Chapter 1

One Dimensional Objects in R

1.1 Creating Objects

Consider the first time you saw a mathematical equation like $x = 3$; this says "let x equal 3." Later, we can use x like it is three, e.g. $2 \times x$ is 2×3 is 6. Often, we want to do the same thing in R.

In R, we have two ways of indicating we want to assign a object a value, using "<-" or "=." In the text, you will generally see us use "<-" for object assignment and "=" when specifying arguments to a function, which we'll discuss later.

Example 1.1. Assigning Objects in R. Below, we assign the expressions in Example 0.2 to objects in R.

```
> #Adding in R is easy
> a<-4+5 #Ask to add using +
> #Subtracting in R is easy, too
> b<-10-15 #Ask to subtract using -
> #Multiplying in R is easy, too
> c<-3*5 #Ask to multiply using *
> #Dividing in R is easy, too
> d<-27/3 #Ask to divide using /
> #Square roots in R are easy, too
> e<-sqrt(16) #Ask for a sqrt with sqrt()
> #We can do several calculations at once but we want to ensure that we're
> #careful with order of operations.
> f<-(3+5)*3/3^2 #Just squares the 3 according to PEMDAS
> g<-((3+5)*3/3)^2 #Squares the result of the expression in parenthesis
```

There are a few difference to notice in this code in comparison to Example 0.2. We're calculating the same values but this time the solution to the calculation isn't printed below in the console, instead the result is saved to a object. We can access the value of the objects by asking for them by name.

```
> a
[1] 9
```

```
> b
[1] -5
> c
[1] 15
> d
[1] 9
> e
[1] 4
> f
[1] 2.666667
> g
[1] 64
```

This tells us that R sees these letters as their corresponding values. We can ask for their values in the console as we did above, or we can consult the upper left portion of RStudio where the objects in the environment are listed; this is seen in Figure 1.1.1.

Remark: We will often find it useful to print an object as we assign it, instead of assigning a value to an object and then asking to print it by name. This can be done in R using one step; this is demonstrated below.

```
> (a<-4+5) #print an assigned variable by placing the line in parentheses
[1] 9
```

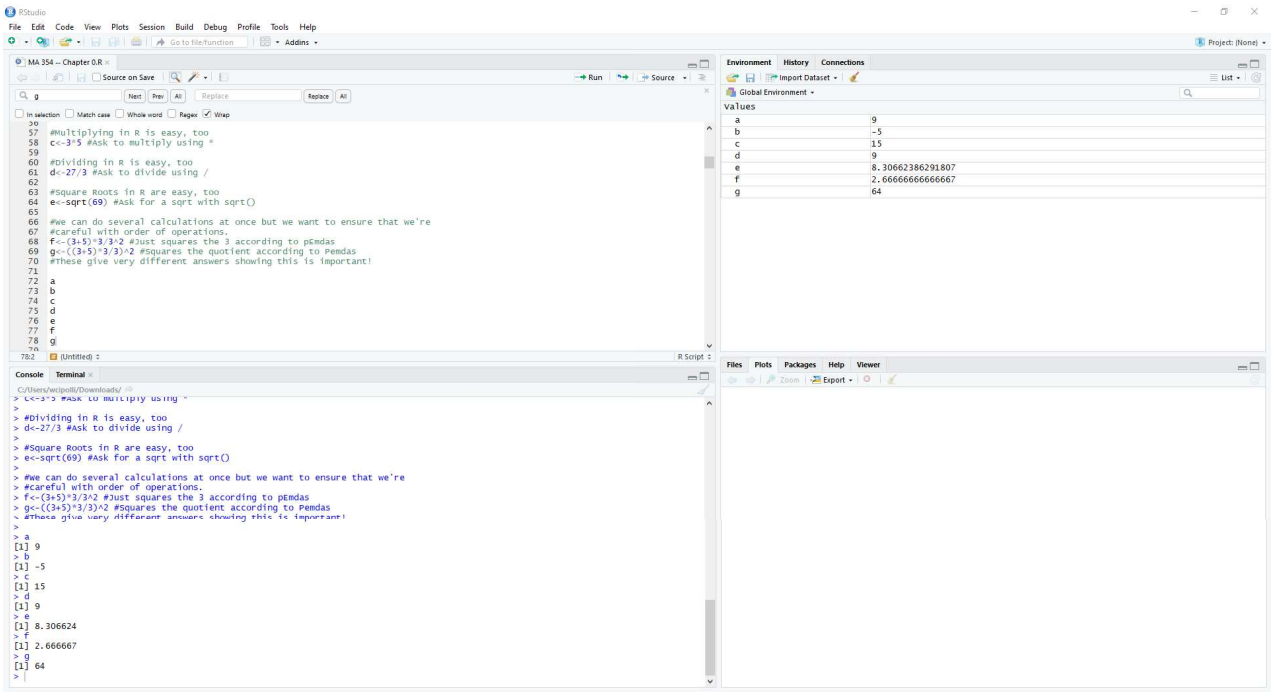

Figure 1.1.1: RStudio– After setting the objects in Example 1.1.

Since these objects are assigned numerical values, we can treat them like numbers; i.e. we can use all the algebraic operators we used above.

Example 1.2. Algebraic Calculations with Objects in R. Let's consider the object creation in Example 1.1 and do some calculations with those created objects. Here, we complete the calculations using the values stored in the objects we created.

```
> a+b
[1] 4
> c-d
[1] 6
> a*f
[1] 24
> e/g
[1] 0.129791
> sqrt(f)
[1] 1.632993
```

Example 1.3. Spotting an Error in R. Let's consider the object creation in Example 1.1, but this time there's a mistake.

```
> A
Error: object 'A' not found
```

While we know that we want the object `a`, or 9, `R` doesn't because we didn't ask for it by the right name. Recall, using a programming language requires us to ask precise questions. When considering object names we should consider capitalization – `R` is case sensitive.

Remark: We also can't use spaces or special characters (e.g., ^, &, %, +, ?). Again `R` has trouble deciphering when we want two words separated by a space to be a object name or recognizing one of the special characters is part of the name instead of the operation. If we want to create a space-like feature in our object names we can use a period or an underscore; e.g, `object.1` and `object_2` are valid object names in `R`.

Example 1.4. Spotting an Error in R. Let's consider the object creation in Example 1.1, but this time there's a mistake.

```
> 2a<-2*a
Error: unexpected symbol in "2a"
```

Objects names that start with numbers are not allowed in `R`; while `2a` is not a valid object name, `twoa`, and `a2` are.

Example 1.5. Spotting an Error in R. Let's consider the object creation in Example 1.1, but this time there's a mistake.

```
> af
Error: object 'af' not found
```

This is similar to the error seen in Example 0.4 where common notation on paper creates an error in `R`. We might know that writing af indicates that we want to multiply those two values but `R` does not, using $*$ is necessary.

Anytime you create a new object in `R` it is stored until you end your session, after which it will removed until you rerun your code. For example, if I were to close my `R` session and start a new session, `R` won't know what `a` is or anything else we specified so far, unless we reran the code.

1.2 Removing Objects from the Environment

There will be times we'll want to remove objects from our R session, this is particularly helpful when testing code; e.g., to show the code still works if I start from an empty R session.

```
> ls()          # lists all the object a from the session
[1] "a" "b" "c" "d" "e" "f" "g"
> rm("a")       # removes the object a from the session
> rm(list=ls()) # removes all objects in the session
```

Table 1.2.1 summarizes the operators used in this subsection.

Operator	Functionality	Example From Notes
=	sets an object equal to a value	a=4+5
<−	works as an equal sign	a<−4+5
ls()	lists objects in the session	ls()
rm()	removes object(s) from the session	rm("a")

Table 1.2.1: A list of basic operators for objects in a R session.

1.3 Data Types

As we make full use of R, we will use a variety of data types and convert between them. Table 1.3.2 summarizes the four main data types of R.

Data Type	Description	Example
character	any character or word (string of characters) input	a<−"hello"
numeric	any real number (integer or double)	a<−2
logical	a binary datatype with possible values TRUE and FALSE	a<−TRUE, or a<−FALSE
complex	complex numbers with real and imaginary parts	a<−1+4i

Table 1.3.2: A list of basic data types for objects in a R session.

Integers and doubles are two popular classes of data that are grouped together as numeric data. R will automatically convert between numeric classes in most cases where it is necessary, but most calculations are done using doubles, so that is often the default treatment of numeric data, unless otherwise specified. We will discuss the factor class of data extensively after we introduce more advanced objects in the following Chapter. Table 1.3.3 summarizes the three main classes of data in R.

Data Type	Description	Example
factor	data stored as integers, and have character labels associated with these unique integers.	
integer	a whole number (integer)	a< -2L
double	a floating point number (includes integers)	a< -3.3337

Table 1.3.3: A list of classes of data types for objects in a `R` session.

Sometimes, we will need to convert data types. This can largely be completed with the `as.datatype()` function – replacing `datatype` for the desired resulting datatype; e.g., `as.character()`, `as.factor()`, `as.numeric()`. We can ensure that an object has the desired data type by asking the question with the `is.datatype()` function – replacing `datatype` with datatype in question; e.g., `is.character()`, `is.factor()`, `is.numeric()`.

Example 1.6. Below we demonstrate several of the functions described above to ask about and convert the data types of objects in `R`.

```
> string<-"45" #starting with a character object
> as.numeric(string)
[1] 45
> is.character(string)
[1] TRUE
> is.numeric(as.numeric(string))
[1] TRUE
> is.double(as.numeric(string))
[1] TRUE
> is.integer(as.numeric(string))
[1] FALSE
> logical <-TRUE #starting with a logical object
> as.numeric(logical) #TRUE maps to 1
[1] 1
> as.numeric(FALSE)   #FALSE maps to 0
[1] 0
> is.logical(logical)
[1] TRUE
> is.numeric(as.numeric(logical))
[1] TRUE
> is.double(as.numeric(logical))
[1] TRUE
> is.integer(as.numeric(logical))
[1] FALSE
> complex<-4+5i #starting with a complex object
> is.numeric(complex)
[1] FALSE
> is.complex(complex)
[1] TRUE
> int<- 7L #starting with an integer object
> is.numeric(int)
[1] TRUE
```

```
> is.integer(int)
[1] TRUE
> is.double(int)
[1] FALSE
> is.integer(as.numeric(int))
[1] FALSE
> is.double(as.numeric(int))
[1] TRUE
> as.character(int)
[1] "7"
> as.complex(int)
[1] 7+0i
```

Above, we can see how easy it is to ask about and convert data types. Thinking about data types will be important as we deal with more complicated objects and data in R.

Example 1.7. Spotting an Error in R. We can ask R to change the data type of objects, but that doesn't mean it's always a good idea. It's important to consider the data being converted, and the behavior of the conversion.

```
> string<-"Changing Data Types in R"     #string (non-number)
> as.numeric(string)
[1] NA
Warning message:
NAs introduced by coercion
```

Note that a character string that has no straightforward way of being converted to a number yields a warning message.

Instead of asking for them one-by-one, as described above, we can ask for the data type directly. There are several ways to ask this question – `class()`, `mode()`, and `typeof()`. The `mode()` function returns the a mutually exclusive data type; e.g., all numbers are reported as numeric whether the data are of integer or double types. The `typeof()` function returns the non-mutually exclusive data types; e.g., type of will return the less abstract data types of integer or double when applicable. The `class()` function returns the data type of of the object that determines how functions are applied.

```
> x<-1
> c(class(x),mode(x),typeof(x))
[1] "numeric" "numeric" "double"
> x<-"Hello"
> c(class(x),mode(x),typeof(x))
[1] "character" "character" "character"
> x<-FALSE
> c(class(x),mode(x),typeof(x))
[1] "logical" "logical" "logical"
> x<-7L
> c(class(x),mode(x),typeof(x))
[1] "integer" "numeric" "integer"
```

```
> x<-4+3i
> c(class(x),mode(x),typeof(x))
[1] "complex" "complex" "complex"
```

Remark: Here, I used the `c()` command to "combine" the output of the three functions that return data types and classes in `R`. This allows me to save some space and compare them on one line. We describe this type of object in the next section.

1.4 Vectors

1.4.1 Creating Vectors

In statistics we will work to understand the world by gathering, organizing and analyzing information. In general, such information can be provided by numbers or words and when completing a statistical analysis we will generally have information for a selection of people or things, termed a sample; e.g., the amount of savings in dollars reported by a sample of 350 randomly selected American adults. Since we will ask questions about these types of information, we will want to store the collection of information as an object.

To do this in `R`, we create a vector, which is just a fancy word for an array of information; we call each observation an element. In Example 1.8, we demonstrate a few ways to create a vector in `R`.

Example 1.8. Creating a Vector in R.

```
#c() allows us to define a vector of numbers in R
> c(1,4,3,2,5,2,6,7,8)
[1] 1 4 3 2 5 2 6 7 8
#c() allows us to define a vector of words in R
> c("Yes","No","No", "No", "Yes", "Yes", "No")
[1] "Yes" "No"  "No"  "No"  "Yes" "Yes" "No"
#seq() creates vector of an ordered sequence -- this saves us timef
> seq(from=0,to=10,by=2)
[1]  0  2  4  6  8 10
#This is shorthand for an ordered sequence with by=1
> 2:10
[1]  2  3  4  5  6  7  8  9  10
#rep() creates a vector of repeated values
> rep(3,5)
[1] 3 3 3 3 3
```

The ability to create a vector using "seq" and "rep" means that we can save a lot of time. While it may be simple to type "c(0,2,4,6,8,10)," it would take a considerable amount of time to do this if I wanted the even numbers between zero and one-hundred or zero and one-thousand.

Example 1.9. Spotting an Error in R. Let's consider the vector creation in Example 1.8, but this time there's a mistake.

```
> c(1,4,3,2,1,5,2,6,,8)
Error in c(1, 4, 3, 2, 5, 2, 6, , 8) : argument 8 is empty
```

We've run the code but R returns an error in red. We see that the text of the error suggests that we left one of the positions of the vector as empty. We can fix this error by putting the missing value in or, if it is a truly missing value, we can enter NA to denote that the value is not available.

Example 1.10. Spotting an Error in R. Let's consider the vector creation in Example 1.8, but this time there's a different mistake.

```
> c("Yes","No",No, "No", "Yes", "Yes", "No")
Error: object 'No' not found
```

We've run the code but R returns an error in red. We see that the text of the error suggests that the object 'No' wasn't found. In this case, we see that we forgot quotations around No. Without the quotations, R thinks that we are referring to an object that we've saved as No; we can't ask for objects we haven't yet created. We can fix this error by simply putting the missing quotations around No.

Example 1.11. Assigning Vectors to Objects in R. We can name and save vectors as objects in R using the same notation in Example 1.11.

```
#c() allows us to define a vector of numbers in R
> g<-c(1,4,3,2,5,2,6,7,8)
#c() allows us to define a vector of words in R
> h<-c("Yes","No","No","No","Yes","Yes","No")
#seq() creates vector of an ordered sequence -- this saves us time
> i<-seq(from=0,to=10,by=2)
#rep() creates a vector of repeated values
> j<-rep(3,5)
```

1.4.2 Calculations with Vectors

Example 1.12. Calculations with Vectors in R. Below, we demonstrate how R completes calculations with vectors using those created in Example 1.11.

```
> g  #Starting Vector
[1] 1 4 3 2 5 2 6 7 8
#+4 adds 4 to each item in the vector
> g+4
[1]  5  8  7  6  9  6 10 11 12
#-3 subtracts 3 from each item in the vector
> g-3
[1] -2  1  0 -1  2 -1  3  4  5
#*10 multiplies each item in the vector by 10
> g*10
[1] 10 40 30 20 50 20 60 70 80
#/2 divides each item in the vector by 2
> g/2
[1] 0.5 2.0 1.5 1.0 2.5 1.0 3.0 3.5 4.0
#We can add, subtract, multiple and divide using vectors too,
#but to do so we usually want to ensure that they are of the same length.
```

```
#We can ask for the length of the vector by simply asking for it.
> length(g) #returns the length of g
[1] 9
> m<-c(1,2,3,4,5,6,7,8,9) #could use 1:9 or seq(from=1,to=9,by=1)
> g+m #Adds corresponding items in g and m
[1]  2  6  6  6 10  8 13 15 17
> g-m #Subtracts corresponding items in g and m
[]1]  0  2  0 -2  0 -4 -1 -1 -1
> g*m #Multiplies corresponding items in g and m
[1]  1  8  9  8 25 12 42 56 72
> g/m #Divides corresponding items in g and m
[1] 1.0000000 2.0000000 1.0000000 0.5000000 1.0000000 0.3333333 0.8571429
[8] 0.8750000 0.8888889
>  g%*%m #Dot product of two vectors
[,1]
[1,]  233
```

There are several important notions in this example that we should keep in mind when we complete calculations with vectors in R.

1. We usually want to ensure that they are of the same length (see Example 1.13).
2. We can check the length of a vector using the length() command.
3. When we have vectors that are too long R will print the vector on multiple lines; at the beginning of each line R will print the position of the first element on that line in brackets.

Example 1.13. Spotting an Error in R. Let's consider completing a calculation from Example 1.12, but this time there's a mistake.

```
> g #Starting Vector
[1] 1 4 3 2 5 2 6 7 8
> length(g) #length of g
[1] 9
> n<-c(1,2,3) #Note the length of n (3) is a multiple of 9,
> p<-c(1,2,3,4) #the lenghth of p (4) is not.
> g+n #this works by repeating n three times
[1]  2  6  6  3  7  5  7  9 11
#The result above is g+(1,2,3,1,2,3,1,2,3).
#Note that because it works doesn't mean we necessarily wanted
#to ask for such a calculation. In fact, we rarely want to complete
#such a calculation. Errors without error messages are the worst kind.
> g+p #this gives a warning
[1]  2  6  6  6  6  4  9 11  9
Warning message:
In g + p : longer object length is not a multiple of shorter object length
#The result above is g+c(1,2,3,4,1,2,3,4,1).
```

Note that because the calculations work doesn't mean we necessarily wanted to ask for such a calculations; in fact, we will rarely want to complete such a calculations. Using a programming language requires us to ask precise questions – R will answer any question we ask if it can, but it can't always catch when we ask a bad question. Errors without error messages are the worst kind.

1.4.3 Subsetting Vectors

We're creating the same vectors from Example 1.8 but this time the vector isn't printed below in the console, but instead saved to a object. We can then access these vectors by asking for them by name, and we can ask for particular items within the vector by asking for it by position.

```
> g
[1] 1 4 3 2 5 2 6 7 8
> h
[1] "Yes" "No"  "No"  "No"  "Yes" "Yes" "No"
> i
[1]  0  2  4  6  8 10
> j
[1] 3 3 3 3 3
#We can ask for the 3rd value in vector h
> h[3]
[1] "No"
#We can ask for the first three values in vector i
> i[c(1,2,3)]
[1] 0 2 4
> i[seq(from=1,to=3,by=1)]
[1] 0 2 4
> i[1:3]
[1] 0 2 4
#We can ask for all but the first value in vector g
> g[-1]
[1] 4 3 2 5 2 6 7 8
#We can ask for all but the first 3 values in vector i
> i[-c(1,2,3)]
[1]  6  8 10
> i[-seq(from=1,to=3,by=1)]
[1]  6  8 10
> i[-(1:3)]
[1]  6  8 10
#We can ask for the elements of i in ascending order
> i[order(i)]
[1]  0  2  4  6  8 10
#We can ask for the elements of i in decending order
> i[order(i,decreasing=TRUE)]
[1] 10  8  6  4  2  0
#We can combine elements of g and i
> k<-c(g,i)
> k
[1]  1  4  3  2  5  2  6  7  8  0  2  4  6  8 10
```

This tells us that R sees these letters as their corresponding vectors. We can ask for their values in the console as we did above or we can consult the upper left portion of RStudio where the objects in the environment are listed; this is seen in Figure 1.4.2. We can see that R treats h, j, and k as vectors of numbers and i as a vector of characters, or words.

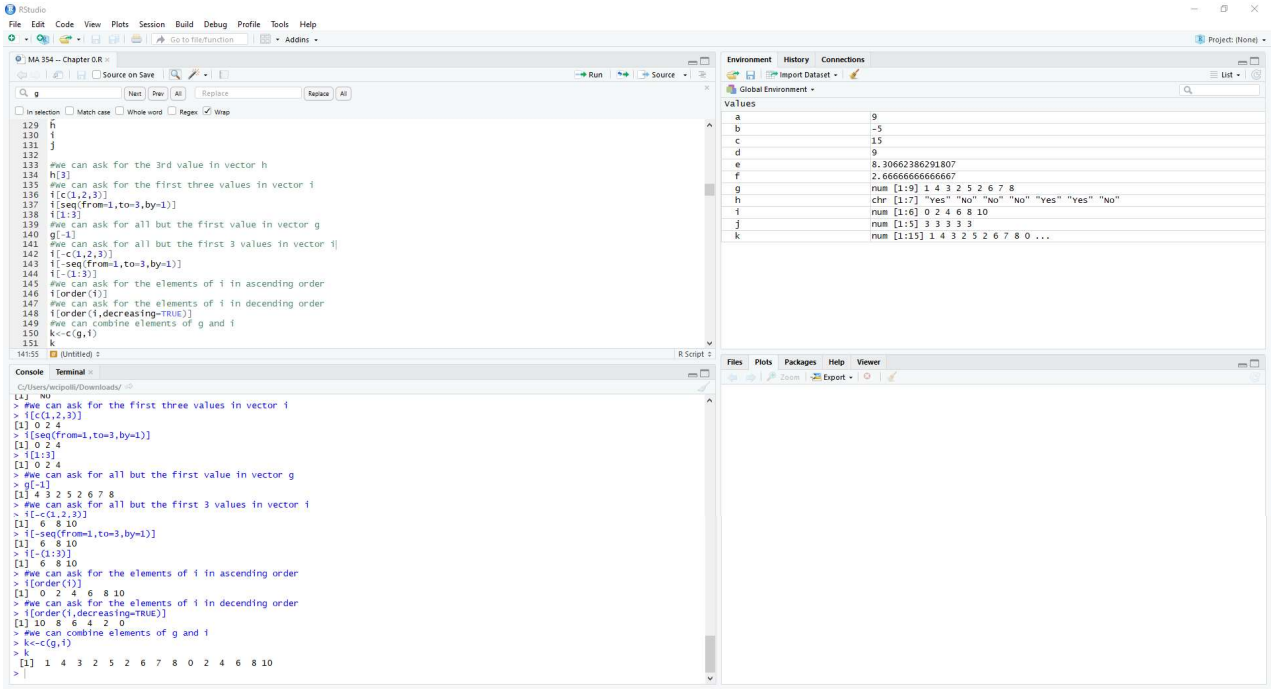

Figure 1.4.2: RStudio– After setting the objects in Example 1.11.

Example 1.14. Spotting an Error in R. Let's discuss asking for the first observation from a vector created in Example 1.11.

```
> k
[1]  1  4  3  2  5  2  6  7  8  0  2  4  6  8 10
> k[0]
numeric(0)
> k[1]
[1] 1
```

It is important to realize that all indexing in R is base-one, unlike many other programming languages that are base zero. It is particularly important because no warning or error message is thrown if we try to access the zero-th element.

Example 1.15. Spotting an Error in R. Let's consider what happens when we ask for observations that go beyond the end of a vector created in Example 1.11.

```
> k
[1]  1  4  3  2  5  2  6  7  8  0  2  4  6  8 10
> k[15]
[1] 10
> k[16]
[1] NA
```

Interestingly, the issue of no warning or error message occurs on the other side of the vector as well. In this case, when we ask for observations beyond the end of the vector, R returns NA denoting that the information we asked for is not available.

Example 1.16. Asking Questions About Vectors in R. There are many questions we might want to ask about a vector in R.

```
#See the first few observations
> head(g)
[1] 1 4 3 2 5 2
#How many elements are in the vector?
> length(g)
[1] 9
#What are the unique elements in g?
> unique(g)
[1] 1 4 3 2 5 6 7 8
#How many unique elements are in the vector?
> length(unique(g))
[1] 8
#Provide a summary of (numerical) observations
> summary(g)
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
1.000   2.000   4.000   4.222   6.000   8.000
#Provide a summary of (categorical) observations
> table(h)
h
No Yes
4   3
#Table also works for numerical vectors with a handful of unique observations
> table(g)
g
1 2 3 4 5 6 7 8
1 2 1 1 1 1 1 1
```

1.4.4 The which() Function

The `which()` function is a very important function in R that tells us the positions belonging to elements that satisfy a specified criteria. We will find this particularly useful when we want to create new objects from, or ask questions about subsets of data. To complete these tasks, we will have to understand how to ask for what we want, precisely. Table 1.4.4 provides a list of common questions and the syntax necessary to ask them.

Operator	Functionality
$x < y$	is x less than y
$x <= y$	is x less than or equal to y
$x > y$	is x greater than y
$x > y$	is x greater than or equal to y
$x == y$	is x equal to y
$x! = y$	is x not equal to y
$x\%in\%y$	is x an element of y
cond1&cond2	logical and – both conditions must be true to return true
cond1&&cond2	logical and – both conditions must be true to return true if the first condition is FALSE the second isn't checked
cond1|cond2	logical or – at least one conditions must be true to return true
cond1||cond2	logical or – at least one conditions must be true to return true if the first condition is TRUE the second isn't checked
!cond	logical not – the condition must be false to return true
is.na(x)	is x NA – a missing value, (or nan)
is.nan(x)	is x not a number – i.e., 0/0
is.null(x)	is x NULL –an empty object

Table 1.4.4: A list of logical operators (booleans) that can be used with the `which`() function in `R`.

Below, we use `which()` to ask for the position of elements that satisfy several conditions using logical operators from Table 1.4.4.

```
> g
[1] 1 4 3 2 5 2 6 7 8
#Which elements in g are less than 3?
> which(g<3)
[1] 1 4 6
#Which elements in g are less than or equal to 3?
> which(g<=3)
[1] 1 3 4 6
#Which elements in g are greater than 3?
> which(g>3)
[1] 2 5 7 8 9
#Which elements in g are greater than or equal to 3?
> which(g>=3)
[1] 2 3 5 7 8 9
#Which elements in g are equal to 2?
> which(g==2)
[1] 4 6
```

```
#Which elements in g are not equal to 2?
> which(g!=2)
[1] 1 2 3 5 7 8 9
#Which elements in g are in the set (1,2,3,4,5)?
> which(g %in% c(1,2,3,4,5))
[1] 1 2 3 4 5 6
#Which elements in g are less than or equal to 6 AND less greater than 2?
> which(g<=6 & g>2)  #Think 2<g<=6
[1] 2 3 5 7
#Which elements in g are less than or equal to 6 OR less greater than 2?
> which(g<=6 | g>2)  #At least one of these is true: 2<g or g<=6 or 2<g<=6
[1] 1 2 3 4 5 6 7 8 9
#Which elements in g are not in the set (1,2,3,4,5)
> which(!(g %in% c(1,2,3,4,5)))
[1] 7 8 9
```

Knowing the position of of elements that satisfy a condition of interest can be helpful for asking other questions about the data.

```
#Which elements in g are equal to 2?
> which(g==2)
[1] 4 6
#How many elements are in the vector are equal to 2?
> length(which(g==2))
[1] 2
#This returns the two 2 elements
> g[which(g==2)]
[1] 2 2
#This also returns the two 2 elements
> subset(x=g,subset=g==2)
[1] 2 2
```

Note that we can use logical statements in `R` to subset a vector by asking for those observations by index using `which()` or using the `subset()` function. Breaking down each piece, we read the code as,

g	'the elements of g, g[i]'
[	'for'
which(	'which'
g==2)]	'g[i] equals 2.'

There are also functions that allow us to ask about empty objects, missing observations, and other problematic observations.

```
#Create a blank object called q
> q<-NULL
#Is q an empty object?
```

```
> is.null(q)
[1] TRUE
#Save a vector to object q
> q<-c(0,2,3,5,7,11,NA,13)
#Is q an empty object?
> is.null(q)
[1] FALSE
#Are there missing observations?
> which(is.na(q))
[1] 7
#is.na also catches nan
> q/0
[1] NaN Inf Inf Inf Inf Inf  NA Inf
> which(is.na(q/0))
[1] 1 7
> which(is.nan(q/0))
[1] 1
```

Example 1.17. Spotting an Error in R. What happens when we ask for something that doesn't exist?

```
#Which elements in g are equal to 2?
> g[which(g==22)]
numeric(0)
> subset(x=g,subset=g==22)
numeric(0)
```

We've asked for the positions the elements in g that are equal to 22. There aren't any. While there is no obvious error message in red, we get an unexpected outcome – numeric(0) isn't a position between 1 and 9, the length of g. The output numeric(0) indicates that a vector of length 0, an empty vector, is the response – this makes sense because there are zero elements in g that fit the condition. Although this is not an error, this can (and will) be a problem when we accidentally ask for something that doesn't exist in a larger block of code.

1.4.5 Describing Vectors in R

Example 1.18. Describing Vectors in R. There are many built-in functions in R that are helpful to describe vectors in R.

```
#What is the minimum element in g?
> min(g)
[1] 1
#What is the maximum element in g?
> max(g)
[1] 8
#What is the sum of all elements in g?
> sum(g)
[1] 38
#Calculate a cumulative or "running" total.
```

```
> cumsum(g)
[1]  1  5  8 10 15 17 23 30 38
#What is the average element in g?
> mean(g)
[1] 4.222222
#What is the standard deviation of elements in g?
> sd(g)
[1] 2.438123
#What are the quantiles of elements in g?
> quantile(g)
0%  25%  50%  75% 100%
1    2    4    6    8
#What is the 90th percentile of elements in g?
> quantile(g,probs= c(0.90))
90%
7.2
```

Example 1.19. Spotting an Error in R. What happens when we ask for a vector summary for a vector with a missing value? Below, we see that the questions we asked in Example 1.18 don't provide numerical summaries as we might expect.

```
> q #Starting Vector
[1]  0  2  3  5  7 11 NA 13
#What is the minimum element in q?
> min(q)
[1] NA
#What is the maximum element in q?
> max(q)
[1] NA
#What is the sum of all elements in q?
> sum(q)
[1] NA
#Calculate a cumulative or "running" total.
> cumsum(q)
[1]  0  2  5 10 17 28 NA NA
#What is the average element in q?
> mean(q)
[1] NA
#What is the standard deviation of elements in q?
> sd(q)
[1] NA
#What are the quantiles of elements in q?
> quantile(q)
Error in quantile.default(q) :
missing values and NaN's not allowed if 'na.rm' is FALSE
#What is the 90th percentile of elements in q?
> quantile(q,probs=c(0.90))
Error in quantile.default(q, probs = c(0.9)) :
missing values and NaN's not allowed if 'na.rm' is FALSE
```

As suggested in the error messages of the `quantile()` function, we can try these functions by

specifying `na.rm=TRUE`. This runs the code by removing the missing observation and calculating the summaries as if that missing observation were removed from the vector.

```
> q #Starting Vector
[1]  0  2  3  5  7 11 NA 13
#What is the minimum element in q?
> min(q,na.rm=TRUE)
[1] 0
#What is the maximum element in q?
> max(q,na.rm=TRUE)
[1] 13
#What is the sum of all elements in q?
> sum(q,na.rm=TRUE)
[1] 41
#Calculate a cumulative or "running" total.
> cumsum(q,na.rm=TRUE)
Error in cumsum(q, na.rm = TRUE) :
2 arguments passed to 'cumsum' which requires 1
#What is the average element in q?
> mean(q,na.rm=TRUE)
[1] 5.857143
#What is the standard deviation of elements in q?
> sd(q,na.rm=TRUE)
[1] 4.775932
#What are the quantiles of elements in q?
> quantile(q,na.rm=TRUE)
0%  25%  50%  75% 100%
0.0  2.5  5.0  9.0 13.0
#What is the 90th percentile of elements in q?
> quantile(q,probs= c(0.90),na.rm=TRUE)
90%
11.8
```

We see that specifying `na.rm=TRUE`, yields numerical summaries (except for `cumsum()`) by treating `q` as the vector `q<-c(0,2,3,5,7,11,13)`.

The `cumsum()` function is not written to accept the `na.rm=TRUE` input – this is made clear by the error message suggesting that the function only requires 1 input. If we truly want the cumulative sum, ignoring the missing value, we have to ignore the missing observation ourselves – we complete this task in two ways below.

```
#Starting Vector
[1]  0  2  3  5  7 11 NA 13
> q.obs<-q[which(!is.na(q))] #keep observations that are not missing
#q.obs<-subset(x=q,subset=!is.na(q)) is equivalent
> q.obs
[1]  0  2  3  5  7 11 13
> q.obs<-q[-which(is.na(q))] #remove observations that are missing
> q.obs
[1]  0  2  3  5  7 11 13
```

```
> cumsum(q.obs)
[1]  0  2  5 10 17 28 41
```

1.4.6 Comparing Vectors in R

Example 1.20. Comparing Vectors in R. There are many questions we might want to ask to compare vectors in R. We demonstrate several base functions that allow such comparisons below.

```
#Which elements do g and i share?
> intersect(g,i)
[1] 4 2 6 8
#Which elements do g and i have combined?
> union(g,i)
[1]  1  4  3  2  5  6  7  8  0 10
#Which elements from g are not in i?
> setdiff(g,i)
[1] 1 3 5 7
#Which elements from i are not in g?
> setdiff(i,g)
[1]  0 10
#Which elements from i are not in g?
```

We can also use the %in% command to ask other questions about vectors in R.

```
> g%in%i
[1] FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE
> any(g%in%i)#Are any elements of g also in i?
[1] TRUE
> all(g%in%i)#Are all elements of g in i? Is g a subset of i?
[1] FALSE
> which(g%in%i)#What are the positions of elements of g which are also in i?
[1] 2 4 6 7 9
> g[which(g%in%i)]#What are the elements of g which are also in i?
[1] 4 2 2 6 8
> subset(x=g,subset=g%in%i)#What are the elements of g which are also in i?
[1] 4 2 2 6 8
```

1.4.7 Vector Data Types

Recall Tables 1.3.2 and 1.3.3, where we described the data types and classes of objects in R. There are four types of vectors in R, the same as objects – character, numeric, logical, and complex. It is important to note that vectors are atomic, which means that vectors can only be made up of objects of the same type.

Example 1.21. Below, we create vectors consisting of all the combinations of these data types. In each case, we see that the printed vector converts all the data to one data type – this is observable as the vector is printed and by asking for the data type of the vector the same way we would for a simple vector.

```
> (x<-c(1,3+1i))
[1] 1+0i 3+1i
> c(class(x),mode(x),typeof(x))
[1] "complex" "complex" "complex"
> (x<-c(1,"hi"))
[1] "1"  "hi"
> c(class(x),mode(x),typeof(x))
[1] "character" "character" "character"
> (x<-c(1,TRUE))
[1] 1 1
> c(class(x),mode(x),typeof(x))
[1] "numeric" "numeric" "double"
> (x<-c(3+1i,TRUE))
[1] 3+1i 1+0i
> c(class(x),mode(x),typeof(x))
[1] "complex" "complex" "complex"
> (x<-c(TRUE,"hi"))
[1] "TRUE" "hi"
> c(class(x),mode(x),typeof(x))
[1] "character" "character" "character"
> (x<-c(3+1i,"hi"))
[1] "3+1i" "hi"
> c(class(x),mode(x),typeof(x))
[1] "character" "character" "character"
> (x<-c(1,3+1i,"hi"))
[1] "1"    "3+1i" "hi"
> c(class(x),mode(x),typeof(x))
[1] "character" "character" "character"
> (x<-c(1,3+1i,TRUE))
[1] 1+0i 3+1i 1+0i
>c(class(x),mode(x),typeof(x))
[1] "complex" "complex" "complex"
> (x<-c(1,TRUE,"hi"))
[1] "1"    "TRUE" "hi"
> c(class(x),mode(x),typeof(x))
[1] "character" "character" "character"
> (x<-c(3+1i,TRUE,"hi"))
[1] "3+1i" "TRUE" "hi"
> c(class(x),mode(x),typeof(x))
[1] "character" "character" "character"
> (x<-c(1,3+1i,"hi",TRUE))
[1] "1"    "3+1i" "hi"   "TRUE"
> c(class(x),mode(x),typeof(x))
[1] "character" "character" "character"
```

The coercion that occurs when we create vectors is automatic. While we didn't receive any warning or error messages here, `R` generally will warn you if you might lose information.

A factor is an important class of vector that can only contain predefined values called levels; these values are of type character (categorical) but stored as integers which map to the character labels.

Factors have a specific behavior that we have to be conscious of.

Example 1.22. Factors are treated difference in `summary()` and other base functions we use to describe vectors. Below, we apply the `summary()` function to a character vector created in 1.8 as well as the factor vector created using the `as.factor()` function.

```
> h
[1] "Yes" "No"  "No"  "No"  "Yes" "Yes" "No"
> (h.factor<-as.factor(h))
[1] Yes No  No  No  Yes Yes No
Levels: No Yes
> nlevels(h.factor)
[1] 2
> summary(h)
Length     Class      Mode
7 character character
> summary(h.factor)
No Yes
4   3
```

Example 1.23. Spotting an Error in R. Sometimes treating a vector as a factor vector can create some problematic effects. For example, below, we will convert a numeric vector to a factor vector; we might want to do this if there are few unique elements in the vector and the numbers represent categories.

```
> k
[1]  1  4  3  2  5  2  6  7  8  0  2  4  6  8 10
> (k.factor<-as.factor(k))
[1] 1  4  3  2  5  2  6  7  8  0  2  4  6  8  10
Levels: 0 1 2 3 4 5 6 7 8 10
> summary(k)
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.0     2.0     4.0     4.5     6.5    10.0
> summary(k.factor)
0  1  2  3  4  5  6  7  8 10
1  1  3  1  2  1  2  1  2  1
```

We see that instead of providing a numerical summary like the mean, when we ask for a summary of the factor vector we get a table consisting of the frequency of each unique observation.

An issue that can arise if we aren't careful is that we might mistakenly ask `R` the wrong question. For example, below we want the sum of the vector `k`. If we ask for the sum of the numeric vector, we get the correct result. If we ask for the sum of the factor vector, we get an error – `R` doesn't calculate the sum for factors.

```
> sum(k)
[1] 68
> sum(k.factor)
Error in Summary.factor(c(2L, 5L, 4L, 3L, 6L, 3L, 7L, 8L, 9L, 1L, 3L,  :
'sum' not meaningful for factors
```

To get around this error, we might use the `as.numeric()` function.

```
> sum(as.numeric(k.factor))
[1] 82
```

We get a very different solution when we ask `R` to take the sum of a numeric-converted factor vector. This is because the `as.numeric()` does not convert the factor vector back to the original vector, but to the integers that represent the categories.

```
> k
[1]  1  4  3  2  5  2  6  7  8  0  2  4  6  8 10
> as.numeric(k.factor)
[1]  2  5  4  3  6  3  7  8  9  1  3  5  7  9 10
> levels(k.factor)
[1] "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "10"
> as.numeric(as.character(k.factor))
[1]  1  4  3  2  5  2  6  7  8  0  2  4  6  8 10
```

Above, we see that the first element of the converted factor vector is 2. This value is 2 because the first element of `k` is 1 and that is the second level – 2 refers to which level the first element belongs. Notice, however, if we first convert the factor vector to a character vector, we can successfully apply the `as.numeric()` function.

This will be important when we read data from a file. Sometimes, data we would like to treat as numeric is treated as a character or factor vector. This is caused by a non-numeric value in the data or a code for missing values. To remedy the situation, we can try to pre-process the data or coerce the vector keeping the behavior above in mind.

1.4.8 Lists

Lists are similar to vectors in that they are a collection of items, however a list is not restricted to a single data type; thinking of lists as a generic vector is helpful because the elements of a list can be of any data type or class of `R` object, even other lists.

Example 1.24. Below, we create vectors consisting of several data types using the `list()` function, just as we used `c()` to create a vector. We see that when using a list, the data are not converted to one type as they were when we used a vector – this is observable as the list is printed and by asking for the data type of the elements the same way we would for a simple vector. **Remark:** When using a list, we use double brackets instead of single brackets as we did with vectors.

```
> (x<-list(1,3+1i,"hi",TRUE))
[[1]]
[1] 1

[[2]]
[1] 3+1i

[[3]]
[1] "hi"
```

```
[[4]]
[1] TRUE
> c(class(x),mode(x),typeof(x))
[1] "list" "list" "list"
> c(class(x[[1]]),mode(x[[1]]),typeof(x[[1]]))
[1] "numeric" "numeric" "double"
> c(class(x[[2]]),mode(x[[2]]),typeof(x[[2]]))
[1] "complex" "complex" "complex"
> c(class(x[[3]]),mode(x[[3]]),typeof(x[[3]]))
[1] "character" "character" "character"
> c(class(x[[4]]),mode(x[[4]]),typeof(x[[4]]))
[1] "logical" "logical" "logical"
```

Remark: The type, class, and mode of these objects is "list," but the individual elements of the list retain their original type. This requires us to be careful when employing the functions that tell us the data type of class. For example, below we see the object `x` is of type list. Although the list has an element that is complex, the `is.complex()` function returns FALSE unless we ask about the complex element directly.

```
> is.list(x)
[1] TRUE
> is.complex(x)
[1] FALSE
> is.complex(x[[2]])
[1] TRUE
```

If desired, we can convert the list to a vector using the `unlist()` function. For example, we apply the `unlist()` function to the list we created above. Note that because we've converted the list to a vector, the elements have been coerced to characters.

```
> (y<-unlist(x))
[1] "1"    "3+1i" "hi"   "TRUE"
> is.list(y)
[1] FALSE
> is.vector(y)
[1] TRUE
> c(class(y),mode(y),typeof(y))
[1] "character" "character" "character"
```

We will discuss lists more complicated lists in the following chapter as they are quite helpful in representing complicated data structures, as well as when coding the output functionality of new functions.

1.4.9 Summary of R Commands

Table 1.4.5 summarizes the operators used in this section.

Operator	Functionality	Example From Notes
c(...)	creates a vector of elements	c(1,4,3,2,5,2,6,7,8)
seq(from=x,to=y,by=z)	creates vector from x to y counting by z	seq(from=0,to=10,by=2)
x:y	equivalent to seq(from=x,to=y,by=1)	a:b
rep(x,y)	creates a vector of x repeated y times	rep(3,5)
order(x)	returns the positions of elements in x in ascending order unless decreasing is set to TRUE	order(i) order(i,decreasing=TRUE)
length(x)	returns the number of elements in x	length(g)
x%*%y	returns the dot product of x and y	g%*%m
head(x)	returns a preview of the beginning of x	head(g)
unique(x)	returns the unique elements in x	unique(g)
summary(x)	returns a summary of x	summary(g)
table(x)	returns a table of frequencies for each observation in x	table(h)
which(*logical*)	return the position of the elements in the given *logical* statement is true	which(g==2)
subset(x,*logical*)	return the elements of x where the given *logical* statement is true	subset(g,g==2)
min(x)	returns the minimum element in x	min(g)
max(x)	returns the maximum element in x	max(g)
sum(x)	returns the sum of all elements in x	sum(g)
cumsum(x)	returns the cumulative sum of x	cumsum(g)
mean(x)	returns the average value of elements in x	mean(g)
sd(x)	returns the standard deviation of elements in x	sd(g)
quantile(x,probs=y)	returns the y-th percentile of elements in x	quantile(g,probs=c(0.90))
intersect(x,y)	returns the common elements of x and y	intersect(g,i)
union(x,y)	returns the elements of x and y combined	union(g,i)
setdiff(x,y)	returns the values in x that are not in y	setdiff(g,i)
any(*logicals*)	returns true if all values are TRUE	any(g%in%i)
all(*logicals*)	returns true if any values are TRUE	all(g%in%i)
list(...)	creates a list of elements	x<-list(1,3+1i,"hi",TRUE)

Table 1.4.5: A list of basic operators for vectors in R.