Vectors

Today we're going to discuss one of the most commonly used tools in RStudio, vectors. Vectors are simply just sequences of data elements that are also of the same data type whether it be integer, double, logical, character, or complex. These tools are useful because of how useful they can be. They can be a variety of elements, can be of any length, and make it easier to understand the data at hand. In this post, I will discuss the rules that vectors abide by, as well as their applications.

Atomicity First, we will expand on one of the vector principles I mentioned earlier, atomicity. Like I said above, vectors are atomic structures which mean that they can only contain data elements of the same data type. EX: A valid vector a <- c(1, 3, 2) b <- c(TRUE, FALSE, TRUE) Not a vector a <- c(1, "this")

Coercion In the case that you try to make a vector that contains different elements, R will change this so that they are all of the same type, a method that is also known as implicit coercion. What R changes the types to is shown by: logical -> integer -> numeric -> complex -> character "http://nicercode.github.io/2014-02-18-UTS/lessons/01-intro_r/data-structures.html" EX: a <- c("a", 1, TRUE) b <- c("a", "1", "TRUE") If you want to go against these implicit coercion rules, then you can use some coercion functions that will change the vector to your desired data type. as.character() as.numeric() as.integer() as.logical() EX: a <- c(TRUE, 1) b <- as.character(a)

Missing Data Because of the way that R is structured, it supports having missing data in vectors. When you want to have missing data, simply type NA. EX: x <- c(0.5, NA, 0.7) x <- c(TRUE, FALSE, NA) https://swcarpentry.github.io/r-novice-inflammation/13-supp-data-structures/ The other functions you can use when there is missing data present is: is.na() anyNA() is.na is used when you want to indicate which elements of a vector are missing data. EX:

```
x <- c("a", NA, "c", "d", NA)
is.na(x)
```

```
## [1] FALSE  TRUE FALSE FALSE  TRUE
```

anyNA() is used when you want to determine if there are any missing values in your vector. EX:

```
x <- c("a", NA, "c", "d", NA)
anyNA(x)
```

```
## [1] TRUE
```

Subsetting Bracket Notation System In order to extra certain values from R, you use brackets [] and you specify which objects you want inside of the vectors. If you have many indices, then you can separate it by factors. EX:

```
x <- c(1, 3, 2)
x[3]
```

```
## [1] 2
```

```
x[x==2]
```

```
## [1] 2
```

```
x[x>1]
```

```
## [1] 3 2
```

Now that we understand the basics of subsetting, I'm going to show images of more complex subsetting which I got from https://stats.idre.ucla.edu/r/modules/subsetting-data/.

```
hsb2.small <- read.csv("https://stats.idre.ucla.edu/stat/data/hsb2_small.csv")
(hsb3 <- hsb2.small[, c(1, 7, 8)])
```

```
##     id read write
## 1   70   57    52
## 2  121   68    59
## 3   86   44    33
## 4  141   63    44
## 5  172   47    52
## 6  113   44    52
## 7   50   50    59
## 8   11   34    46
## 9   84   63    57
## 10  48   57    55
## 11  75   60    46
## 12  60   57    65
## 13  95   73    60
## 14 104   54    63
## 15  38   45    57
## 16 115   42    49
## 17  76   47    52
## 18 195   57    57
## 19 114   68    65
## 20  85   55    39
## 21 167   63    49
## 22 143   63    63
## 23  41   50    40
## 24  20   60    52
## 25  12   37    44
```

```
(hsb4 <- hsb2.small[, 1:4])
```

```
##     id female race ses
## 1   70      0    4   1
## 2  121      1    4   2
## 3   86      0    4   3
## 4  141      0    4   3
## 5  172      0    4   2
## 6  113      0    4   2
## 7   50      0    3   2
## 8   11      0    1   2
## 9   84      0    4   2
## 10  48      0    3   2
## 11  75      0    4   2
## 12  60      0    4   2
## 13  95      0    4   3
## 14 104      0    4   3
## 15  38      0    3   1
## 16 115      0    4   1
## 17  76      0    4   3
## 18 195      0    4   2
## 19 114      0    4   3
## 20  85      0    4   2
## 21 167      0    4   2
## 22 143      0    4   2
## 23  41      0    3   2
## 24  20      0    1   3
## 25  12      0    1   2
```

```
(hsb6 <- hsb2.small[hsb2.small$ses == 1, ])
```

```
##     id female race ses schtyp prog read write math science socst
## 1   70      0    4   1      1    1   57    52   41      47    57
## 15  38      0    3   1      1    2   45    57   50      31    56
## 16 115      0    4   1      1    1   42    49   43      50    56
```

Adding Elements You can also add to current existing vectors by utilizing commas once again. https://swcarpentry.github.io/r-novice-inflammation/13-supp-data-structures/ EX:

```
z <- c("Sarah", "Tracy", "Jon")
z <- c(z, "Annette")
```

Instead of adding just one element, you can also combine two vectors using a method similar to the one I used above. http://www.r-tutor.com/r-introduction/vector/combining-vectors EX:

```
n = c(2, 3, 5)
s = c("aa", "bb", "cc", "dd", "ee")
c(n, s)
```

```
## [1] "2"  "3"  "5"  "aa" "bb" "cc" "dd" "ee"
```

Recycling Another attribute of vectors is recycling. Recycling occurs when a shorter vector is “recycled” in order to be the same length of a longer vector. https://www.safaribooksonline.com/library/view/the-art-of/9781593273842/ch02s03.html

```
c(6,0,9,20,22) + c(1,2,4)
```

```
## Warning in c(6, 0, 9, 20, 22) + c(1, 2, 4): longer object length is not a
## multiple of shorter object length
```

```
## [1]  7  2 13 21 24
```

As you can see, the two vectors are different lengths so the shorter one is repeated and added to the longer one until it matches the length of the longer vector. Another way that recycling can be utilized is by adding a singular number, not another vector, to a vector and that number is repeatedly used until it is the same length as the vector. EX:

```
a <- c(1, 2, 3)
a + 10
```

```
## [1] 11 12 13
```

This method of recycling can be useful if you are dealing with bigger data sets such as lists because or data that you obtained online because you can just simply add a number or add two vectors to see its effect without having to manually add each number individually. According to https://www.r-bloggers.com/how-to-use-vectorization-to-streamline-simulations/, vectorization streamlines simulations in that you can utilize multiple arithmetic operators and summations to a vector all at once.

Matrix Matrices are simply vectors that are two-dimensional arrays. This is why when you make them, you have to specify the number of rows and columns that they have. First, I'll show a basic example so you can understand the basics. EX:

```
m <- matrix(nrow = 2, ncol = 2)
m
```

```
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
```

To verify the number of rows and columns that we put in m, we use dim(). EX:

```
m <- matrix(nrow = 2, ncol = 2)
dim(m)
```

```
## [1] 2 2
```

Matrices are also filled column-wise which will be seen in the example below. EX:

```
m <- matrix(1:6, nrow=2, ncol =3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Now that you've seen the basic structure of matrices, I'll show other ways of creatng a matrix. http://nicercode.github.io/2014-02-18-UTS/lessons/01-intro_r/data-structures.html EX:

```
m <- 1:10
dim(m) <- c(2,5)
```

Now that you understand how to build matrices and how they work, I'll show a more complex matrix to show how this can be useful in real life. https://www.safaribooksonline.com/library/view/learning-r/9781449357160/ch04.html EX:

```
(a_matrix <- matrix(
  1:12,
  nrow = 4,            #ncol = 3 works the same
  dimnames = list(
    c("one", "two", "three", "four"),
    c("ein", "zwei", "drei")
  )
))
```

```
##       ein zwei drei
## one     1    5    9
## two     2    6   10
## three   3    7   11
## four    4    8   12
```

As seen by the example above, you can even list the columns and rows in your matrix by simply utilizing lists. This can be useful in real life because if you are given information on a certain topic you are planning on researching, you can create a matrix with your given data so that you research will be easier to read and understand. Of course the example above is a simplified version of what matrices would look like if you used real data for a real experiment, but this non-specific version helps you understand the gist of those potential experiments.

Conclusion In this post I went over the basics of vectors and the different applications you can use for them. Even though vectors are simply

used to contain data, it also makes it easier for you to manipulate data because of tools such as recycling, atomization, and matrices. Recycling means that instead of having to manually add ten or find the square root of each number in a data set, you can simply utilize recycling and atomization to save yourself time, energy, and potential mistakes. As you can see by subsetting and bracket notation, you can adjust the vectors so that it only shows the information that you are focused on. For example, in class we constantly go over points scored by NBA players and teams. By using subsetting, we can see the specifics of a certain player and type so that we can more clearly see the information that we are focusing on. By creating matrices, you can easily organize data that you may have obtained from an experiment. Matrices make this information easier to read and understand and the ability to name the rows and columns only make them easier to understand.

References http://nicercode.github.io/2014-02-18-UTS/lessons/01-intro_r/data-structures.html https://swcarpentry.github.io/r-novice-inflammation/13-supp-data-structures/ https://stats.idre.ucla.edu/r/modules/subsetting-data/ http://www.r-tutor.com/r-introduction/vector/combining-vectors https://www.safaribooksonline.com/library/view/the-art-of/9781593273842/ch02s03.html https://www.r-bloggers.com/how-to-use-vectorization-to-streamline-simulations/ https://www.safaribooksonline.com/library/view/learning-r/9781449357160/ch04.html