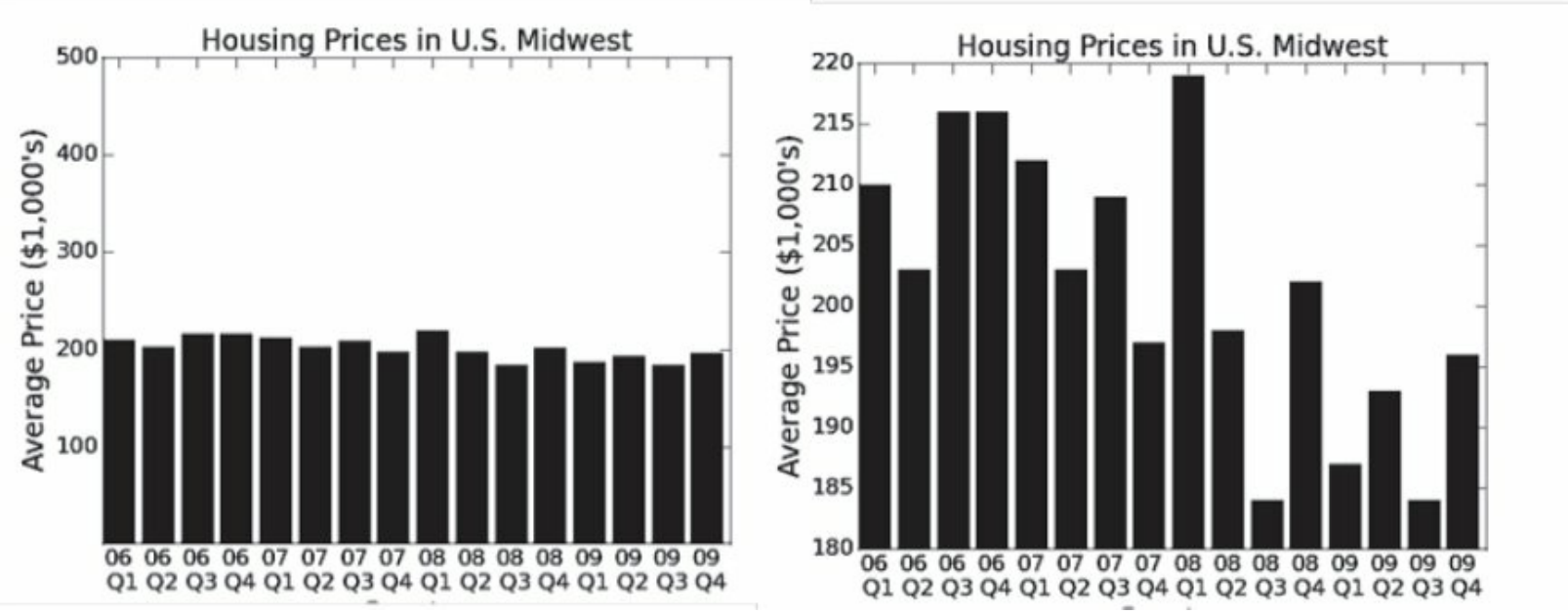

Figure 21.1 Housing prices in the U.S. Midwest

Looking at the chart on the left of Figure 21.1, it seems as if housing prices were pretty stable during the period 2006-2009. But wait a minute! Wasn't there a collapse of U.S. residential real estate followed by a global financial crisis in late 2008? There was indeed, as shown in the chart on the right.

These two charts show exactly the same data, but convey very different impressions. The chart on the left was designed to give the impression that housing prices had been stable. On the y-axis, the designer used a scale ranging from the absurdly low average price for a house of $1,000 to the improbably high average price of $500,000. This minimized the amount of space devoted to the area where prices are changing, giving the impression that the changes were relatively small. The chart on the right was designed to give the impression that housing prices moved erratically, and then crashed. The designer used a narrow range of prices, so the sizes of the changes were exaggerated.

The code in Figure 21.2 produces the two plots we looked at above and a plot intended to give an accurate impression of the movement of housing prices. It uses two plotting facilities that we have not yet seen. The call `pylab.bar(quarters, prices, width)` produces a **bar chart** with bars of the given width. The left edges of the bars are the values of the elements of the list `quarters` and the heights of the bars are the values of the corresponding elements of the list `prices`. The function call `pylab.xticks(quarters+width/2, labels)` describes the labels to be associated with the bars. The first argument specifies where each label is to be placed and the second argument the text of the labels. The function `yticks` behaves analogously. The call `plotHousing('fair')` produces the plot in Figure 21.3.

```
def plotHousing(impression):
    """Assumes impression a str. Must be one of 'flat',
         'volatile,' and 'fair'
       Produce bar chart of housing prices over time"""
    f = open('midWestHousingPrices.txt', 'r')
    #Each line of file contains year quarter price
    #for Midwest region of U.S.
    labels, prices = ([], [])
    for line in f:
        year, quarter, price = line.split()
        label = year[2:4] + '\n Q' + quarter[1]
        labels.append(label)
        prices.append(int(price)/1000)
    quarters = pylab.arange(len(labels)) #x coords of bars
    width = 0.8 #Width of bars
    pylab.bar(quarters, prices, width)
    pylab.xticks(quarters+width/2, labels)
    pylab.title('Housing Prices in U.S. Midwest')
    pylab.xlabel('Quarter')
    pylab.ylabel('Average Price ($1,000\'s)')
    if impression == 'flat':
        pylab.ylim(1, 500)
    elif impression == 'volatile':
        pylab.ylim(180, 220)
    elif impression == 'fair':
        pylab.ylim(150, 250)
    else:
        raise ValueError

plotHousing('flat')
pylab.figure()
plotHousing('volatile')
```

Figure 21.2 Plotting housing prices

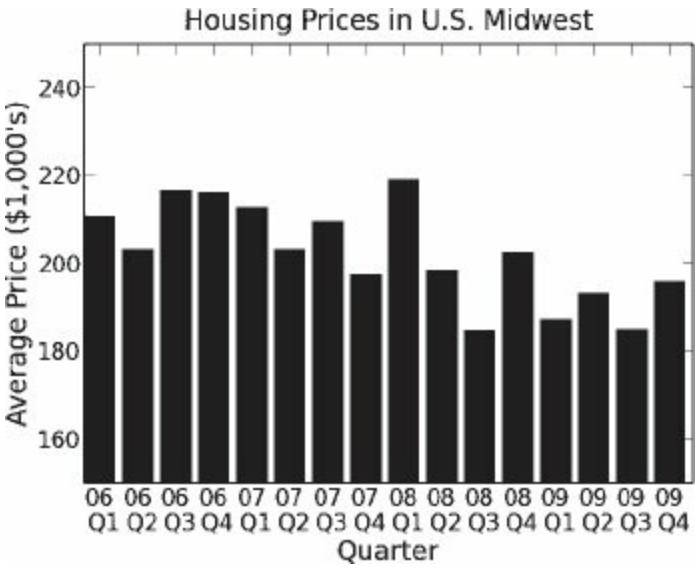

Figure 21.3 A different view of housing prices

21.4 Cum Hoc Ergo Propter Hoc[148]

It has been shown that college students who regularly attend class have higher average grades than students who attend class only sporadically. Those of us who teach these classes would like to believe that this is because the students learn something from the classes we teach. Of course, it is at least equally likely that those students get better grades because students who are more likely to attend classes are also more likely to study hard.

Correlation is a measure of the degree to which two variables move in the same direction. If x moves in the same direction as y, the variables are positively correlated. If they move in opposite directions they are negatively correlated. If there is no relationship, the correlation is 0. People's heights are positively correlated with the heights of their parents. The correlation between hours spent playing video games and grade point average is negative.

When two things are correlated, there is a temptation to assume that one has caused the other. Consider the incidence of flu in North America. The number of cases rises and falls in a predictable pattern. There are almost no cases in the summer; the number of cases starts to rise in the early fall and then starts dropping as summer approaches. Now consider the number of children attending school. There are very few children in school in the summer, enrollment starts to rise in the early fall, and then drops as summer approaches.

The correlation between the opening of schools and the rise in the incidence of flu is inarguable. This has led many to conclude that that going to school is an important causative factor in the spread of flu. That might be true, but one cannot conclude it based simply on the correlation. Correlation does not imply causation! After all, the correlation could be used just as easily to justify the belief that flu outbreaks cause schools to be in session. Or perhaps there is no causal relationship in either direction, and there is some **lurking variable** that we have not considered that causes each. In fact, as it happens, the flu virus survives considerably longer in cool dry air than it does in warm wet air, and in North America both the flu season and school sessions are correlated with cooler and dryer

weather.

Given enough retrospective data, it is always possible to find two variables that are correlated, as illustrated by the chart in Figure 21.4.[149]

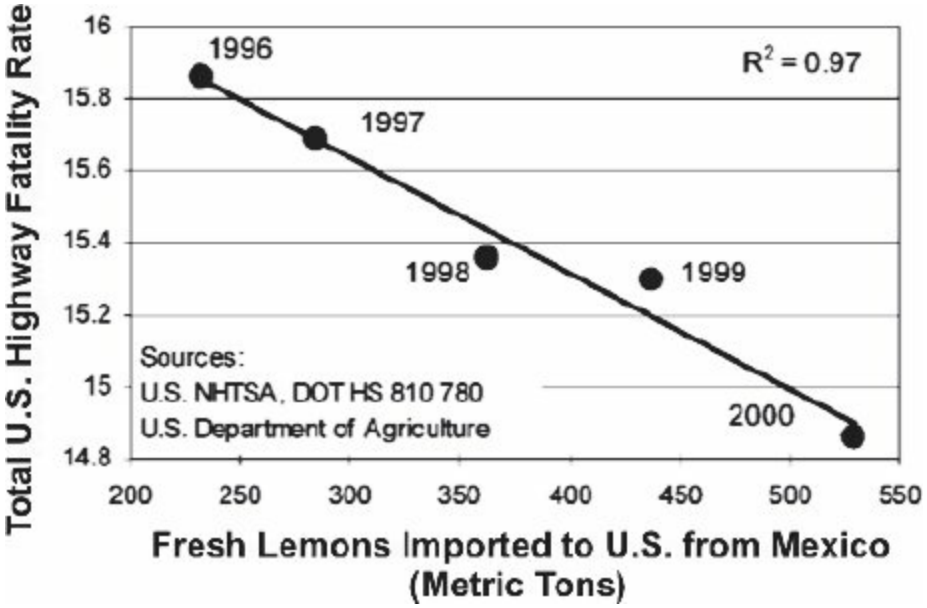

Figure 21.4 Do Mexican lemons save lives?

When such correlations are found, the first thing to do is to ask whether there is a plausible theory explaining the correlation.

Falling prey to the *cum hoc ergo propter hoc* fallacy can be quite dangerous. At the start of 2002, roughly six million American women were being prescribed hormone replacement therapy (HRT) in the belief that it would substantially lower their risk of cardiovascular disease. That belief was supported by several highly reputable published studies that demonstrated a reduced incidence of cardiovascular death among women using HRT.

Many women, and their physicians, were taken by surprise when the *Journal of the American Medical Society* published an article asserting that HRT in fact increased the risk of cardiovascular disease.[150] How could this have happened?

Reanalysis of some of the earlier studies showed that women undertaking HRT were likely to be from groups with better than average diet and exercise regimes. Perhaps the women undertaking HRT were on average more health conscious than the other women in the study, so that taking HRT and improved cardiac health were coincident effects of a common cause.

21.5 Statistical Measures Don't Tell the Whole Story

There are an enormous number of different statistics that can be extracted from a data set. By carefully choosing among these, it is possible to convey a variety of different impressions about the same data. A good antidote is to look at the data set itself.

In 1973, the statistician F.J. Anscombe published a paper with the table in Figure 21.5. It contains the <x, y> coordinates of points from each of four data sets. Each of the four data sets has the same mean value for x (9.0), the same mean value for y (7.5), the same variance for x (10.0), the same

variance for y (3.75), and the same correlation between x and y (0.816). And if we use linear regression to fit a line to each, we get the same result for each, $y = 0.5x + 3$.

x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Figure 21.5 Statistics for Anscombe's Quartet

Does this mean that there is no obvious way to distinguish these data sets from each other? No. One simply needs to plot the data to see that the data sets are not at all alike (Figure 21.6).

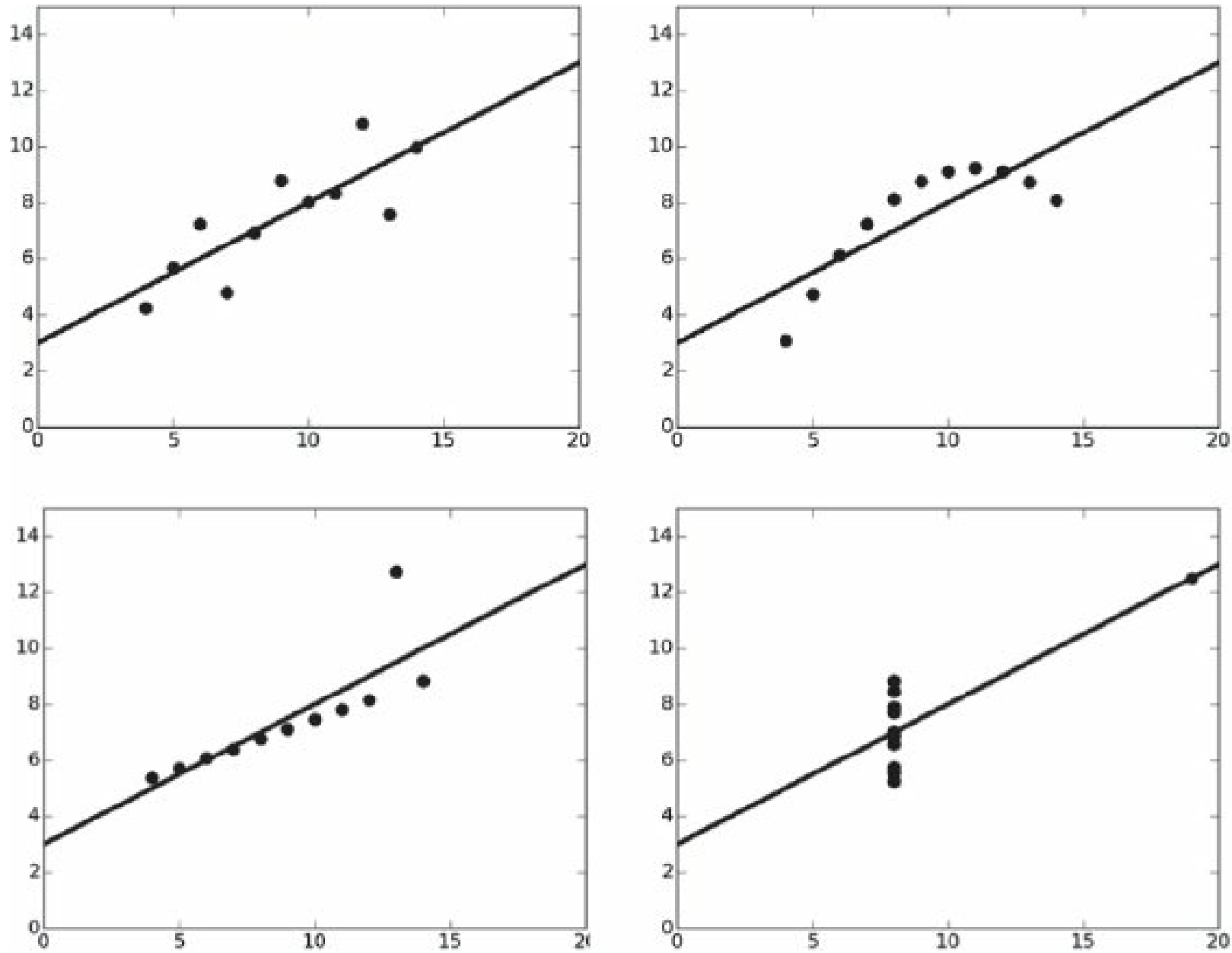

Figure 21.6 Data for Ancsombe's Quartet

The moral is simple: if possible, always take a look at some representation of the raw data.

21.6 Sampling Bias

During World War II, whenever an Allied plane would return from a mission over Europe the plane would be inspected to see where the flak from antiaircraft artillery had impacted. Based upon this data, mechanics reinforced those areas of the planes that seemed most likely to be hit by flak.

What's wrong with this? They did not inspect the planes that failed to return from missions because they had been downed by flak. Perhaps these unexamined planes failed to return precisely because they were hit in the places where the flak would do the most damage. This particular error is called **non-response bias**. It is quite common in surveys. At many universities, for example, students are asked during one of the lectures late in the term to fill out a form rating the quality of the professor's lectures. Though the results of such surveys are often unflattering, they could be worse.

Those students who think that the lectures are so bad that they aren't worth attending are not included in the survey.[151]

As discussed in Chapter 17, all statistical techniques are based upon the assumption that by sampling a subset of a population we can infer things about the population as a whole. If random sampling is used, we can make precise mathematical statements about the expected relationship of the sample to the entire population. Unfortunately, many studies, particularly in the social sciences, are based on what is called **convenience** (or **accidental) sampling**. This involves choosing samples based on how easy they are to procure. Why do so many psychological studies use populations of undergraduates? Because they are easy to find on college campuses. A convenience sample *might* be representative, but there is no way of knowing whether it actually *is* representative.

21.7 Context Matters

It is easy to read more into the data than it actually implies, especially when viewing the data out of context. On April 29, 2009, CNN reported that, "Mexican health officials suspect that the swine flu outbreak has caused more than 159 deaths and roughly 2,500 illnesses." Pretty scary stuff—until one compares it to the 36,000 deaths attributable annually to the seasonal flu in the U.S.

An often quoted, and accurate, statistic is that most auto accidents happen within 10 miles of home. So what? Most driving is done within 10 miles of home! And besides, what does "home" mean in this context? The statistic is computed using the address at which the automobile is registered as "home." Might one reduce the probability of getting into an accident by merely registering one's car in some distant place?

Opponents of government initiatives to reduce the prevalence of guns in the United States are fond of quoting the statistic that roughly 99.8% of the firearms in the U.S. will not be used to commit a violent crime in any given year. But without some context, it's hard to know what that implies. Does it imply that there is not much gun violence in the U.S.? The National Rifle Association reports that there are roughly 300 million privately owned firearms in the U.S.—0.2% of 300 million is 600,000!

21.8 Beware of Extrapolation

It is all too easy to extrapolate from data. We did that in Section 18.1.1 when we extended fits derived from linear regression beyond the data used in the regression. Extrapolation should be done only when one has a sound theoretical justification for doing so. Be especially wary of straight-line extrapolations.

Consider the plot on the left in Figure 21.7. It shows the growth of Internet usage in the United States from 1994 to 2000. As you can see, a straight line provides a pretty good fit.

The plot on the right of Figure 21.7 uses this fit to project the percentage of the U.S. population using the Internet in following years. The projection is a bit hard to believe. It seems unlikely that by 2009 everybody in the U.S. was using the Internet, and even less likely that by 2015 more than 140%

of the U.S. population was using the Internet.

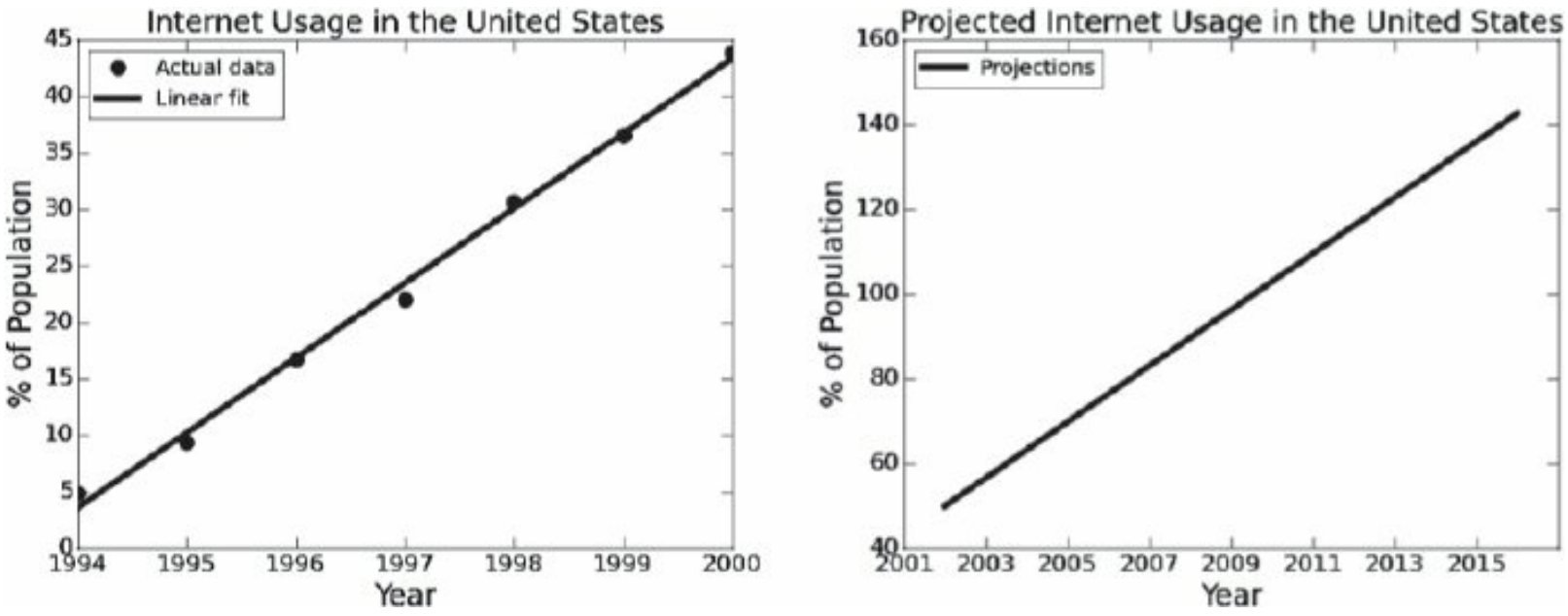

Figure 21.7 Growth of Internet usage in U.S.

21.9 The Texas Sharpshooter Fallacy

Imagine that you are driving down a country road in Texas. You see a barn that has six targets painted on it, and a bullet hole at the very center of each target. "Yes sir," says the owner of the barn, "I never miss." "That's right," says his spouse, "there ain't a man in the state of Texas who's more accurate with a paint brush." Got it? He fired the six shots, and then painted the targets around them.

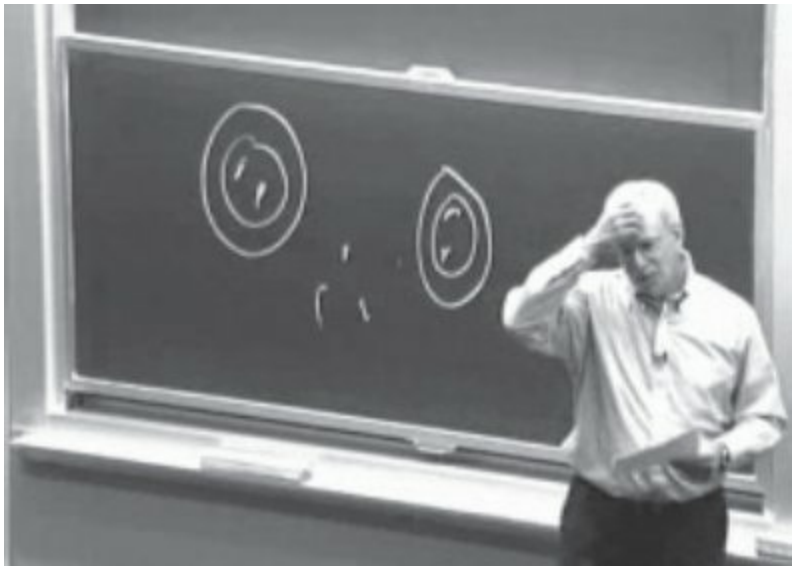

Figure 21.8 Professor puzzles over students' chalk-throwing accuracy

A classic of the genre appeared in 2001.[152] It reported that a research team at the Royal Cornhill Hospital in Aberdeen had discovered that "anorexic women are most likely to have been born in the spring or early summer… Between March and June there were 13% more anorexics born than

average, and 30% more in June itself."

Let's look at that worrisome statistic for those women born in June. The team studied 446 women who had been diagnosed as anorexic, so the mean number of births per month was slightly more than 37. This suggests that the number born in June was 48 (37∗1.3). Let's write a short program (Figure 21.9) to estimate the probability that this occurred purely by chance.

```
def juneProb(numTrials):
    june48 = 0
    for trial in range(numTrials):
      june = 0
      for i in range(446):
          if random.randint(1,12) == 6:
              june += 1
      if june >= 48:
          june48 += 1
    jProb = round(june48/numTrials, 4)
    print('Probability of at least 48 births in June =', jProb)
```

Figure 21.9 Probability of 48 anorexics being born in June

When we ran `juneProb(10000)` it printed

```
Probability of at least 48 births in June = 0.0427
```

It looks as if the probability of at least 48 babies being born in June purely by chance is around 4.5%. So perhaps those researchers in Aberdeen are on to something. Well, they might have been on to something had they started with the hypothesis that more babies who will become anorexic are born in June, and then run a study designed to check that hypothesis.

But that is not what they did. Instead, they looked at the data and then, imitating the Texas sharpshooter, drew a circle around June. The right statistical question to have asked is what is the probability that there was at least one month (out of 12) in which at least 48 babies were born. The program in Figure 21.10 answers that question.

```
def anyProb(numTrials):
    anyMonth48 = 0
    for trial in range(numTrials):
      months = [0]*12
      for i in range(446):
          months[random.randint(0,11)] += 1
      if max(months) >= 48:
          anyMonth48 += 1
    aProb = round(anyMonth48/numTrials, 4)
    print('Probability of at least 48 births in some month =',aProb)
```

Figure 21.10 Probability of 48 anorexics being born in some month

The call `anyProb(10000)` printed

```
Probability of at least 48 births in some month = 0.4357
```

It appears that it is not so unlikely after all that the results reported in the study reflect a chance occurrence rather a real association between birth month and anorexia. One doesn't have to come from Texas to fall victim to the Texas Sharpshooter Fallacy.

What we see here is that the statistical significance of a result depends upon the way the experiment was conducted. If the Aberdeen group had started out with the hypothesis that more anorexics are born in June, their result would be worth considering. But if they started off with the hypothesis that there exists a month in which an unusually large proportion of anorexics are born, their result is not very compelling. In effect, they were testing multiple hypothesis and probably should have applied a Bonferroni correction (see Section 19.6).

What next steps might the Aberdeen group have taken to test their newfound hypothesis? One possibility is to conduct a **prospective study**. In a prospective study, one starts with a set of hypotheses, recruits subjects before they have developed the outcome of interest (anorexia in this case), and then follows the subjects for a period of time. If the group had conducted a prospective study with a specific hypothesis and gotten similar results, one might be convinced.

Prospective studies can be expensive and time-consuming to perform. In a **retrospective study**, one has to examine existing data in ways that reduce the likelihood of getting misleading results. One common technique, as discussed in Section 18.4, is to split the data into a training set and a held out test set. For example, they could have chosen 446/2 women at random from their data (the training set), and tallied the number of births for each month. They could have then compared that to the number of births each month for the remaining women (the holdout set).

21.10 Percentages Can Confuse

An investment advisor called a client to report that the value of his stock portfolio had risen 16%

over the last month. He admitted that there had been some ups and downs over the year, but was pleased to report that the average monthly change was +0.5%. Image the client's surprise when he got his statement for the year, and observed that the value of his portfolio had declined over the year.

He called his advisor, and accused him of being a liar. "It looks to me," he said, "like my portfolio declined by about 8%, and you told me that it went up by 0.5% a month." "I did not," the financial advisor replied, "I told you that the average monthly change was +0.5%." When he examined his monthly statements, the investor realized that he had not been lied to, just misled. His portfolio went down by 15% in each month during the first half of the year, and then went up by 16% in each month during the second half of the year.

When thinking about percentages, we always need to pay attention to the basis on which the percentage is computed. In this case, the 15% declines were on a higher average basis than the 16% increases.

Percentages can be particularly misleading when applied to a small basis. You might read about a drug that has a side effect of increasing the incidence of some illness by 200%. But if the base incidence of the disease is very low, say one in 1,000,000, you might well decide that the risk of taking the drug was more than counterbalanced by the drug's positive effects.

21.11 Statistically Significant Differences Can Be Insignificant

An admissions officer at the Maui Institute of Technology (MIT), wishing to convince the world that MIT's admissions process is "gender-blind," trumpeted, "At MIT, there is no significant difference between the grade point averages of men and women." The same day, an ardent female chauvinist proclaimed that "At MIT, the women have a significantly higher grade point average than the men." A puzzled reporter at the student newspaper decided to examine the data and expose the liar. But when she finally managed to pry the data out of the university, she concluded that both were telling the truth.

What does the sentence, "At MIT, the women have a significantly higher grade point average than the men," actually mean? People who have not studied statistics (most of the population) would probably conclude that there is a "meaningful" difference between the GPAs of women and men attending MIT. In contrast, those who have recently studied statistics might conclude only that 1) the average GPA of women is higher than that of men, and 2) the null hypothesis that the difference in GPA can be attributed to randomness can be rejected at the 5% level.

Suppose, for example, that there were 2500 women and 2500 men studying at MIT. Suppose further that the mean GPA of men was 3.5, the mean GPA of women was 3.51, and the standard deviation of the GPA for both men and women was 0.25. Most sensible people would consider the difference in GPAs "insignificant." However, from a statistical point of view the difference is "significant" at close to the 2% level. What is the root of this strange dichotomy? As we showed in Section 19.5, when a study has enough power—i.e, enough examples—even insignificant differences can be statistically significant.

A related problem arises when a study is very small. Suppose you flipped a coin twice and it came up heads both times. Now, let's use the two-tailed one-sample t-test we saw in Section 19.3 to test the null hypothesis that the coin is fair. If we assume that the value of heads is 1 and the value of

tails is 0, we can get the p-value using the code

```
stats.ttest_1samp([1, 1], 0.5)[1]
```

It returns a p-value of 0, indicating that if the coin is fair the probability of getting two consecutive heads is nil.

21.12 The Regressive Fallacy

The **regressive fallacy** occurs when people fail to take into account the natural fluctuations of events.

All athletes have good days and bad days. When they have good days, they try not to change anything. When they have a series of unusually bad days, however, they often try to make changes. Whether or not the changes are actually constructive, regression to the mean (Section 15.3) makes it likely that over the next few days the athlete's performance will be better than the unusually poor performances preceding the changes. But that may not stop the athlete from assuming that there is a **treatment effect**, i.e., attributing the improved performance to the changes he or she made.

The Nobel prize-winning psychologist Daniel Kahneman tells a story about an Israeli Air Force flight instructor who rejected Kahneman's assertion that "rewards for improved performance work better than punishment for mistakes." The instructor's argument was "On many occasions I have praised flights cadets for clean execution of some aerobatic maneuver. The next time they try the same maneuver they usually do worse. On the other hand, I have often screamed into a cadet's earphone for bad execution, and in general he does better on the next try."[153] It is natural for humans to imagine a treatment effect, because we like to think causally. But sometimes it is simply a matter of luck.

Imagining a treatment effect when there is none can be dangerous. It can lead to the belief that vaccinations are harmful, that snake oil cures all aches and pains, or that investing exclusively in mutual funds that "beat the market" last year is a good strategy.

21.13 Just Beware

It would be easy, and fun, to fill a few hundred pages with a history of statistical abuses. But by now you probably got the message: It's just as easy to lie with numbers as it is to lie with words. Make sure that you understand what is actually being measured and how those "statistically significant" results were computed before you jump to conclusions. As Darrell Huff said, "If you torture the data long enough, it will confess to anything."[154]

---

[145] Darrell Huff, *How to Lie with Statistics*, 1954.

[146] Charles Babbage, 1791-1871, was an English mathematician and mechanical engineer who is credited with having designed the first programmable computer. He never succeeded in building a working machine, but in 1991 a working mechanical device for evaluating polynomials was built from his original plans.

[147] We should note that Calhoun was in office over 150 years ago. It goes without saying that no contemporary politician would find ways to abuse statistics to support a wrong-headed position.

[148] Statisticians, like attorneys and physicians, sometimes use Latin for no obvious reason other than to seem erudite. This phrase means, "with this, therefore because of this."

[149] Stephen R. Johnson, "The Trouble with QSAR (or How I Learned to Stop Worrying and Embrace Fallacy)," *J. Chem. Inf. Model.*, 2008.

[150] Nelson HD, Humphrey LL, Nygren P, Teutsch SM, Allan JD. Postmenopausal hormone replacement therapy: scientific review. *JAMA*. 2002;288:872-881.

[151] The move to online surveys, which allows students who do not attend class to participate in the survey, does not augur well for the egos of professors.

[152] Eagles, John, et al., "Season of birth in females with anorexia nervosa in Northeast Scotland," *International Journal of Eating Disorders*, 30, 2, September 2001.

[153] *Thinking, Fast and Slow*, Daniel Kahneman, Farrar, Straus and Giroux, 2011, p.175.

[154] Darrell Huff, *How to Lie with Statistics*, 1954. A similar remark is attributed to the Nobel prize winning economist Ronald Coase.