Numeric Vectors

STAT 133

Gaston Sanchez

`github.com/ucb-stat133/stat133-fall-2016`

Data Types and Structures

To make the best of the R language, you'll need a strong understanding of the basic **data types** and **data structures** and how to operate on them.

Vectors

Vectors Reminder

- A vector is the most basic data structure in R
- Vectors are contiguous cells containing data
- Can be of any length (including zero)
- R has five basic type of vectors:
  integer, double, complex, logical, character
- vectors are **atomic** structures
- the values in a vector must be ALL of the same type

Vectors

The most simple type of vectors are scalars or single values:

```
# integer
x <- 1L
# double (real)
y <- 5
# complex
z <- 3 + 5i
# logical
a <- TRUE
# character
b <- "yosemite"
```

Vectors

The function to create a vector from individual values is `c()`, short for **concatenate**:

```
# some vectors
x <- c(1, 2, 3, 4, 5)

y <- c("one", "two", "three")

z <- c(TRUE, FALSE, FALSE)
```

Atomic Vectors

If you mix different data values, R will coerce them so they are all of the same type

```r
# mixing numbers and characters
x <- c(1, 2, 3, "four", "five")

# mixing numbers and logical values
y <- c(TRUE, FALSE, 3, 4)

# mixing numbers and logical values
z <- c(TRUE, FALSE, "TRUE", "FALSE")

# mixing integer, real, and complex numbers
w <- c(1L, -0.5, 3 + 5i)
```

Vectors of a given class

Sometimes is useful to initialize vectors of a particular class by simply specifying the number of elements:

```
# five element vectors
int <- integer(5)
num <- numeric(5)
comp <- complex(5)
logi <- logical(5)
char <- character(5)
```

Vector class functions

- `integer()`, `is.integer()`, `as.integer()`
- `numeric()`, `is.numeric()`, `as.numeric()`
- `complex()`, `is.complex()`, `as.complex()`
- `logical()`, `is.logical()`, `as.logical()`
- `character()`, `is.character()`, `as.character()`

Numeric Vectors

Vectors of sequence of **integers** can be created with the colon operator ":"

```
# positive: from 1 to 5
1:5

# negative: from -7 to -2
-7:-2

# decreasing: from 3 to -3
3:-3
```

Numeric Vectors

More vectors of numeric sequences (not only integers) can be created with the function `seq()`

```
# sequences
seq(1)
seq(from = 1, to = 5)
seq(from = -3, to = 9)
seq(from = -3, to = 9, by = 2)
seq(from = -3, to = 3, by = 0.5)
seq(from = 1, to = 20, length.out = 5)
```

Sequence generation

Two sequencing variants of `seq()` are `seq_along()` and `seq_len()`

- `seq_along()` returns a sequence of integers of the same length as its argument
- `seq_len()` generates a sequence from 1 to the value provided

Sequence generation

```r
# some flavors
flavors <- c("chocolate", "vanilla", "lemon")

# sequence of integers from flavors
seq_along(flavors)

## [1] 1 2 3

# sequence from 1 to 5
seq_len(5)

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

Replicate elements

Another way to create vectors is with the replicating function `rep()`

```
rep(1, times = 5)
rep(c(2, 4, 6), times = 2)
rep(1:3, times = c(3, 2, 1))
rep(c(2, 4, 6), each = 2)
rep(c(2, 4, 6), length.out = 5)
rep(c(2, 4, 6), each = 2, times = 2)
```

Random Vectors

R provides a series of random number generation functions that can also be used to create numeric vectors

generator	distribution
`runif()`	uniform
`rnorm()`	normal
`rbinom()`	binomial
`rbeta()`	beta
`rgamma()`	gamma
`rgeom()`	geometric

Check `help(?Distributions)` to see the list of all the available distributions

Random Vectors

```r
runif(n = 5, min = 0, max = 1)

rnorm(n = 5, mean = 0, sd = 1)

rbinom(n = 5, size = 1, prob = 0.5)

rbeta(n = 5, shape1 = 0.5, shape2 = 0.5)
```

Sampled Vectors

There's also the function `sample()` that generates random samples (with and without replacement)

```r
# shuffle
sample(1:10, size = 10)

# sample with replacement
values <- c(2, 3, 6, 7, 9)
sample(values, size = 20, replace = TRUE)
```

Vector Functions

Basic Vector Functions

- `length()`
- `sort()`
- `rev()`
- `order()`
- `unique()`
- `duplicated()`

Basic Vector Functions

```r
# numeric vector
num <- c(9, 4, 5, 1, 4, 1, 4, 7)

# how many elements?
length(num)
```

```
## [1] 8
```

```r
# sorting elements
sort(num)
```

```
## [1] 1 1 4 4 4 5 7 9
```

```r
sort(num, decreasing = TRUE)
```

```
## [1] 9 7 5 4 4 4 1 1
```

Basic Vector Functions

```
# reversed elements
rev(num)

## [1] 7 4 1 4 1 5 4 9

# position of sorted elements
order(num)

## [1] 4 6 2 5 7 3 8 1

order(num, decreasing = TRUE)

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

Basic Vector Functions

```
# unique elements
unique(num)

## [1] 9 4 5 1 7

# duplicated elements
duplicated(num)

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

num[duplicated(num)]

## [1] 4 1 4
```

Math Operations

Arithmetic Operators

operation	usage
unary +	`+ x`
unary -	`- x`
sum	`x + y`
subtraction	`x - y`
multiplication	`x * y`
division	`x / y`
power	`x ^ y`
modul0 (remainder)	`x %% y`
integer division	`x %/% y`

Arithmetic Operators

```
+2
-2
2 + 3
2 - 3
2 * 3
2 / 3
2 ^ 3
2 %% 3
2 %/% 3
```

Math Functions

- `abs()`, `sign()`, `sqrt()`
- `ceiling()`, `floor()`, `trunc()`, `round()`, `signif()`
- `cummax()`, `cummin()`, `cumprod()`, `cumsum()`
- `log()`, `log10()`, `log2()`, `log1p()`
- `sin()`, `cos()`, `tan()`
- `acos()`, `acosh()`, `asin()`, `asinh()`, `atan()`, `atanh()`
- `exp()`, `expm1()`
- `gamma()`, `lgamma()`, `digamma()`, `trigamma()`

Math Functions

```
abs(c(-1, -0.5, 3, 0.5))
```

```
## [1] 1.0 0.5 3.0 0.5
```

```
sign(c(-1, -0.5, 3, 0.5))
```

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

```
round(3.14159, 1)
```

```
## [1] 3.1
```

```
log10(10)
```

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

Vectorization

Vectorized Operations

A vectorized computation is any computation that when applied to a vector operates on all of its elements

```
c(1, 2, 3) + c(3, 2, 1)
```

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

```
c(1, 2, 3) * c(3, 2, 1)
```

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

```
c(1, 2, 3) ^ c(3, 2, 1)
```

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

Vectorization

All arithmetic, trigonometric, math and other vector functions are vectorized:

```
log(c(1, 2, 3))

## [1] 0.0000000 0.6931472 1.0986123

cos(seq(1, 3))

## [1]  0.5403023 -0.4161468 -0.9899925

sqrt(1:3)

## [1] 1.000000 1.414214 1.732051
```

Recycling

When vectorized computations are applied, some problems may occur when dealing with two vectors of different length

```
c(2, 1) + c(1, 2, 3)

## Warning in c(2, 1) + c(1, 2, 3):  longer object length
is not a multiple of shorter object length

## [1] 3 3 5
```

Recycling Rule

The recycling rule states that the shorter vector is replicated enough times so that the result has the length of the longer vector

```
c(1, 2, 3, 4) + c(2, 1)
```

```
## [1] 3 3 5 5
```

```
1:10 * 1:5
```

```
##  [1]  1  4  9 16 25  6 14 24 36 50
```

Recycling Rule

The Recycling Rule can be very useful, like when operating between a vector and a "scalar"

```
x <- c(2, 4, 6, 8)
x + 3  # add 3 to all elements in x

## [1]  5  7  9 11

x / 3  # divide all elemnts by 3

## [1] 0.6666667 1.3333333 2.0000000 2.6666667

x ^ 3  # all elements to the power of 3

## [1]   8  64 216 512
```

Comparison Operators

operation	usage
less than	`x < x`
greater than	`x > y`
less than or equal	`x <= y`
greater than or equal	`x >= y`
equality	`x == y`
different	`x != y`

Comparison operators produce logical values

Comparison Operators

```
5 > 1
5 < 7
5 > 10
5 >= 5
5 <= 5
5 == 5
5 != 3
5 != 5
```

```
TRUE > FALSE
TRUE < FALSE
TRUE == TRUE
TRUE != FALSE
TRUE != TRUE
```

Comparison Operators

Comparison Operators are also vectorized

```
values <- -3:3

values > 0

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

values < 0

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

values == 0

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

Comparison operators and recycling rule

```
c(1, 2, 3, 4, 5) > 2
```

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

```
c(1, 2, 3, 4, 5) >= 2
```

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

```
c(1, 2, 3, 4, 5) < 2
```

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

Comparison operators and recycling rule

```
c(1, 2, 3, 4, 5) <= 2
```

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

```
c(1, 2, 3, 4, 5) == 2
```

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

```
c(1, 2, 3, 4, 5) != 2
```

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

Comparison operators

When comparing vectors of different types, one is coerced to the type of the other, the (decreasing) order of precedence being character, complex, numeric, integer, logical

```
'5' == 5
```

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

```
5L == 5
```

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

```
5 + 0i == 5
```

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

Comparison Operators

In addition to comparison operators, we have the functions `all()` and `any()`

```
all(c(1, 2, 3, 4, 5) > 0)

all(c(1, 2, 3, 4, 5) > 1)

any(c(1, 2, 3, 4, 5) < 0)

any(c(1, 2, 3, 4, 5) > 4)
```

Summary Functions

- `max()` maximum
- `min()` minimum
- `range()` range
- `mean()` mean
- `var()` variance
- `sd()` standard deviation
- `prod()` product of all elements
- `sum()` sum of all elements

Summary Functions

```
x <- 1:7
max(x)
min(x)
range(x)
mean(x)
var(x)
sd(x)
prod(x)
sum(x)
```

Logical Operators

operation	usage
NOT	`!x`
AND (elementwise)	`x & y`
AND (1st element)	`x && y`
OR (elementwise)	`x | y`
OR (1st element)	`x || y`
exclusive OR	`xor(x, y)`

Logical operators act on logical and number-like vectors

Logical Operators

```
!TRUE
!FALSE
TRUE & TRUE
TRUE & FALSE
FALSE & FALSE
TRUE | TRUE
TRUE | FALSE
FALSE | FALSE
xor(TRUE, FALSE)
xor(TRUE, TRUE)
xor(FALSE, FALSE)
```

Logical and Comparison Operators

Many operations involve using logical and comparison operators:

```
x <- 5

(x > 0) & (x < 10)
(x > 0) | (x < 10)
(-2 * x > 0) & (x/2 < 10)
(-2 * x > 0) | (x/2 < 10)
```

`which()` functions

- `which()`: which indices are `TRUE`
- `which.min()`: location of first minimum
- `which.max()`: location of first maximum

Other Functions

```
(values <- -3:3)
```

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

```
# logical comparison
values > 0
```

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

```
# positions (i.e. indices) of positive values
which(values > 0)
```

```
## [1] 5 6 7
```

Function `which()`

```
# indices of various comparisons
which(values > 0)

## [1] 5 6 7

which(values < 0)

## [1] 1 2 3

which(values == 0)

## [1] 4
```

Function `which()`

```
# logical comparison
values > 0
```

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

```
# logical subsetting
values[values > 0]
```

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

```
# positions of positive values
which(values > 0)
```

```
## [1] 5 6 7
```

```
# numeric subsetting
values[which(values > 0)]
```

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

which.max() and which.min()

```
which.max(values)
```

```
## [1] 7
```

```
which(values == max(values))
```

```
## [1] 7
```

```
which.min(values)
```

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

```
which(values == min(values))
```

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

Set Operations

Functions to perform set union, intersection, (asymmetric!) difference, equality and membership on two vectors

- `union(x, y)`
- `intersect(x, y)`
- `setdiff(x, y)`
- `setequal(x, y)`
- `is.element(el, set)`
- `%in%` operator

Set Operations

```
x <- c(1, 2, 3, 4, 5)
y <- c(2, 4, 6)

union(x, y)
intersect(x, y)
setdiff(x, y)
setequal(x, y)
setequal(c(4, 6, 2), y)
is.element(1, x)
is.element(6, x)
3 %in% x
3 %in% y
```

General Functions

Functions for inspecting a vector

- `class(x)`
- `length(x)`
- `head(x)`
- `tail(x)`
- `summary(x)`

General Functions

```r
ages <- c(21, 28, 23, 25, 24, 26, 27, 21)

class(ages)
length(ages)
head(ages)
tail(ages)
summary(ages)
```

Exercise

Find out what the following expressions return:

```
1:1

seq(0, 1, length.out = 10)

rep(c(1, 2, 3), times = c(1, 2, 3))
```

Exercise

Write three different ways in which the vector `1, 2, 3, 4, 5` can be created:

Exercise

Write three different ways in which the vector `1, 2, 3, 4, 5` can be created:

```
c(1, 2, 3, 4, 5)
seq(from = 1, to = 5)
1:5

# another option
0:4 + 1
```

Exercise

Generate a random vector of `n=100` elements:

```
set.seed(1)
a <- rnorm(100)
```

Find the following:

- what's the vector class
- what's the mean and standard deviation
- what's the sum of all elements in absolute value
- how many elements are positive (≥ 0)
- how many elements are negative (< 0)
- the three smallest and largest numbers

Exercise

```r
# class
class(a)
```

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

```r
# mean value
mean(a)
```

```
## [1] 0.1088874
```

```r
# std dev
sd(a)
```

```
## [1] 0.8981994
```

```r
# sum of elems in abs-value
sum(abs(a))
```

```
## [1] 71.67207
```

```r
# how many positive
sum(a >= 0)
```

```
## [1] 54
```

```r
# how many negative
sum(a < 0)
```

```
## [1] 46
```

Exercise

```r
# 3 smallest numbers
sort(a)[1:3]

## [1] -2.214700 -1.989352 -1.804959

# 3 largest numbers
sort(a, decreasing = TRUE)[1:3]

## [1] 2.401618 2.172612 1.980400
```