2

Simple Random Sampling

Data from a simple random sample (SRS) can be analyzed using R functions that are designed for data that can be considered as independent and identically distributed, and an SRS can be selected using the R *sample* function. For other types of probability samples, however, you either need to write your own function to account for the survey design or employ functions that have been written by other R users in contributed packages. This chapter reviews how to select a sample and compute estimates from an SRS using functions in base R. It also introduces you to the *srswor* and *srswr* functions from the `sampling` package (Tillé and Matei, 2021) to select an SRS, with or without replacement; and the *svydesign*, *svymean*, and *svytotal* functions from the `survey` package (Lumley, 2020) to calculate the statistics.

All code in this chapter can be found in file `ch02.R` on the book website (see the Preface for the website address). The data sets are available from the book website and in the R package `SDAResources` (Lu and Lohr, 2021). The variables in the data sets are described in Appendix A.

In this and future chapters, load the following three packages before starting the computations. You installed these packages in Chapter 1. In the R console, type:

```
library(survey)
library(sampling)
library(SDAResources)
```

Before calculating statistics, let's first look at how to use R functions to select an SRS from a population.

2.1 Selecting a Simple Random Sample

Example 2.5 of SDA. *Selecting an SRS from a population.* SDA used a random number table to select an SRS of size 4 from a population of size 10. There are several options for selecting an SRS in R.

Using the *sample* function in base R. Base R contains the function *sample* that can be used to select an SRS. We can select an SRS (without replacement) of size 4 from a population of size 10 as follows:

```
# Set the seed for random number generation
set.seed(108742)
# Select an SRS of size n=4 from a population of size N=10 without replacement
srs4 <- sample(1:10, 4, replace = FALSE)
# Print the sample to see
# Can print an object by typing its name, or typing "print(object)"
```

DOI: 10.1201/9781003228196-2

```
srs4
## [1] 1 8 9 5
```

The first line of the code uses the function

`set.seed(seed)`

where *seed* is an integer that you supply to the function. If you want to be able to reproduce your sample later, call *set.seed* immediately before calling the function that generates a sample, and record the value of *seed* that you used. You will then get the same sample the next time you call the *sample* function with the same value of *seed*.[1] If you do not call *set.seed* during your R session, the starting point will be generated by the program.

For this example, the *sample* function provided a sample containing units 1, 8, 9, and 5. Running the sample function again, without using *set.seed*, gives a different sample. But when we reset the seed to the original value of 108742, we obtain the first sample $\{1, 8, 9, 5\}$ again.

```
# Run again, without setting a new seed.
sample(1:10, 4, replace = FALSE)
## [1]  9  7  3 10
# Now go back to original seed.
set.seed(108742)
sample(1:10, 4, replace = FALSE)
## [1] 1 8 9 5
```

The *sample* function is called with

`sample(x,size,replace=FALSE)`

to select an SRS without replacement of *size* observations from the population in x. We sample 4 observations from the population in the vector $[1, 2, \ldots, 10]$.

The *sample* function will also select a simple random sample with replacement by calling it with `replace = TRUE`. Unit "9" appears twice in the following with-replacement sample.

```
# Using the sample function to select an SRS with replacement
set.seed(101)
srswr4 <- sample(1:10, 4, replace = TRUE)
srswr4
## [1] 9 9 7 1
```

Using the *srswor* or *srswr* function from the sampling package. An alternative is to use function *srswor* or *srswr* to select an SRS without or with replacement, respectively. These are in the `sampling` package (Tillé and Matei, 2021), which you installed in Chapter 1. Now load the package and select a sample by calling:

`srswor(n,N)`

where n is the sample size and N is the population size.

```
# Load the sampling package if you have not already done so.
# library(sampling)
set.seed(1329)
# Select an SRS of size n=4 from a population of size N=10 without replacement.
s1<-srswor(4,10)
# List the units in the sample (the population units having s1=1).
```

[1] Samples generated by the *sample* function in R versions 3.6.0 and later, however, will differ from samples generated by earlier versions of R because the *sample* function was revised in version 3.6.0 to fix a bug.

```
s1
##  [1] 0 0 1 1 1 0 0 0 1 0
(1:10)[s1==1]
## [1] 3 4 5 9
```

The function *srswor* returns of vector of length N, with ones in the positions of the units selected for the sample. The sample in *s1* consists of units 3, 4, 5, and 9.

Function *srswr* for drawing a with-replacement sample is similar, but returns a vector containing the number of times each unit is in the sample.

```
# Select an SRS of size n=4 from a population of size N=10 with replacement.
set.seed(35882)
s2<-srswr(4,10)
# the selected units are 2 and 9
s2
##  [1] 0 2 0 0 0 0 0 0 2 0
(1:10)[s2!=0]
## [1] 2 9
# number of replicates, units 2 and 9 both appear twice
s2[s2!=0]
## [1] 2 2
# can use the getdata function to extract the sample from data frame with population
popdf<-data.frame(popid=1:10)
getdata(popdf,s2)
## [1] 2 2 9 9
```

The *getdata* function in the `sampling` package extracts the sample from the population listing. For this example, since units 2 and 9 are each selected twice, *getdata* repeats each of these units twice.

Note that the *sample* function has the sample size as the second argument, while the *srswor* and *srswr* functions have the sample size as the first argument. When you type `srswr(4,10)`, R assigns the values to variables in the order given in the function definition (see Usage in the help file for the function). If you want to assign values in a different order (or even if you want to use the same order but want to be able to read the call easily), name the variables when calling the function. All of the following are equivalent:

```
# Call a function using variable names
set.seed(35882)
srswr(4,10)
##  [1] 0 2 0 0 0 0 0 0 2 0
set.seed(35882)
srswr(n=4,N=10)
##  [1] 0 2 0 0 0 0 0 0 2 0
set.seed(35882)
srswr(N=10,n=4)
##  [1] 0 2 0 0 0 0 0 0 2 0
```

Example 2.6 of SDA. In SDA, the sample in `agsrs.csv` was selected from the population `agpop.csv` using random numbers generated in a spreadsheet. In the following, we show how to use R code to select a different SRS of size 300 from the 3078 counties in data set *agpop*. The function *getdata* is used to extract the sampled units from the population data, and the first 6 observations are printed.

```
# Select a different sample of size 300 from agpop
# Load the SDAResources package containing all the data in SDA book
```

```
# library(SDAResources)  # we comment this since we already loaded the package
data(agpop)  # Load the data set agpop
N <- nrow(agpop)
N  # 3078 observations
## [1] 3078
# Select an SRS of size n=300 from agpop
set.seed(8126834)
index <- srswor(300,N)
# each unit k is associated with index 1 or 0, with 1 indicating selection
index[1:10]
##  [1] 0 0 0 1 0 0 0 0 0 0
# agsrs2 is an SRS with size 300 selected from agpop
# extract the sampled units from the data frame containing the population
agsrs2 <- getdata(agpop,index)
agsrs2 <- agpop[(1:N)[index==1],] # alternative way to extract the sampled units
head(agsrs2)
##            county state acres92 acres87 acres82 farms92 farms87 farms82
## 4     JUNEAU AREA    AK     210     214     127       8       8      12
## 30 DE KALB COUNTY    AL  210733  213440  221502    1894    2047    2228
## 38    HALE COUNTY    AL  167583  154581  179618     382     441     481
## 46     LEE COUNTY    AL   67962   79836  100949     336     402     407
## 50 MADISON COUNTY    AL  224370  235478  292873     871     977    1101
## 62 RUSSELL COUNTY    AL  112620  143568  141048     213     276     314
##    largef92 largef87 largef82 smallf92 smallf87 smallf82 region
## 4         0        0        0        5        4        8      W
## 30       13        5        6      114      133      168      S
## 38       38       33       39       12       22       17      S
## 46       10       10       20       15       22       20      S
## 50       59       59       61       46       76       89      S
## 62       25       30       33       14       14       25      S
```

The functions *sample*, *srswor*, and *srswr* select a sample but do not provide sampling weights. After drawing the sample, you need to create a variable of sampling weights for the sampled units. For *agsrs2*, the weight variable *sampwt* has the value 3078/300 for all units.

```
# Create the variable of sampling weights
n <- nrow(agsrs2)
agsrs2$sampwt <- rep(3078/n,n)
# Check that the weights sum to N
sum(agsrs2$sampwt)
## [1] 3078
```

2.2 Computing Statistics from a Simple Random Sample

In this section, we look at two ways of computing estimates from an SRS: by using standard functions in R to calculate estimates from the formulas and by using functions in the `survey` package. We shall use the `survey` package to calculate estimates for the other chapters of this book, but show how to compute estimates for an SRS using the formulas so you can see that the two methods give the same numbers. Since R is a highly flexible language, you can also write your own functions to compute estimates using the formulas.

Examples 2.6, 2.7, and 2.11 of SDA. This section analyzes variables in data set `agsrs.csv`, described in Example 2.6 of SDA and in Appendix A of this book. The primary response variable for these examples is *acres92* (acreage devoted to farms in 1992).

First, we draw a histogram of the variable *acres92*, shown in Figure 2.1. The optional argument `breaks=20` tells R to use 20 bins for the histogram. The *col* (specifies color of bars), *xlab* (gives label for the x-axis), and *main* (specifies the title for the graph) arguments are also optional but make the picture look nicer.

```
# Draw a histogram
hist(agsrs$acres92,breaks=20,col="gray",xlab="Acres devoted to farms, 1992",
     main="Histogram: Number of acres devoted to farms, 1992")
```

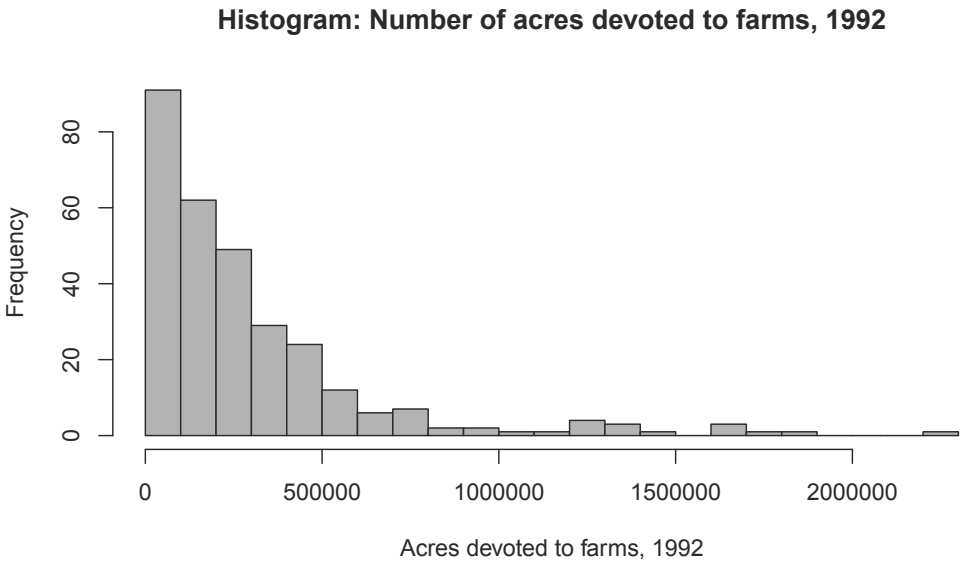

FIGURE 2.1: Histogram of farm acreage in 1992 (data *agsrs*).

Using the formulas in SDA. Statistics from an SRS can be computed using functions supplied in the base R package with the formulas in SDA. Functions such as *t.test* will compute confidence intervals without a finite population correction (fpc).

```
# Base R functions such as t.test will calculate statistics for an SRS,
# but without the fpc.
t.test(agsrs$acres92)
##
##  One Sample t-test
##
## data:  agsrs$acres92
## t = 14.975, df = 299, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  258749.6 337044.5
## sample estimates:
## mean of x
##    297897
```

The confidence interval from the *t.test* function is wider than that in Example 2.11 of SDA, however. Standard errors and confidence intervals that incorporate the fpc can be calculated directly with the formulas. If you are familiar with writing functions in R, you could also write your own function to do these calculations.

```
# Calculate the statistics by direct formula
n <- length(agsrs$acres92)
ybar <- mean(agsrs$acres92)
ybar
## [1] 297897
hatvybar<-(1-n/3078)*var(agsrs$acres92)/n
seybar<-sqrt(hatvybar)
seybar
## [1] 18898.43
# Calculate confidence interval by direct formula using t distribution
Mean_CI <- c(ybar - qt(.975, n-1)*seybar, ybar + qt(.975, n-1)*seybar)
names(Mean_CI) <- c("lower", "upper")
Mean_CI
##    lower    upper
## 260706.3 335087.8
# To obtain estimates for the population total,
# multiply each of ybar, seybar, and Mean_CI by N = 3078
seybar*3078
## [1] 58169381
Mean_CI*3078
##      lower      upper
##  802453859 1031400361
# Calculate coefficient of variation of mean
seybar/ybar
## [1] 0.06343948
```

Using functions in the *survey* package. Most of the statistics discussed in Chapter 2 can be computed in R using functions *svydesign*, *svymean*, and *svytotal* from the `survey` package (Lumley, 2020).

The data set *agsrs* does not contain a variable of sampling weights, so we need to create one. Also define the variable *lt200k*, which takes on value 1 if *acres92* $<$ 200,000 and the value 0 if *acres92* $\geq$ 200,000. The mean of variable *lt200k* estimates the proportion of farms that have fewer than 200,000 acres.

```
# Create the variable of sampling weights
n <- nrow(agsrs)
agsrs$sampwt <- rep(3078/n,n)
# Create variable lt200k
agsrs$lt200k <- rep(0,n)
agsrs$lt200k[agsrs$acres92 < 200000] <- 1
# look at the first 10 observations with column 3 (acres92) and column 17 (lt200k)
agsrs[1:10,c(3,17)]
##    acres92 lt200k
## 1   175209      1
## 2   138135      1
## 3    56102      1
## 4   199117      1
## 5    89228      1
## 6    96194      1
## 7    57253      1
```

```
## 8   210692      0
## 9    78498      1
## 10  219444      0
```

Now specify the survey design with function *svydesign*. The function has numerous optional arguments, so we call it using the variable names in the arguments. The arguments used for an SRS are:

id In general survey designs, *id* specifies the cluster identifiers. For an SRS we use `id = ~1`, which tells *svydesign* that there is no clustering. The tilde (~) is used by R to specify a formula, and the syntax for formulas will be discussed in later chapters.

weights Names the variable in the data frame that contains the sampling weights. The weights argument can actually be omitted for calculating means in an SRS (the function will calculate weights from the *fpc* argument if it is supplied, or set all weights equal to 1 if neither *weights* nor *fpc* is included), but it is good to get in the habit of using a weight variable, so we include it here.

fpc Information for calculating the finite population correction. For an SRS, we can use `fpc = rep(N, n)` with N the population size and n the sample size. For *agsrs*, $N = 3078, n = 300$, and `fpc = rep(3078, 300)`.

data Name of the data frame containing the variables to be analyzed.

```
# If you did not load the survey package at the beginning of the chapter, do it now
# library(survey)
# Specify the survey design.
# This is an SRS, so the only design features needed are the weights
#    or information used to calculate the fpc.
dsrs <- svydesign(id = ~1, weights = ~sampwt, fpc = rep(3078,300), data = agsrs)
dsrs
## Independent Sampling design
## svydesign(id = ~1, weights = ~sampwt, fpc = rep(3078, 300), data = agsrs)
```

When we print *dsrs*, we are told that this is an "Independent Sampling design"—that is, there is no stratification or clustering. We will come back to the function *svydesign* in later chapters as we encounter other survey designs.

Now that the survey design has been specified, we can calculate estimated means and totals using the *svymean* and *svytotal* functions. For each of these, the first argument contains the name(s) of the variable(s) to be analyzed, and the second argument is the name of the design object that was created by *svydesign*. The function *confint* will construct a 95% confidence interval (you can specify other confidence levels with the optional *level* argument, but we omit this since we always use 95% intervals in this book) using a t distribution with *df* degrees of freedom. If you do not specify the df, a normal distribution will be used for the confidence intervals. For an SRS, the t distribution has $n - 1$ df, where n is the sample size.

```
# Calculate the mean for acres92 and its standard error using the svymean function.
smean <- svymean(~acres92,dsrs)
smean
##           mean    SE
## acres92 297897 18898
# Use the confint function to compute a 95% confidence interval from
# the information in smean, df = n-1 = 300-1 = 299
confint(smean, df=299)
##            2.5 %   97.5 %
```

```
## acres92 260706.3 335087.8
# Repeat these steps with the svytotal function to obtain estimated totals.
stotal <- svytotal(~acres92,dsrs)
stotal
##             total       SE
## acres92 916927110 58169381
confint(stotal, df=299)
##             2.5 %     97.5 %
## acres92 802453859 1031400361
# Calculate the CV of the mean
SE(smean)/coef(smean)
##            acres92
## acres92 0.06343948
# or
smean<-as.data.frame(smean)
smean[[2]]/smean[[1]]
## [1] 0.06343948
```

You can analyze multiple variables at a time by putting them in a formula. The following code estimates the population means for variables *acres92* and *lt200k*.

```
# Estimate population means for multiple variables
agsrs_means <- svymean(~acres92+lt200k,dsrs)
agsrs_means
##               mean         SE
## acres92 297897.05 18898.4344
## lt200k       0.51     0.0275
confint(agsrs_means, df=299)
##                 2.5 %       97.5 %
## acres92 2.607063e+05 3.350878e+05
## lt200k  4.559508e-01 5.640492e-01
```

Estimating proportions from an SRS. For a binary numeric variable (taking on values 0 or 1), the estimated proportion is the mean of the variable, and the proportion of the population having $lt200k = 1$ is the mean of variable *lt200k*. From the above output, we can see that the value of $\hat{p} = 0.51$ is the estimated proportion where *lt200k* takes on the value 1. The standard error is 0.0275, and a 95% confidence interval of p is $[0.456, 0.564]$.

Sometimes you want to estimate the proportion of the population that falls in each of multiple categories. The variable *region* in the *agsrs* data describes the census region for each county in the sample and takes on the values "NE," "NC," "S," and "W". Running *svymean* with the variable *region* gives the estimated proportion in each category.

```
# Analyzing a categorical variable that is coded as characters
# First, display the category names and counts
table(agsrs$region)
##
##  NC  NE   S   W
## 107  24 130  39
# Find the estimated proportions in each category
region_prop <- svymean(~region,dsrs)
region_prop
##              mean     SE
## regionNC 0.35667 0.0263
## regionNE 0.08000 0.0149
## regionS  0.43333 0.0272
```

```
## regionW  0.13000 0.0185
confint(region_prop,df=299)
##                2.5 %    97.5 %
## regionNC 0.30487557 0.4084578
## regionNE 0.05066780 0.1093322
## regionS  0.37975605 0.4869106
## regionW  0.09363889 0.1663611
region_total <- svytotal(~region,dsrs)
region_total
##            total     SE
## regionNC 1097.82 81.005
## regionNE  246.24 45.878
## regionS  1333.80 83.799
## regionW   400.14 56.872
confint(region_total,df=299)
##              2.5 %    97.5 %
## regionNC  938.4070 1257.2330
## regionNE  155.9555  336.5245
## regionS  1168.8891 1498.7109
## regionW   288.2205  512.0595
```

Numeric and categorical variables. Numeric variables are variables for which you want to calculate statistics such as means (for example, *acres92* is a numeric variable). Categorical variables are those for which the values represent categories. Region is a categorical variable. We want to estimate the proportion of the population in each region, but we cannot calculate an "average" region. Here, *region* is automatically recognized as a categorical variable because it contains characters other than numbers.

Some surveys code categories as numbers; be careful to treat such variables as categorical rather than numeric. For example, a survey variable *haircolor* might take values 1–6, where 1 represents black, 2 represents brown, 3 represents blond, 4 represents red, 5 represents bald, and 6 represents other. You can calculate the mean for the variable *haircolor*, but it has no meaning. If you want to estimate the population proportion with each hair color, you can either (1) define binary variables for each category, for example, *redhair* = 1 if *haircolor* = 4 and 0 otherwise and find the mean of each variable, or (2) declare the variable *haircolor* to be categorical.

In R, you specify that a variable is categorical with the *factor* function. You can either declare the variable to be a factor variable in the data set or in the function call of *svymean*.

```
# Analyzing a categorical variable that is coded as numbers
# First, analyze lt200k as a numeric variable (works only if all values are 0 or 1)
# This gives the mean of variable lt200k, which is the proportion with lt200k = 1.
svymean(~lt200k,dsrs)
##         mean     SE
## lt200k 0.51 0.0275
# Now, analyze lt200k as a factor variable. This gives the proportion
# in each category
svymean(~factor(lt200k),dsrs)
##                 mean     SE
## factor(lt200k)0 0.49 0.0275
## factor(lt200k)1 0.51 0.0275
```

2.3 Additional Code for Exercises

This section contains additional code and references to functions used in three of the exercises in Chapter 2 of SDA. Some of these make use of advanced features of R; you should skip this section if you are new to R.

Exercise 2.27 of SDA asks you to estimate the sampling distribution of $\bar{y}$ by repeatedly taking samples of size n with replacement from the sample in *agsrs*, where y is the variable *acres92*. The following code constructs the histogram of the statistics from the bootstrap replicates that is shown in Figure 2.2. We use the *apply* function to calculate the mean value of *acres92* from each bootstrap replicate.

```
# Calculating bootstrap means for Exercise 2.27 in SDA
set.seed (244)
B = 1000
n = length(agsrs$acres92)
boot.samples = matrix(sample(agsrs$acres92, size = B * n, replace = TRUE),B, n)
boot.statistics = apply(boot.samples, 1, mean)
hist(boot.statistics, main="Estimated Sampling Distribution of ybar",
     xlab="Mean of acres92 from Bootstrap Replicate",
     col="gray",border="white")
```

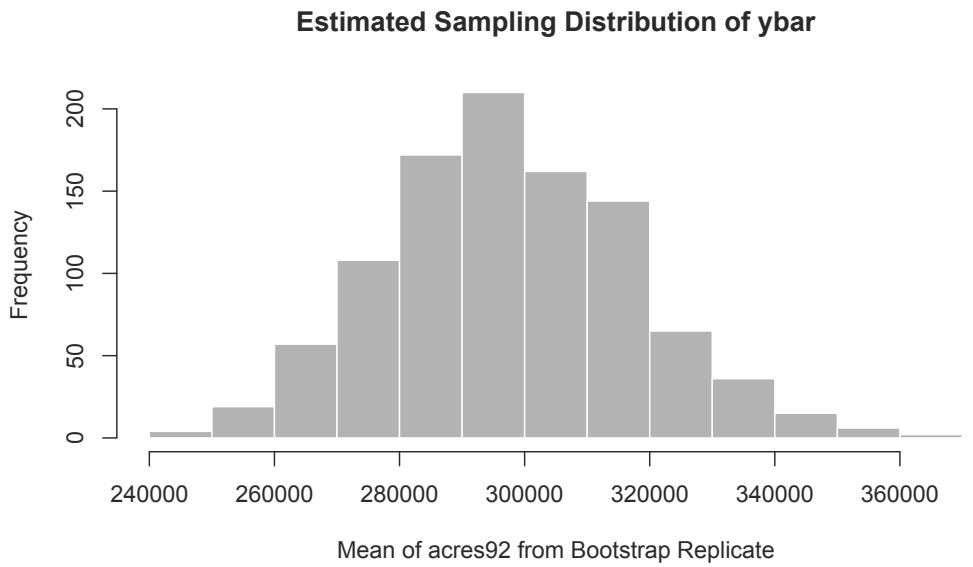

FIGURE 2.2: Histogram of means from bootstrap replicates.

Exercise 2.32 of SDA. The function *srswor1* in the `sampling` package will select an SRS using the algorithm in this exercise.

Exercise 2.34 of SDA. The function *binom.test* will calculate Clopper-Pearson confidence intervals for an SRS. For complex surveys, the function *svyciprop* in the `survey` package will calculate a variety of asymmetric confidence intervals for a proportion (see Section 3.4).

2.4 Summary, Tips, and Warnings

Table 2.1 lists the major functions used in this chapter to select or analyze data from an SRS. The base, graphics, stats, and utils packages are automatically included when you install R on your system.

TABLE 2.1
Functions used for Chapter 2.

Function	Package	Usage
sample	base	Select a simple random sample with or without replacement
mean	base	Calculate the mean of a vector
var	base	Calculate the variance of a vector
table	base	Compute the counts in each category of a vector
factor	base	Convert a vector to a factor object
set.seed	base	Initialize the seed for the random number generator
hist	graphics	Draw a histogram of data from an SRS
qt	stats	qt(.975,df) calculates the 0.975 quantile of a t distribution with the specified df
t.test	stats	Calculate a t confidence interval for an SRS (assumed to be with-replacement)
confint	stats	Calculate confidence intervals
data	utils	Loads the data set enclosed in parentheses; `data()` lists the available data sets
srswor	sampling	Select a simple random sample without replacement
srswr	sampling	Select a simple random sample with replacement
getdata	sampling	Extract the sampled data from a data frame containing the population units
svydesign	survey	Specify the survey design, for example, SRS
svymean	survey	Calculate mean and standard error of mean (if the variable is numeric), or proportion in each category (if variable is categorical)
svytotal	survey	Calculate total and standard error of total

Tips and Warnings

- If you want to be able to draw the same sample from a population at a later date, use the *set.seed* function before calling the sample-generating function.
- Although the *weights* argument is optional in the *svydesign* function for an SRS, it is a good practice to include it for computing estimates in *svymean* or *svytotal*. Most data sets from complex surveys come with weights that must be used to compute estimates correctly.
- When computing weights for an SRS, check that the sum of the weights equals the population size.
- If you are analyzing categorical variables with the *svymean* or *svytotal* functions, make sure to declare them to be *factor* variables. Otherwise, the functions will treat them as numeric variables and calculate the mean instead of computing the proportion in each category.