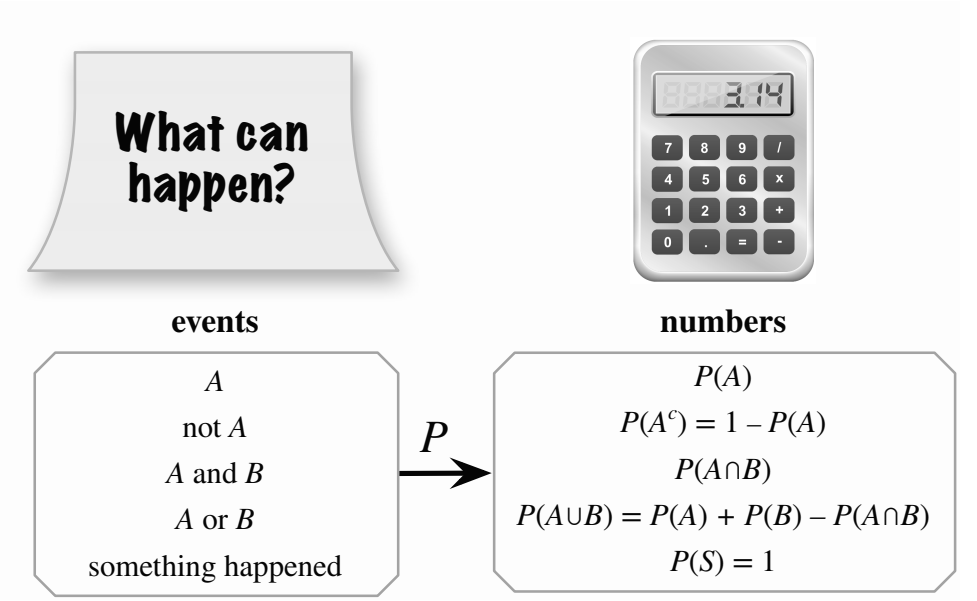

FIGURE 1.7
It is important to distinguish between *events* and *probabilities*. The former are sets, while the latter are numbers. Before the experiment is done, we generally don't know whether or not a particular event will occur (happen). So we assign it a probability of happening, using a probability function P. We can use set operations to define new events in terms of old events, and the properties of probabilities to relate the probabilities of the new events to those of the old events.

1.8 R

R is a very powerful, popular environment for statistical computing and graphics, freely available for Mac OS X, Windows, and UNIX systems. Knowing how to use R is an extremely useful skill. R and various supporting information can be obtained at `https://www.r-project.org`. RStudio is an excellent alternative interface for R, freely available at `https://www.rstudio.com`.

In the R section at the end of each chapter, we provide R code to let you try out some of the examples from the chapter, especially via simulation. These sections are not intended to be a full introduction to R; many R tutorials are available for free online, and many books on R are available. But the R sections show how to implement various simulations, computations, and visualizations that naturally accompany the material from each chapter. The R code at the end of each chapter is also available at `http://stat110.net`.

Vectors

R is built around *vectors*, and getting familiar with "vectorized thinking" is very important for using R effectively. To create a vector, we can use the `c` command (which stands for *combine* or *concatenate*). For example,

```
v <- c(3,1,4,1,5,9)
```

defines **v** to be the vector $(3, 1, 4, 1, 5, 9)$. (The left arrow `<-` is typed as `<` followed by `-`. The symbol `=` can be used instead, but the arrow is more suggestive of the fact that the variable on the left is being set equal to the value on the right.) Similarly, `n <- 110` sets n equal to 110; R views n as a vector of length 1.

```
sum(v)
```

adds up the entries of **v**, `max(v)` gives the largest value, `min(v)` gives the smallest value, and `length(v)` gives the length.

A shortcut for getting the vector $(1, 2, \ldots, n)$ is to type `1:n`; more generally, if m and n are integers then `m:n` gives the sequence of integers from m to n (in increasing order if $m \leq n$ and in decreasing order otherwise).

To access the ith entry of a vector **v**, use `v[i]`. We can also get subvectors easily:

```
v[c(1,3,5)]
```

gives the vector consisting of the 1st, 3rd, and 5th entries of **v**. It's also possible to get a subvector by specifying what to exclude, using a minus sign:

```
v[-(2:4)]
```

gives the vector obtained by removing the 2nd through 4th entries of **v** (the parentheses are needed since `-2:4` would be $(-2,-1,\ldots,4)$).

Many operations in R are interpreted *componentwise*. For example, in math the cube of a vector doesn't have a standard definition, but in R typing `v^3` simply cubes each entry individually. Similarly,

```
1/(1:100)^2
```

is a very compact way to get the vector $(1, \frac{1}{2^2}, \frac{1}{3^2}, \ldots, \frac{1}{100^2})$.

In math, $\mathbf{v}+\mathbf{w}$ is undefined if **v** and **w** are vectors of different lengths, but in R the shorter vector gets "recycled"! For example, `v+3` adds 3 to each entry of **v**.

Factorials and binomial coefficients

We can compute $n!$ using `factorial(n)` and $\binom{n}{k}$ using `choose(n,k)`. As we have seen, factorials grow extremely quickly. What is the largest n for which R returns a number for `factorial(n)`? Beyond that point, R will return `Inf` (infinity), with a warning message. But it may still be possible to use `lfactorial(n)` for larger values of n, which computes $\log(n!)$. Similarly, `lchoose(n,k)` computes $\log\binom{n}{k}$.

Sampling and simulation

The `sample` command is a useful way of drawing random samples in R. (Technically, they are *pseudo-random* since there is an underlying deterministic algorithm, but they "look like" random samples for almost all practical purposes.) For example,

```
n <- 10; k <- 5
sample(n,k)
```

generates an ordered random sample of 5 of the numbers from 1 to 10, without replacement, and with equal probabilities given to each number. To sample with replacement instead, just add in `replace = TRUE`:

```
n <- 10; k <- 5
sample(n,k,replace=TRUE)
```

To generate a random permutation of $1, 2, \ldots, n$ we can use `sample(n,n)`, which because of R's default settings can be abbreviated to `sample(n)`.

We can also use `sample` to draw from a non-numeric vector. For example, `letters` is built into R as the vector consisting of the 26 lowercase letters of the English alphabet, and `sample(letters,7)` will generate a random 7-letter "word" by sampling from the alphabet, without replacement.

The `sample` command also allows us to specify general probabilities for sampling each number. For example,

```
sample(4, 3, replace=TRUE, prob=c(0.1,0.2,0.3,0.4))
```

samples three numbers between 1 and 4, with replacement, and with probabilities given by $(0.1, 0.2, 0.3, 0.4)$. If the sampling is without replacement, then at each stage the probability of any not-yet-chosen number is *proportional* to its original probability.

Generating *many* random samples allows us to perform a *simulation* for a probability problem. The `replicate` command, which is explained below, is a convenient way to do this.

Matching problem simulation

Let's show by simulation that the probability of a matching card in Example 1.6.4 is approximately $1 - 1/e$ when the deck is sufficiently large. Using R, we can perform the experiment a bunch of times and see how many times we encounter at least one matching card:

```
n <- 100
r <- replicate(10^4,sum(sample(n)==(1:n)))
sum(r>=1)/10^4
```

In the first line, we choose how many cards are in the deck (here, 100 cards). In the second line, let's work from the inside out:

- `sample(n)==(1:n)` is a vector of length `n`, the ith element of which equals 1 if the ith card matches its position in the deck and 0 otherwise. That's because for two numbers a and b, the expression `a==b` is TRUE if $a = b$ and FALSE otherwise, and TRUE is encoded as 1 and FALSE is encoded as 0.
- `sum` adds up the elements of the vector, giving us the number of matching cards in this run of the experiment.
- `replicate` does this 10^4 times. We store the results in `r`, a vector of length 10^4 containing the numbers of matched cards from each run of the experiment.

In the last line, we add up the number of times where there was at least one matching card, and we divide by the number of simulations.

To explain what the code is doing within the code rather than in separate documentation, we can add *comments* using the `#` symbol to mark the start of a comment. Comments are ignored by R but can make the code much easier to understand for the reader (who could be you—even if you will be the only one using your code, it is often hard to remember what everything means and how the code is supposed to work when looking at it a month after writing it). Short comments can be on the

same line as the corresponding code; longer comments should be on separate lines. For example, a commented version of the above simulation is:

```
n <- 100                                        # number of cards
r <- replicate(10^4,sum(sample(n)==(1:n)))      # shuffle; count matches
sum(r>=1)/10^4                                  # proportion with a match
```

What did you get when you ran the code? We got 0.63, which is quite close to $1 - 1/e$.

Birthday problem calculation and simulation

The following code uses `prod` (which gives the product of a vector) to calculate the probability of at least one birthday match in a group of 23 people:

```
k <- 23
1-prod((365-k+1):365)/365^k
```

Better yet, R has built-in functions, `pbirthday` and `qbirthday`, for the birthday problem! `pbirthday(k)` returns the probability of at least one match if the room has `k` people. `qbirthday(p)` returns the number of people needed in order to have probability `p` of at least one match. For example, `pbirthday(23)` is 0.507 and `qbirthday(0.5)` is 23.

We can also find the probability of having at least one *triple birthday match*, i.e., three people with the same birthday; all we have to do is add `coincident=3` to say we're looking for triple matches. For example, `pbirthday(23,coincident=3)` returns 0.014, so 23 people give us only a 1.4% chance of a triple birthday match. `qbirthday(0.5,coincident=3)` returns 88, so we'd need 88 people to have at least a 50% chance of at least one triple birthday match.

To simulate the birthday problem, we can use

```
b <- sample(1:365,23,replace=TRUE)
tabulate(b)
```

to generate random birthdays for 23 people and then tabulate the counts of how many people were born on each day (the command `table(b)` creates a prettier table, but is slower). We can run 10^4 repetitions as follows:

```
r <- replicate(10^4, max(tabulate(sample(1:365,23,replace=TRUE))))
sum(r>=2)/10^4
```

If the probabilities of various days are not all equal, the calculation becomes much more difficult, but the simulation can easily be extended since `sample` allows us to specify the probability of each day (by default `sample` assigns equal probabilities, so in the above the probability is 1/365 for each day).