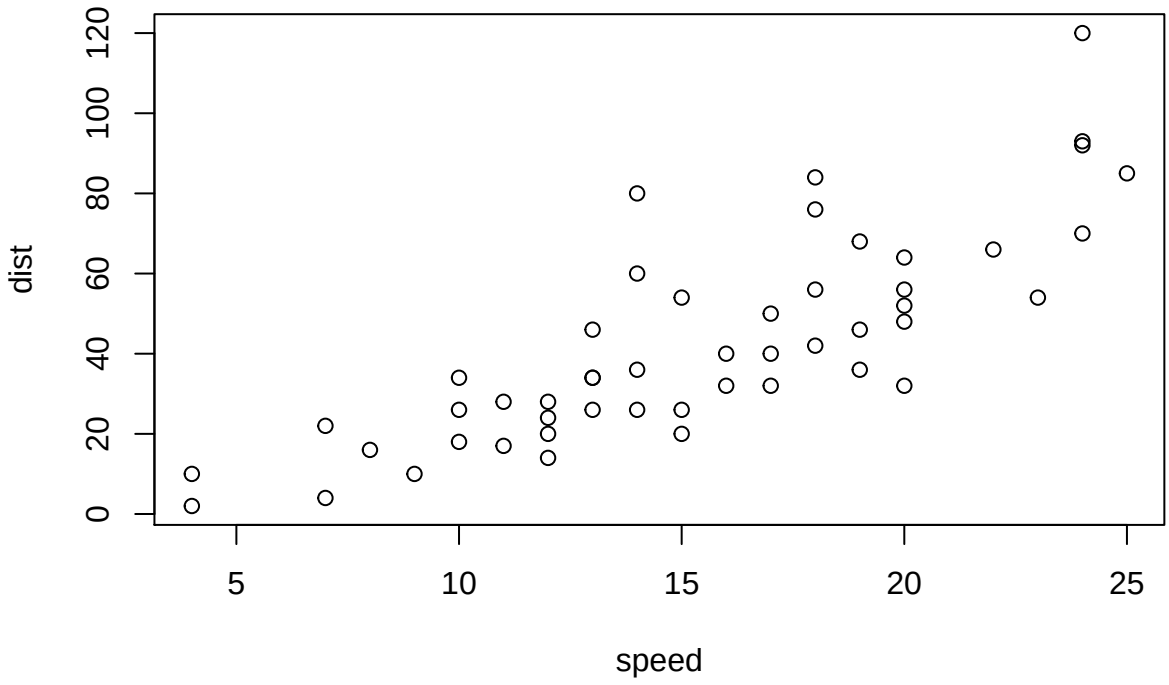
Introduction to R: vectors

Reference: http://hamelg.blogspot.com/ This is an R Markdown Notebook. When you execute code within the notebook, the results appear beneath the code.

Try executing this chunk by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Ctrl+Shift+Enter*.

```
plot(cars)
```



Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by pressing *Ctrl+Alt+I*.

When you save the notebook, an HTML file containing the code and output will be saved alongside it (click the *Preview* button or press *Ctrl+Shift+K* to preview the HTML file).

The preview shows you a rendered HTML copy of the contents of the editor. Consequently, unlike *Knit*, *Preview* does not run any R code chunks. Instead, the output of the chunk when it was last run in the editor is displayed.

Vectors

To create and store a vector with specific values, use the c() function and assign the result to a variable. c() takes a comma separated sequence of elements as input and combines them into a vector:

```r
x <- c(1,2,3)  # Create a numeric vector and assign it to x

print(x)  # Print the value of x to the screen
```

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

```r
y <- c("Life","Is","Study")  # Create a character vector

print(y)  # Print y to the screen
```

```
## [1] "Life"  "Is"    "Study"
```

You can also combine two vectors using c():

```r
z <- c(x,y) # Combine vectors x and y

print(z)
```

```
## [1] "1"     "2"     "3"     "Life"  "Is"    "Study"
```

If you try to combine vectors of different types as shown above, R will automatically convert the vector into the type that fits best. In this case, the numbers were converted into their character equivalents. When you create a vector, each element in the vector is assigned an index based on its position in the vector. The first element is at index position 1, the second element is at index position 2 and so on.

*Note: unlike many other programming languages, indexes in R start at 1 instead of 0.

When you print a vector to the screen, each line starts with a number in square brackets followed by vector values. The number in square brackets indicates the index of the next value listed on that line. For large vectors, this labeling can be helpful. For instance, consider a vector consisting of 100 random numbers between 0 and 1:

```r
random_data <- runif(100)  # Create a vector of 100 random numbers

print(random_data) # Print the vector
```

```
##   [1] 0.40308759 0.78786312 0.10851956 0.35837620 0.41837578 0.68770657
##   [7] 0.60281622 0.41025970 0.18765686 0.09141181 0.41456118 0.85571898
##  [13] 0.46387840 0.63827162 0.68505540 0.47449198 0.47974448 0.70004725
##  [19] 0.13113454 0.72735693 0.80442558 0.95145593 0.49199332 0.90103435
##  [25] 0.12963272 0.12688616 0.76829774 0.58404126 0.62211750 0.44546602
##  [31] 0.66616507 0.21452721 0.19890718 0.84046858 0.84321747 0.34958848
##  [37] 0.93827114 0.29315322 0.50927980 0.91771924 0.86215398 0.20946076
##  [43] 0.37182660 0.42381607 0.44236353 0.88523634 0.42812481 0.91080540
##  [49] 0.93223746 0.30600968 0.70981155 0.32185438 0.63833242 0.34469484
##  [55] 0.15306420 0.30176534 0.82971796 0.03729013 0.43398298 0.81909170
##  [61] 0.47998122 0.71702124 0.89827437 0.11237509 0.14486052 0.81032091
##  [67] 0.12106263 0.75464889 0.59938850 0.76484596 0.51016341 0.47194717
##  [73] 0.98191859 0.96366459 0.50098802 0.04849938 0.37295066 0.35319264
##  [79] 0.15305105 0.75128154 0.54135304 0.54324714 0.56812111 0.29694907
##  [85] 0.04979585 0.69641902 0.80539344 0.63115942 0.62928263 0.65184088
##  [91] 0.26386869 0.36225133 0.82841175 0.29975750 0.50678094 0.20040792
##  [97] 0.65267248 0.07521402 0.93568692 0.29569662
```

You can access a specific value in a vector by typing the name of the vector and then wrapping the index associated with the value you want to access in square brackets:

```r
random_data[7]  # Get the value at index 7
```

```
## [1] 0.6028162
```

Attempting to access an index that doesn't exist returns NA. NA denotes a missing value.

```
random_data[200]
```

```
## [1] NA
```

You can access ranges of values by placing a colon between the starting and ending indices of the range:

```
subset1 <- random_data[7:14]  # Get values from index 7 to 14

print(subset1)
```

```
## [1] 0.60281622 0.41025970 0.18765686 0.09141181 0.41456118 0.85571898
## [7] 0.46387840 0.63827162
```

You can even access a specific subset of values by wrapping a vector in the square brackets:

```
subset2 <- random_data[c(1,10,100)] #Get the first, tenth and 100th values

print(subset2)
```

```
## [1] 0.40308759 0.09141181 0.29569662
```

A subset of a vector is just a shorter vector. In fact, singular values are technically vectors of length 1, so all of the values we've used up till now were vectors all along! You can check the length of a vector with the length() function:

```
length(10)  # A singular value is a vector of length 1
```

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

```
length(random_data)
```

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

Here are a few other useful ways to index into vectors:

```
# Adding a minus sign excludes a given index:

y <- c("Life","Is","Study")
y <- y[-2]            # Exclude index 2
print(y)
```

```
## [1] "Life"  "Study"
```

```
# A minus sign can also exclude a given range of indices:

random_data <- runif(50)                # Generate 50 random numbers
random_data_sub <- random_data[-(2:49)] # Exclude the range 2 through 49
print(random_data_sub)
```

```
## [1] 0.5070463 0.7861768
```

You can also index a vector with a logical vector of the same length. In this case, the subset is created from each index where the corresponding logical vector is TRUE. Indexing with a logical vector is a common way to filter a numeric or character vector for values that fulfill certain criteria:

```
# Create a logical vector identifying values over 0.5 in random_data

logical_over_half <- (random_data > 0.5)
print(logical_over_half)
```

```
##  [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
## [12]  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
## [23]  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
## [34] FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE
## [45]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

```
 # Use the logical vector to create a subset of the values over 0.5
over_half <- random_data[logical_over_half]

print(over_half)
```

```
##  [1] 0.5070463 0.6895292 0.8167473 0.6847566 0.6403038 0.8143138 0.9689064
##  [8] 0.7215140 0.8243902 0.6248586 0.7186652 0.9884088 0.5469072 0.8231311
## [15] 0.7928608 0.7409424 0.8506990 0.8206447 0.6892814 0.5536946 0.9325937
## [22] 0.6014662 0.8046209 0.9041584 0.7260342 0.5092592 0.5485366 0.9964490
## [29] 0.9963518 0.9556758 0.5970786 0.7861768
```

```
 # Use the logical vector and the not symbol (!) to get values under 0.5

under_half <- random_data[!logical_over_half]

print(under_half)
```

```
##  [1] 0.45742447 0.32449809 0.01711891 0.16393128 0.44190229 0.06834795
##  [7] 0.38133717 0.29675896 0.34852461 0.46607750 0.02195595 0.31568054
## [13] 0.38316312 0.28814349 0.03340180 0.11903649 0.37545148 0.36858961
```

```
 # You can perform logical indexing all in one step:

random_data[random_data > 0.5]
```

```
##  [1] 0.5070463 0.6895292 0.8167473 0.6847566 0.6403038 0.8143138 0.9689064
##  [8] 0.7215140 0.8243902 0.6248586 0.7186652 0.9884088 0.5469072 0.8231311
## [15] 0.7928608 0.7409424 0.8506990 0.8206447 0.6892814 0.5536946 0.9325937
## [22] 0.6014662 0.8046209 0.9041584 0.7260342 0.5092592 0.5485366 0.9964490
## [29] 0.9963518 0.9556758 0.5970786 0.7861768
```

```
 # You can also use more complicated logical expressions. # In this case we grab all values between 0.4

random_data[(random_data < 0.6) & (random_data > 0.4)]
```

```
## [1] 0.5070463 0.4574245 0.4419023 0.5469072 0.5536946 0.4660775 0.5092592
## [8] 0.5485366 0.5970786
```

Finally, you can use %in% to create a subset of elements that are contained within some other vector:

```
my_vector <- c("a","b","c","d","a","a","f")

my_vector[my_vector %in% c("a","c")]
```

```
## [1] "a" "c" "a" "a"
```

Vectorized Operations

One of the biggest benefits of R is that it is built around performing operations on vectors. Many R functions and operations behave in a "vectorized" manner, meaning they act upon each element of a vector individually and return the result of each of the operations in a new vector. Vectorized operations simplify the process of

performing the same calculations on related data. All the basic operators and functions we've learned so far that operate on single values work on vectors longer than length 1.

```
example_vector <- c(1,2,3)

# + adds to each value in the vector
example_vector + 10
```

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

```
# - performs subtraction on each value
example_vector - 10
```

```
## [1] -9 -8 -7
```

Other math operators like *, /, ^ and %% work the same way as do functions like round(), floor() and ceiling():

```
example_vector2 <- c(1.6, 2.5, 3.5)

round(example_vector2)
```

```
## [1] 2 2 4
```

```
floor(example_vector2)
```

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

Vectorized operations make it easy to carry out vector transformations quickly without worrying about programming constructs like for and while loops. Vector operations that involve two or more vectors are typically executed in an element-wise fashion. For example, if you take two numeric vectors of the sample length and add them, the result is a new vector containing the sums of the values at each index:

```
vector1 <- c(1,2,3,4)
vector2 <- c(10,20,30,40)

print( vector1+vector2 )
```

```
## [1] 11 22 33 44
```

```
# Other math operations also work in this way:

vector1*vector2  # Element-wise multiplication
```

```
## [1]  10  40  90 160
```

```
vector1/vector2  # Element-wise division
```

```
## [1] 0.1 0.1 0.1 0.1
```

```
vector1 %% vector2  # Element-wise modulus
```

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

```
# If you want a vector inner product, use %*%

vector1 %*% vector2
```

```
##      [,1]
## [1,]  300
```

*Note: An inner product is the sum of the element-wise multiplication of two vectors. It always returns a single value. Vectorized operations can also work on character vectors. Let's consider the function paste()

which takes two or more objects as input and concatenates them into a character vector. If you pass paste() character vectors longer than length 1, it combines them in an element-wise fashion:

```
x <- c("Life","Is","Study")
y <- c("Blogging","Is","Fun")

paste(x,y)
```

```
## [1] "Life Blogging" "Is Is"         "Study Fun"
```

The data type conversion functions.

```
x <- c(1,2,3)
print(x)
```

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

```
typeof(x)
```

```
## [1] "double"
```

```
x <- as.character(x)
print(x)
```

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

```
typeof(x)
```

```
## [1] "character"
```

Generating Vectors

Creating vectors by hand with the c() function works fine for short vectors, but it becomes cumbersome quickly when you're working with longer vectors. R includes a variety of convenience functions to generate vectors. You can generate all whole numbers in a range using a colon:

```
x <- 1:20
print(x)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

You can also generate sequences using the seq() function. Seq takes the arguments from, to, and by which specify the starting point, stopping point and size of the sequence increment:

```
y <- seq(from = 1, to = 20, by = 1)
print(y)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
z <- seq(0, 100, 10)   # You can omit the argument names
print(z)
```

```
##  [1]   0  10  20  30  40  50  60  70  80  90 100
```

Use rep() to create a vector of the same value repeated a specified number of times:

```
r <- rep(x=1, times=20)
print(r)
```

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

As we saw earlier, you can use the runif() function to draw random values from specified range:

```r
x <- runif(n=20, min=0, max=100)
print(x)
```

```
##  [1] 11.426977 75.176831 41.025707 20.286739 47.117117 58.283705 14.820375
##  [8] 97.454623 58.850560  1.757481 88.841417 78.566963 68.142395 91.709181
## [15] 80.938103 83.119280 28.616023 51.778521 54.613125 64.122732
```

The function runif() draws numbers from a uniform distribution, so all values within the range are equally likely. R also has functions for drawing random numbers from other types of distributions, such as rnorm() for the normal distribution, rexp() for the exponential distribution and rbinom() for the binomial distribution. We won't go into these any further right now, but suffice it to say R is very useful if you have to deal with probability distributions. You can accomplish a surprising amount in R using only vectors and vector commands in the console, but real-world data is usually structured in 2 dimensional tables.