Week 3 Module 2: Constructing Vectors

CSCI E-5a: Introduction to R

Let's clear the global computing environment:

```
rm( list = ls() )
```

Module Overview and Learning Outcomes

Hello! And welcome to Module 2: Constructing Vectors.

In this module, we'll study four ways to construct vectors.

- In section 1, we'll review the `c()` function, which enables us to construct arbitrary vectors.
- In section 2, we'll study the `rep()` function, which we can use to create vectors with repeated values.
- In section 3, we'll learn about the colon operator, which allows us to quickly create numerical sequences in steps of 1.
- In section 4, we'll study the `seq()` function, which generalizes the colon operator and allows us to create arbitrary arithmetic sequences.

At the end of this module, you'll be able to:

- Create a vector consisting of an arbitrary sequence of values by using the `c()` function.
- Create a vector consisting of repeated values by using the `rep()` function.
- Create a vector consisting of a numerical sequence in steps of 1 by using the colon operator.
- Create a vector consisting of an arbitrary arithmetic sequence by using the `seq()` function.

There are 3 new built-in R functions in this module:

- The `rep()` function
- The colon operator
- The `seq()` function

All right! Let's get started by reviewing the `c()` function.

Section 1: The c() function

Main Idea: *We can review the `c()` function*

In this section, we'll review the `c()` function.

The `c()` function *combines* values of the same class together into a single vector.

This operation works by explicitly listing each element of the vector:

```
prime.numeric.vector <- c( 2, 3, 5, 7, 11, 13 )

prime.numeric.vector
```

```
## [1]  2  3  5  7 11 13
```

We can also use the `c()` function to combine vectors together, as long as they are both the same class:

```
first.numeric.vector <- c(1, 2, 3)

second.numeric.vector <- c(8, 9, 10)

c(first.numeric.vector, second.numeric.vector)
```

```
## [1]  1  2  3  8  9 10
```

The `c()` function is the most general vector construction method possible, in the sense that you can make any vector you want as long as you're willing to individually specify each element.

Thus, you can generate completely arbitrary sequences of values with the `c()` function.

Of course, this is always subject to the condition that all the values in the vector must have the same atomic data type.

However, this generality comes with a price, because if the elements in the vector follow some sort of pattern, the `c()` function can't take advantage of that pattern.

It's important to realize that although all of the examples in this section have used numeric vectors, the `c()` function can also be used to create logical or character vectors.

So that's a review of the `c()` function.

Now let's learn about the `rep()` function.

Section 2: The rep() function

Main Idea: *We can construct vectors with repeating values by using the `rep()` function*

In this section, we'll study the `rep()` function, which we can use to create vectors with repeated values.

One of the simplest functions for constructing a vector with some sort of regular pattern is the `rep()` function, which repeats a given value for a specified number of times.

To create a vector which consists of the value `x = 6` repeated 12 times, we use the `times` option:

```
rep( x = 6, times = 12 )
```

```
##  [1] 6 6 6 6 6 6 6 6 6 6 6 6
```

The `rep()` isn't restricted to working with only atomic data values, and can repeat a vector as well:

```
rep( x = c(1, 2, 3, 4, 5), times = 3)
```

```
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

We can also specify the option `each`, which will cause the `rep()` function to repeat each element in the input data argument a certain number of times before moving on to the next element.

For instance, to create a vector with each number from 1 to 5 repeated 10 times, we have:

```
rep( x = c(1, 2, 3, 4, 5), each = 10)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4
## [39] 4 4 5 5 5 5 5 5 5 5 5 5
```

So that's how to construct vectors with repeated values by using the `rep()` function.

Now let's learn about the colon operator, which allows us to quickly create numerical sequences in steps of 1.

Exercise 1: Creating a vector

Create a vector with a sequence consisting of three parts:

- First, the value 5 is repeated 10 times.
- Second, the value -2 is repeated 6 times.
- Finally, the value 8 is repeated 3 times.

Report your result using a `cat()` statement, displaying the values with 2 decimal places.

Solution

Section 3: The colon operator

Main Idea: *We can construct numerical sequences in steps of 1 by using the colon operator*

In this section, we'll learn about the colon operator, which allows us to quickly create numerical sequences in steps of 1.

The colon operator takes two numbers and produces a vector of numbers starting at the first value and ending at the second number in increments or decrements of 1.

Here's a very simple example, where we start at the value 1 and end at the value 5:

```
1:5
```

```
## [1] 1 2 3 4 5
```

Notice how the rules operate: we start at the value 1 and end at the value 5, always increasing in steps of 1.

We could reverse this process:

```
5:1
```

```
## [1] 5 4 3 2 1
```

Now we start at the value 5 and end at the value 1, this time decreasing in steps of 1.

You can use negative numbers with the colon operator:

```
-2:3
```

```
## [1] -2 -1  0  1  2  3
```

In fact, you don't even have to use integer values with the colon operator:

```
2.5:8.5
```

```
## [1] 2.5 3.5 4.5 5.5 6.5 7.5 8.5
```

As always, the colon operator produces a numeric vector with increments of 1.

What happens if you try to use the colon operator with two numbers that aren't a whole number of units apart?

For instance, this code:

```
2.5:6.7
```

```
## [1] 2.5 3.5 4.5 5.5 6.5
```

As usual, R starts at the value 2.5, and increments by 1 up to 6.5.

The next step would be 7.5, but this is greater than the upper limit in the colon operator expression, so R does not go beyond this.

So the answer is: R will start at the value and continually increment by 1 until it exceeds the second value in the colon operator expression, and at that point R will stop.

Exercise 2: The colon operator

Create a vector of the form:

$$1000,\ 999,\ 998,\ \ldots, 922,\ 921,\ 920$$

Report the first 10 elements of this vector using a `cat()` statement, displaying the values with 2 decimal places.

Solution

When we construct a vector with the colon operator, then we can use it as an input to more vector construction methods.

For instance, we can take the vector we developed in Exercise 2 and then combine it with the number 10:

```
c( 1000:920, 10 )
```

```
##  [1] 1000  999  998  997  996  995  994  993  992  991  990  989  988  987  986
## [16]  985  984  983  982  981  980  979  978  977  976  975  974  973  972  971
## [31]  970  969  968  967  966  965  964  963  962  961  960  959  958  957  956
## [46]  955  954  953  952  951  950  949  948  947  946  945  944  943  942  941
## [61]  940  939  938  937  936  935  934  933  932  931  930  929  928  927  926
## [76]  925  924  923  922  921  920   10
```

So that's how to create a numerical sequence in steps of 1 by using the colon operator.

Now let's see how to construct arbitrary arithmetic sequences by using the `seq()` function.

Exercise 3: Creating a vector

Without using the c() function create this vector:

$$-3.7, -2.7, -1.7, -0.7,\ 0.3,\ 1.3,\ 2.3,\ 3.3,\ 4.3$$

Solution

Exercise 4: Creating a vector

Use the colon operator twice to create two vectors that are then combined to create the final vector:

$$\{\ 10,\ 9,\ 8,\ 7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1,\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10\ \}$$

Solution

Section 4: The `seq()` function

Main Idea: *We can create arbitrary arithmetic sequences by using the `seq()` function*

In this section, we'll study the `seq()` function, which generalizes the colon operator and allows us to create arbitrary arithmetic sequences.

An important point about the colon operator is that it always increments or decrements in steps of 1.

We can generalize this process by allowing the increment or decrement to be some value other than 1, as long as it's a fixed value.

An *arbitrary arithmetic sequence* is a sequence of numbers where the increment or decrement is always the same amount.

Thus, the arbitrary arithmetic sequence starting at the value -1 with an increment of 3 with 5 terms is:

$$-1,\ 2,\ 5,\ 8,\ 11$$

The `seq()` function allows us to specify numerical sequences in a variety of different ways.

The function has four optional input arguments:

- The `from` option determines the first value of the sequence.
- The `to` option determines the last value of the sequence.
- The `by` option determines the increment or decrement for each element of the sequence.
- The `length.out` option determines the number of terms in the sequence.

If you think about it, once we know any three of these values, then the fourth is fixed.

Thus, the you can use the `seq()` function in four different ways:

- You can generate a sequence by specifying the first and last values, as well as the increment. Then `seq()` will determine how long the sequence needs to be.
- You can generate a sequence by specifying the first and last values, as well as the length. Then `seq()` will determine the increment value.
- You can generate a sequence by specifying the first value, the increment, and the length. Then `seq()` will determine the ending point.
- You can generate a sequence by specifying the last value, the increment, and the length. Then `seq()` will determine the starting point.

Let's try some of these methods out.

Often, we'll know the starting and ending points of the sequence, and then we have to specify either the increment or the length of the sequence.

For example, how can we construct a sequence that starts at -1.2, ends at 0.7, and has an increment of 0.01?

- We can set the `from` parameter equal to -1.2.
- The `to` parameter equal to 0.7.
- The `by` parameter equal to 0.01.

Putting this together, we have:

```
seq( from = -1.2, to = 0.7, by = 0.01)
```

```
##   [1] -1.20 -1.19 -1.18 -1.17 -1.16 -1.15 -1.14 -1.13 -1.12 -1.11 -1.10 -1.09
##  [13] -1.08 -1.07 -1.06 -1.05 -1.04 -1.03 -1.02 -1.01 -1.00 -0.99 -0.98 -0.97
##  [25] -0.96 -0.95 -0.94 -0.93 -0.92 -0.91 -0.90 -0.89 -0.88 -0.87 -0.86 -0.85
##  [37] -0.84 -0.83 -0.82 -0.81 -0.80 -0.79 -0.78 -0.77 -0.76 -0.75 -0.74 -0.73
##  [49] -0.72 -0.71 -0.70 -0.69 -0.68 -0.67 -0.66 -0.65 -0.64 -0.63 -0.62 -0.61
##  [61] -0.60 -0.59 -0.58 -0.57 -0.56 -0.55 -0.54 -0.53 -0.52 -0.51 -0.50 -0.49
##  [73] -0.48 -0.47 -0.46 -0.45 -0.44 -0.43 -0.42 -0.41 -0.40 -0.39 -0.38 -0.37
##  [85] -0.36 -0.35 -0.34 -0.33 -0.32 -0.31 -0.30 -0.29 -0.28 -0.27 -0.26 -0.25
##  [97] -0.24 -0.23 -0.22 -0.21 -0.20 -0.19 -0.18 -0.17 -0.16 -0.15 -0.14 -0.13
## [109] -0.12 -0.11 -0.10 -0.09 -0.08 -0.07 -0.06 -0.05 -0.04 -0.03 -0.02 -0.01
## [121]  0.00  0.01  0.02  0.03  0.04  0.05  0.06  0.07  0.08  0.09  0.10  0.11
## [133]  0.12  0.13  0.14  0.15  0.16  0.17  0.18  0.19  0.20  0.21  0.22  0.23
## [145]  0.24  0.25  0.26  0.27  0.28  0.29  0.30  0.31  0.32  0.33  0.34  0.35
## [157]  0.36  0.37  0.38  0.39  0.40  0.41  0.42  0.43  0.44  0.45  0.46  0.47
## [169]  0.48  0.49  0.50  0.51  0.52  0.53  0.54  0.55  0.56  0.57  0.58  0.59
## [181]  0.60  0.61  0.62  0.63  0.64  0.65  0.66  0.67  0.68  0.69  0.70
```

Exercise 5: Constructing a sequence

Construct a sequence that starts at 2.3 and goes to 4.5 in increments of 0.2.

Solution

Instead of specifying the increment, we can specify the length of the sequence by using the `length` option.

Let's construct a sequence starting at 3 and ending at 6.8 of length 10:

```
seq( from = 3, to = 6.8, length.out = 10 )
```

```
##  [1] 3.000000 3.422222 3.844444 4.266667 4.688889 5.111111 5.533333 5.955556
##  [9] 6.377778 6.800000
```

Exercise 6: Constructing a sequence

Construct a sequence starting at 12.8 and ending at 49.6 of length 200.

Solution

So far, we've always specified our sequences by giving a starting and an ending value.

But sometimes this isn't convenient, and we'll want to specify just the starting value alone.

To do this, we then have to specify the increment value and the length of the sequence.

For example, suppose we wish to construct a sequence of the first 12 even numbers.

To do this, we can express the concept of the first 12 even numbers by specifying three of the named parameters:

- To specify one parameter, the first even number is 2, so we should set the parameter `first` equal to 2.
- To specify a second parameter, note that even numbers go up by an increment of 2: 2, 4, 6, 8, . . . , so the increment option `by` should be equal to 2.
- Finally, the sequence is specified to be of length 12, so we should set the `length.out` parameter to 12.

Let's see this:

```
seq( from = 2, by = 2, length.out = 12 )
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24
```

There are many different ways to apply the `seq()` function, and it will take a little while to get comfortable with them.

Exercise 7: Odd numbers

Write R code to generate the sequence of the first 15 positive odd numbers.

Solution

So that's how to generate arbitrary arithmetic sequences by using the `seq()` function.

Now let's review what we've learned in this module.

Module Review

In this module, we studied four ways to construct vectors.

- In section 1, we reviewed the `c()` function, which enables us to construct arbitrary vectors.
- In section 2, we studied the `rep()` function, which we can use to create vectors with repeated values.
- In section 3, we learned about the colon operator, which allows us to quickly create numerical sequences in steps of 1.
- In section 4, we studied the `seq()` function, which generalizes the colon operator and allows us to create arbitrary arithmetic sequences.

Now that you've completed this module, you should be able to:

- Create a vector consisting of an arbitrary sequence of values by using the `c()` function.
- Create a vector consisting of repeated values by using the `rep()` function.
- Create a vector consisting of a numerical sequence in steps of 1 by using the colon operator.
- Create a vector consisting of an arbitrary arithmetic sequence by using the `seq()` function.

There were 3 new built-in R functions in this module:

- The `rep()` function
- The colon operator
- The `seq()` function

All right! That's it for Module 2: Constructing Vectors.

Now let's move on to Module 3: Indexing Vectors.

Solutions to the Exercises

Exercise 1: Creating a vector

Create a vector with a sequence consisting of three parts:

- First, the value 5 is repeated 10 times.
- Second, the value -2 is repeated 6 times.
- Finally, the value 8 is repeated 3 times.

Report your result using a `cat()` statement, displaying the values with 2 decimal places.

Answer

Here's my solution:

```
exercise.2.1.vector <-
    c(
        rep( x = 5, times = 10 ),
        rep( x = -2, times = 6),
        rep( x = 8, times = 3 )
    )

cat(
    "Exericse 2.1 vector:",
    formatC(
        exercise.2.1.vector,
        format = "f",
        digits = 2
    ),
    fill = TRUE
)
```

```
## Exericse 2.1 vector: 5.00 5.00 5.00 5.00 5.00 5.00 5.00 5.00 5.00 5.00 -2.00
## -2.00 -2.00 -2.00 -2.00 -2.00 8.00 8.00 8.00
```

Exercise 2: The colon operator

Create a vector of the form:

$$1000,\ 999,\ 998,\ \ldots, 922,\ 921,\ 920$$

Report the first 10 elements of this vector using a `cat()` statement, displaying the values with 2 decimal places.

Solution

```
long.vector <-
    1000:920

cat(
    "long.vector:",
    formatC(
        head( long.vector, n = 10 ),
        format = "f",
        digits = 2
    )
)
```

```
## long.vector: 1000.00 999.00 998.00 997.00 996.00 995.00 994.00 993.00 992.00 991.00
```

Exercise 3: Creating a vector

Create a vector of the form:

$$-3.7, -2.7, -1.7, -0.7,\ 0.3,\ 1.3,\ 2.3,\ 3.3,\ 4.3$$

Solution

Here's one approach:

```
c( -3.7:-0.7, 0.3:4.3 )
```

```
## [1] -3.7 -2.7 -1.7 -0.7  0.3  1.3  2.3  3.3  4.3
```

Here's another:

```
-3.7:4.3
```

```
## [1] -3.7 -2.7 -1.7 -0.7  0.3  1.3  2.3  3.3  4.3
```

Exercise 4: Creating a vector

Use the colon operator twice to create two vectors that are then combined to create the final vector:

$$\{ 10,\ 9,\ 8,\ 7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1,\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10 \}$$

Solution

Here's one simple approach:

```
c( 10:0, 1:10 )
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1  0  1  2  3  4  5  6  7  8  9 10
```

Here's an alternative way to do this:

```
c(10:1, 0:10 )
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1  0  1  2  3  4  5  6  7  8  9 10
```

Really, both methods are fine.

Exercise 5: Constructing a sequence

Construct a sequence that starts at 2.3 and goes to 4.5 in increments of 0.2.

Solution

```
seq( from = 2.3, to = 4.5, by = 0.2 )
```

```
##  [1] 2.3 2.5 2.7 2.9 3.1 3.3 3.5 3.7 3.9 4.1 4.3 4.5
```

Exercise 6: Constructing a sequence

Construct a sequence starting at 12.8 and ending at 49.6 of length 200.

Solution

```
seq( from = 12.8, to = 49.6, length.out = 200 )
```

```
##   [1] 12.80000 12.98492 13.16985 13.35477 13.53970 13.72462 13.90955 14.09447
##   [9] 14.27940 14.46432 14.64925 14.83417 15.01910 15.20402 15.38894 15.57387
##  [17] 15.75879 15.94372 16.12864 16.31357 16.49849 16.68342 16.86834 17.05327
##  [25] 17.23819 17.42312 17.60804 17.79296 17.97789 18.16281 18.34774 18.53266
##  [33] 18.71759 18.90251 19.08744 19.27236 19.45729 19.64221 19.82714 20.01206
##  [41] 20.19698 20.38191 20.56683 20.75176 20.93668 21.12161 21.30653 21.49146
##  [49] 21.67638 21.86131 22.04623 22.23116 22.41608 22.60101 22.78593 22.97085
##  [57] 23.15578 23.34070 23.52563 23.71055 23.89548 24.08040 24.26533 24.45025
##  [65] 24.63518 24.82010 25.00503 25.18995 25.37487 25.55980 25.74472 25.92965
##  [73] 26.11457 26.29950 26.48442 26.66935 26.85427 27.03920 27.22412 27.40905
##  [81] 27.59397 27.77889 27.96382 28.14874 28.33367 28.51859 28.70352 28.88844
##  [89] 29.07337 29.25829 29.44322 29.62814 29.81307 29.99799 30.18291 30.36784
##  [97] 30.55276 30.73769 30.92261 31.10754 31.29246 31.47739 31.66231 31.84724
## [105] 32.03216 32.21709 32.40201 32.58693 32.77186 32.95678 33.14171 33.32663
## [113] 33.51156 33.69648 33.88141 34.06633 34.25126 34.43618 34.62111 34.80603
## [121] 34.99095 35.17588 35.36080 35.54573 35.73065 35.91558 36.10050 36.28543
## [129] 36.47035 36.65528 36.84020 37.02513 37.21005 37.39497 37.57990 37.76482
## [137] 37.94975 38.13467 38.31960 38.50452 38.68945 38.87437 39.05930 39.24422
## [145] 39.42915 39.61407 39.79899 39.98392 40.16884 40.35377 40.53869 40.72362
## [153] 40.90854 41.09347 41.27839 41.46332 41.64824 41.83317 42.01809 42.20302
## [161] 42.38794 42.57286 42.75779 42.94271 43.12764 43.31256 43.49749 43.68241
## [169] 43.86734 44.05226 44.23719 44.42211 44.60704 44.79196 44.97688 45.16181
## [177] 45.34673 45.53166 45.71658 45.90151 46.08643 46.27136 46.45628 46.64121
## [185] 46.82613 47.01106 47.19598 47.38090 47.56583 47.75075 47.93568 48.12060
## [193] 48.30553 48.49045 48.67538 48.86030 49.04523 49.23015 49.41508 49.60000
```

Exercise 7: Odd numbers

Write R code to generate the sequence of the first 15 odd numbers.

Solution

```
seq( from = 1, by = 2, length.out = 15 )
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29
```