Introducing vectors

Mark Andrews

July 7, 2018

Introduction

Vectors are simply arrays or lists of numbers. Most of time, when we are working with data, we work with *data frames*. Data frames can be seen as similar to spreadsheets, i.e. with multiple rows and multiple columns, and each column representing a variable. Here, each column is a vector and often we need to work directly with it.

We'll start by creating a vector of random numbers by using the `runif()` function.

```
set.seed(10101) # re-set the random number generator (for reproducibility)
(x <- runif(100)) # get a set of 100 random numbers, each between 0 and 1
```

```
##   [1] 0.190306607 0.910839343 0.227716102 0.824990480 0.915575991
##   [6] 0.505208322 0.593897328 0.266588832 0.543251008 0.968824523
##  [11] 0.885457902 0.728701118 0.323091091 0.051260170 0.739026165
##  [16] 0.672824457 0.661583063 0.464123954 0.781285814 0.746652909
##  [21] 0.976360638 0.058002294 0.520461000 0.024606424 0.319649113
##  [26] 0.599577925 0.152016634 0.081722761 0.099273221 0.630299117
##  [31] 0.352381369 0.546844742 0.966773009 0.343588795 0.484798875
##  [36] 0.872714166 0.582099408 0.878948864 0.939847989 0.007139965
##  [41] 0.885120127 0.450129902 0.781579257 0.279908721 0.226114098
##  [46] 0.986621519 0.594569547 0.503888618 0.216989901 0.877448510
##  [51] 0.174646803 0.912904222 0.024360589 0.690325505 0.971542208
##  [56] 0.680246338 0.553203338 0.442732332 0.902920122 0.721032068
##  [61] 0.451187217 0.592516636 0.724036959 0.789851444 0.019221483
##  [66] 0.181603466 0.696527751 0.554127032 0.040518970 0.434538747
##  [71] 0.832726117 0.733160632 0.715696271 0.464023750 0.418075769
##  [76] 0.391863183 0.186251086 0.832025870 0.749561106 0.590161412
##  [81] 0.521371588 0.567971813 0.068352656 0.437979033 0.515919067
##  [86] 0.130152999 0.424100534 0.237360322 0.383141873 0.202172476
##  [91] 0.009509650 0.376098109 0.507953159 0.098784285 0.864315904
##  [96] 0.816745557 0.332331035 0.899268875 0.828766771 0.598339274
```

Let's examine the vector

```
class(x) # What kind of object is it?
```

```
## [1] "numeric"
```

```
length(x)
```

```
## [1] 100
```

```
str(x)   # compactly display internal structure of x
```

```
##  num [1:100] 0.19 0.911 0.228 0.825 0.916 ...
```

```
summary(x) # summarize the info in x
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00714 0.30971 0.54505 0.52533 0.75749 0.98662
```

Indexing

- What is the value of element 68?

```
x[68]
```

```
## [1] 0.554127
```

Slicing

Slicing will give us a contiguous subset of the vector. For example, what are the values of elements 12 to 18 inclusive?

```
x[12:18]
```

```
## [1] 0.72870112 0.32309109 0.05126017 0.73902617 0.67282446 0.66158306
## [7] 0.46412395
```

Multiple indices and subsets

What are the values of elements 17, 89, 39, 42? To do this, first create a vector of those elements and call it *indx* and then use this to slice x

```
indx <- c(17, 89, 39, 42)
x[indx]
```

```
## [1] 0.6615831 0.3831419 0.9398480 0.4501299
```

This is the same thing as doing

```
x[c(17, 89, 39, 42)]
```

```
## [1] 0.6615831 0.3831419 0.9398480 0.4501299
```

What if we needed to find all elements that are equal to or less than the value of 0.2? Here, we can create an indexing vector called *indx* and then use this to extract the elements:

```
(indx <- which(x <= 0.2)) # Get indices of elements whose values are = or < 12.
```

```
##  [1]  1 14 22 24 27 28 29 40 51 53 65 66 69 77 83 86 91 94
```

```
x[indx]
```

```
##  [1] 0.190306607 0.051260170 0.058002294 0.024606424 0.152016634
##  [6] 0.081722761 0.099273221 0.007139965 0.174646803 0.024360589
## [11] 0.019221483 0.181603466 0.040518970 0.186251086 0.068352656
## [16] 0.130152999 0.009509650 0.098784285
```

We also use Boolean indices here:

```
(indx <- x <= 0.2)
```

```
##   [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [12] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
##  [23] FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
##  [34] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
##  [45] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE
##  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [67] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
##  [78] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
##  [89] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE
```

```
x[indx]
```

```
##  [1] 0.190306607 0.051260170 0.058002294 0.024606424 0.152016634
##  [6] 0.081722761 0.099273221 0.007139965 0.174646803 0.024360589
## [11] 0.019221483 0.181603466 0.040518970 0.186251086 0.068352656
## [16] 0.130152999 0.009509650 0.098784285
```

Note that to find how many elements are less than or equal to e.g. 0.2, we can do

```
indx <- x == 0.2
sum(indx) # Sums up the Boolean vector
```

```
## [1] 0
```

which gives the same result as

```
indx <- which(x == 0.2)
length(indx)
```

```
## [1] 0
```

Warning: Remember that <= is an inequality test, and <- is an assignment operator, and == is an equality test and = is an assignment operator.

Descriptive statistics

We can easily get things like *mean*, *median*, *sd*, etc, etc.

```
mean(x)
```

```
## [1] 0.5253291
```

```
median(x)
```

```
## [1] 0.5450479
```

```
sd(x)
```

```
## [1] 0.2882439
```

```
var(x)
```

```
## [1] 0.08308454
```

```
min(x)
```

```
## [1] 0.007139965
```

```
max(x)
```

```
## [1] 0.9866215
```

```
range(x)
```

```
## [1] 0.007139965 0.986621519
```

```
IQR(x) # inter quartile range
```

```
## [1] 0.4477783
```

This will give us some standard percentiles,

```
quantile(x)
```

```
##          0%         25%         50%         75%        100%
## 0.007139965 0.309714015 0.545047875 0.757492283 0.986621519
```

and we can ask for specific percentiles too:

```
quantile(x, probs = c(0.025, 0.25, 0.5, 0.75, 0.975))
```

```
##       2.5%        25%        50%        75%      97.5%
## 0.02166256 0.30971401 0.54504787 0.75749228 0.97025131
```

Concatenating vectors

We can join up vectors using the generic "combine" function *c()*:

```
z <- c(0.1, 0.2, 0.3, 0.27, 0.42)
(y <- c(x, z))
```

```
##   [1] 0.190306607 0.910839343 0.227716102 0.824990480 0.915575991
##   [6] 0.505208322 0.593897328 0.266588832 0.543251008 0.968824523
##  [11] 0.885457902 0.728701118 0.323091091 0.051260170 0.739026165
##  [16] 0.672824457 0.661583063 0.464123954 0.781285814 0.746652909
##  [21] 0.976360638 0.058002294 0.520461000 0.024606424 0.319649113
##  [26] 0.599577925 0.152016634 0.081722761 0.099273221 0.630299117
##  [31] 0.352381369 0.546844742 0.966773009 0.343588795 0.484798875
##  [36] 0.872714166 0.582099408 0.878948864 0.939847989 0.007139965
##  [41] 0.885120127 0.450129902 0.781579257 0.279908721 0.226114098
##  [46] 0.986621519 0.594569547 0.503888618 0.216989901 0.877448510
##  [51] 0.174646803 0.912904222 0.024360589 0.690325505 0.971542208
##  [56] 0.680246338 0.553203338 0.442732332 0.902920122 0.721032068
##  [61] 0.451187217 0.592516636 0.724036959 0.789851444 0.019221483
##  [66] 0.181603466 0.696527751 0.554127032 0.040518970 0.434538747
##  [71] 0.832726117 0.733160632 0.715696271 0.464023750 0.418075769
##  [76] 0.391863183 0.186251086 0.832025870 0.749561106 0.590161412
##  [81] 0.521371588 0.567971813 0.068352656 0.437979033 0.515919067
##  [86] 0.130152999 0.424100534 0.237360322 0.383141873 0.202172476
##  [91] 0.009509650 0.376098109 0.507953159 0.098784285 0.864315904
##  [96] 0.816745557 0.332331035 0.899268875 0.828766771 0.598339274
## [101] 0.100000000 0.200000000 0.300000000 0.270000000 0.420000000
```