

Operations and Vetores

João Vieira & Pedro Fonseca



Introduction to R Programming

May 1, 2020



- ① Vectors
- ② Vectorized Operations
- ③ Named Vectors



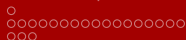
The `c` Operator

- Vectors are essential building blocks for handling multiple items in R.
- To create vectors use the *combine* operator (`c`):

```
> (myvec <- c(1, 3, 1, 42))  
[1] 1 3 1 42
```

```
> (myvec2 <- c(myvec, x, y ,z))  
[1] 1 3 1 42 -5 10 -50
```

```
> (myvec3 <- c(myvec, 1, 2))  
[1] 1 3 1 42 1 2
```



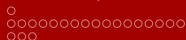
Subsetting



- Get the first element:

```
> myvec[1]  
[1] 1
```
- Get the second element:

```
> myvec[2]  
[1] 3
```



Subsetting

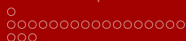


- Get the first three elements:

```
> myvec[1:3]  
[1] 1 3 1
```
- Omit the first element:

```
> myvec[-1]  
[1] 3 1 42
```
- Omit more than one element:

```
> myvec[-c(1,2)]  
[1] 1 42
```



Overwriting



- Substitute an element:

```
> myvec[3] <- 6
```

```
> myvec
```

```
[1] 1 3 6 42
```

- Substitute more than one element:

```
> myvec[c(2,3,4)] <- c(2,3,4)
```

```
> myvec
```

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



Functions to Generate Vectors

- Different ways to make a sequence:

```
> 1:10
```

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

```
> 5:1
```

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

```
> seq(1, 10)
```

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

```
> seq(from = 18, to = 27, by = 3)
```

```
[1] 18 21 24 27
```



Functions to Generate Vectors

```
> rep(x = 1, times = 4)
```

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

```
> rep(c(3, 6), times = 3)
```

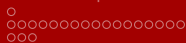
```
[1] 3 6 3 6 3 6
```

```
> rep(c(3, 62, 8.3), each = 2)
```

```
[1] 3.0 3.0 62.0 62.0 8.3 8.3
```

```
> rep(c(3, 6), times = 3, each = 2)
```

```
[1] 3 3 6 6 3 3 6 6 3 3 6 6
```

Sorting the Elements of a Vector

- Sorting a vector in increasing or decreasing order:

```
> myvec2 <- c(1, 3, 1, 42, -5, 10, -50)
```

```
> myvec2
```

```
[1] 1 3 1 42 -5 10 -50
```

```
> sort(myvec2)
```

```
[1] -50 -5 1 1 3 10 42
```

```
> sort(myvec2, decreasing = TRUE)
```

```
[1] 42 10 3 1 1 -5 -50
```



Sorting the Elements of a Vector

```
> sort(c(2.5, -1, -10, 3.44))  
[1] -10.00 -1.00  2.50  3.44
```

```
> sort(c(2.5,-1,-10,3.44), decreasing = TRUE)  
[1]  3.44  2.50 -1.00 -10.00
```

```
> sort(c(2.5,-1,-10,3.44), TRUE)  
[1]  3.44  2.50 -1.00 -10.00
```

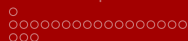


Some Statistical Functions

- *rnorm*(*n*) generates *n* pseudo-random numbers from a normal distribution (default: $\mu = 0$, $\sigma = 1$)

```
> rnorm(3)
[1] -0.5604756 -0.2301775  1.5587083
```

```
> rnorm(4, mean = 5, sd = 2)
[1] 5.1865810 5.6658985 3.8355640 0.8719963
```
- Other functions related do the normal distribution: *dnorm* (density), *pnorm* (distribution function), *qnorm* (quantile function).
- Equivalent functions are available for the most commonly used probability distributions: F, t-student, Uniform, Poisson...



The *set.seed* Function



- Functions like *rnorm*, *rpois* and *runif* generate pseudo-random numbers. This means that you and I will get different results when using these functions. Solution: use the *set.seed* function.
- Try this command many times:

```
> rnorm(2)
```
- Each time you will get a different output. Now try this:

```
> set.seed(123)  
> rnorm(2)
```
- You will get the same output every time.
- The argument of *set.seed* is irrelevant as long as we all use the same value.

Main Ideas



- One of the main advantages of R is vectorized calculation. This means that:
 - ▶ Most R functions accept vectors as inputs;
 - ▶ Vector arithmetic is performed element-wise by default.
- Vectorization calculation is a huge advantage efficiency and parsimony.
- Vectorization also makes code easier to write and read.

Examples



```
> x <- c(1, 2, 3)
> y <- c(0.5, 0.5, 0.5)
```

```
> 1/x
[1] 1.0000000 0.5000000 0.3333333
```

```
> 3+y
[1] 3.5 3.5 3.5
```

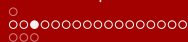
```
> x+y
[1] 1.5 2.5 3.5
```



Examples

```
> x <- c(1, 2, 3)
> y <- c(0.5, 0.5, 0.5)

> x^y
[1] 1.000000 1.414214 1.732051
> sqrt(x)
[1] 1.000000 1.414214 1.732051
> 1/1:3
[1] 1.0000000 0.5000000 0.3333333
> seq(from = 2, to = 6, by = 2)/2
[1] 1 2 3
```



Examples



```
> x1 <- c(1, 5, 7)
> x2 <- rep(1, times = 3)

> log(x1)
[1] 0.000000 1.609438 1.945910

> log(x1) - x2
[1] -1.0000000  0.6094379  0.9459101

> x <- x1 + x2
> x
[1] 2 6 8
```


Rounding



- *round()* rounds the values in its first argument to the specified number of decimal places (default 0).

```
> set.seed(123)
```

```
> z <- rnorm(3)
```

```
> z
```

```
[1] -0.5604756 -0.2301775  1.5587083
```

```
> round(z, digits = 3)
```

```
[1] -0.560 -0.230  1.559
```

```
> round(z)
```

```
[1] -1  0  2
```

Rounding



```
> y <- c(3.271109, 3.374961, 2.313307, 4.837787)
```

```
> round(y, 2)
```

```
[1] 3.27 3.37 2.31 4.84
```



Statistical Functions

```
> z  
[1] -0.5604756 -0.2301775  1.5587083  
  
> abs(z) # Absolut value  
[1] 0.5604756 0.2301775 1.5587083  
  
> max(z)  
[1] 1.558708  
  
> min(z)  
[1] -0.5604756
```



Statistical Functions



```
> z  
[1] -0.5604756 -0.2301775  1.5587083  
  
> mean(z)  
[1] 0.2560184  
  
> median(z)  
[1] -0.2301775  
  
> sd(z)  
[1] 1.140186
```



Statistical Functions

```
> z  
[1] -0.5604756 -0.2301775  1.5587083  
  
> var(z)  
[1] 1.300025  
  
> sum(z)  
[1] 0.7680552  
  
> quantile(z, 0.5)  
      50%  
-0.2301775
```



The *which* Function



- The *which* function is useful to find which elements of a vector that verify a given condition:

```
> set.seed(123)
```

```
> vec <- rnorm(n = 10, mean = 2, sd = 1)
```

```
> round(vec, 2)
```

```
[1] 1.44 1.77 3.56 2.07 2.13 3.72 2.46 0.73 1.31 1.55
```

```
> (indexes <- which(vec > 2))
```

```
[1] 3 4 5 6 7
```

```
> round(vec[indexes], 3)
```

```
[1] 3.559 2.071 2.129 3.715 2.461
```



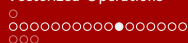
The *which* Function

```
> set.seed(123)

> vec2 <- rpois(n = 10, lambda = 2)
> which(vec2 == 2)
[1] 3 7 9 10

> (vec2 <- rpois(n = 10, lambda = 2))
[1] 1 3 2 4 4 0 2 4 2 2

> which(vec2 == 2)
[1] 3 7 9 10
```



The *which* Function

```
> set.seed(123)

> vec2 <- rpois(n = 10, lambda = 2)
> vec2
[1] 1 3 2 4 4 0 2 4 2 2

> max(vec2)
[1] 4

> which(vec2 == max(vec2))
[1] 4 5 8
```




The *which* Function

- The *which* function gives the positions of the elements of the vectors that verify the condition, not their values!

```
> set.seed(123)
```

```
> vec2 <- rpois(n = 10, lambda = 2)
```

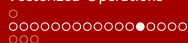
```
> vec2
```

```
[1] 1 3 2 4 4 0 2 4 2 2
```

- What are the actual values of *vec2* (not their positions) that verify the condition?

```
> vec2[which(vec > 1)]
```

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



The *length* Function

```
> round(vec[which(vec > 2)], 3)
[1] 3.559 2.071 2.129 3.715 2.461
```

- Use *length()* to obtain the number of elements in a vector:

```
> length(vec)
[1] 4
```

- How many elements of *vec* are greater than 2?

```
> length(which(vec > 1))
[1] 2
```



Trigonometric Functions

- R trigonometric take radians as argument, not degrees:

- ▶ $\sin(\frac{\pi}{2})$:

```
> sin(pi/2)
```

```
[1] 1
```

- ▶ $\cos(\pi)$:

```
> cos(pi)
```

```
[1] -1
```

- ▶ $\tan(\frac{\pi}{3})$:

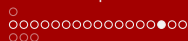
```
> tan(pi/3)
```

```
[1] 1.732051
```

- ▶ $\cotangent(\frac{\pi}{3})$:

```
> 1/tan(pi/3)
```

```
[1] 0.5773503
```



Trigonometric Functions



- Which value has a cosine = -1?
 `> acos(-1)`
 `[1] 3.141593`
- Which value has a tangent = 0.5?
 `> atan(0.5)`
 `[1] 0.4636476`

 `> tan(0.4636476)`
 `[1] 0.5`



Trigonometric Functions

- Trigonometric functions are also vectorized:

```
> (x <- seq(from = 0.25, to = 1, by = 0.25))  
[1] 0.25 0.50 0.75 1.00
```

```
> cos(x)  
[1] 0.9689124 0.8775826 0.7316889 0.5403023
```

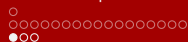
```
> 1/tan(x) # cotangent of x  
[1] 3.9163174 1.8304877 1.0734261 0.6420926
```

```
> cos(x)/sin(x) # cotangent of x  
[1] 3.9163174 1.8304877 1.0734261 0.6420926
```

Trigonometric Functions



- R has many more trigonometric functions. Try:
 > ?Trig



Recycling



- What happens when we conduct calculations with two vectors of different length?

```
> myvec <- c(1, 2, 3, 4)
> myvec2 <- rep(0.5, times = 8)

> myvec + myvec2
[1] 1.5 2.5 3.5 4.5 1.5 2.5 3.5 4.5
```

Recycling



```
> myvec3 <- rep(0.5, times = 7)
```

```
> myvec + myvec3
```

```
[1] 1.5 2.5 3.5 4.5 1.5 2.5 3.5
```

Warning message:

In myvec + myvec3 :

longer object length is not a multiple of shorter
object length

Recycling



- When conducting operations that require input vectors to be of the same length, R automatically recycles, or repeats, the shorter one, until it is long enough to match the longer one.
- It will only throw an error message if the length of the shorter vector is not a multiple of the vector of the larger vector.

Vectors Names



- We can also name the elements of a vector:

```
> x <- c(x1 = 1, x2 = 4, x3 = 7)
> x
```

```
x1 x2 x3
1  4  7
```

- Get the names of a vector:

```
> names(x)
[1] "x1" "x2" "x3"
```

Vectors Names



- The *names* function can also be used to provide names to a vector:

```
> y <- 1:3  
> names(y) <- c("y1", "y2", "y3")
```

```
> y  
y1 y2 y3  
1  2  3
```

Subsetting Named Vectors



- Vectors can also be subsetted by name:

```
> y
y1 y2 y3
 1  2  3
```

```
> y["y1"]
y1
 1
```

```
> y[c("y1", "y3")]
y1 y3
 1  3
```



The *paste* and *paste0* functions

- *paste* and *paste0* can be useful to generate vector names:

```
> paste("y", 1:length(y), sep = "")
```

```
[1] "y1" "y2" "y3"
```

```
> paste("name", 1:length(y), sep = "_")
```

```
[1] "name_1" "name_2" "name_3"
```

```
> paste("year", 1990:1993, sep = "-")
```

```
[1] "year-1990" "year-1991" "year-1992" "year-1993"
```

```
> paste0("X", 1:5)
```

```
[1] "X1" "X2" "X3" "X4" "X5"
```

Questions?



“The man who asks a question is a fool for a minute, the man who does not ask is a fool for life.”

— Confucius