Vectors (part 2)

R Data Objects

Gaston Sanchez

STAT 33B, Fall 2025

About

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.

Creating Vectors

Numeric Vectors

Vectors of numeric sequences (in one-unit steps) can be created with the colon operator `:`

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

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

# decreasing integers: from 3 to -3
3:-3

# non-integers
-2.5:2.5
```

Numeric Vectors

More vectors of numeric sequences (not just one-unit step sequences) can be created with the function `seq()`

```
# sequences
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
```

```r
# 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()` that allows you to create several repetition patterns.

You have to provide an input vector, and use one or more of the arguments `times`, `each` and `length.out`

```
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, for example:

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

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

Random Vectors

```
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()`: number of elements in a vector
- `sort()`: arranges elements
- `rev()`: reverses elements
- `order()`: index of arranged vector
- `unique()`: gives unique elements
- `duplicated()`: indicates which elements are 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`
modulo (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
```

Comparison Operators

Operator	Description
`x == y`	equal
`x != y`	not equal
`x < y`	less than
`x > y`	greater than
`x <= y`	less than or equal
`x >= y`	greater than or equal

Comparison operators produce logical values

Comparison Operators

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

Comparison Operators

```
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)
```

Logical Operators

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

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)
```

Summary Statistic Functions

Operator	Description
`max(x)`	maximum
`min(x)`	minimum
`range(x)`	range values
`mean(x)`	mean
`median(x)`	median
`var(x)`	variance
`sd(x)`	standard deviation
`IQR(x)`	interquartile range

Summary Statistic Functions

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

`which()` Functions

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

which() 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
```

which() Functions

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

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

```r
which(values < 0)
```

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

```r
which(values == 0)
```

```
[1] 4
```

which() Functions

```
# 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() Functions

```
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

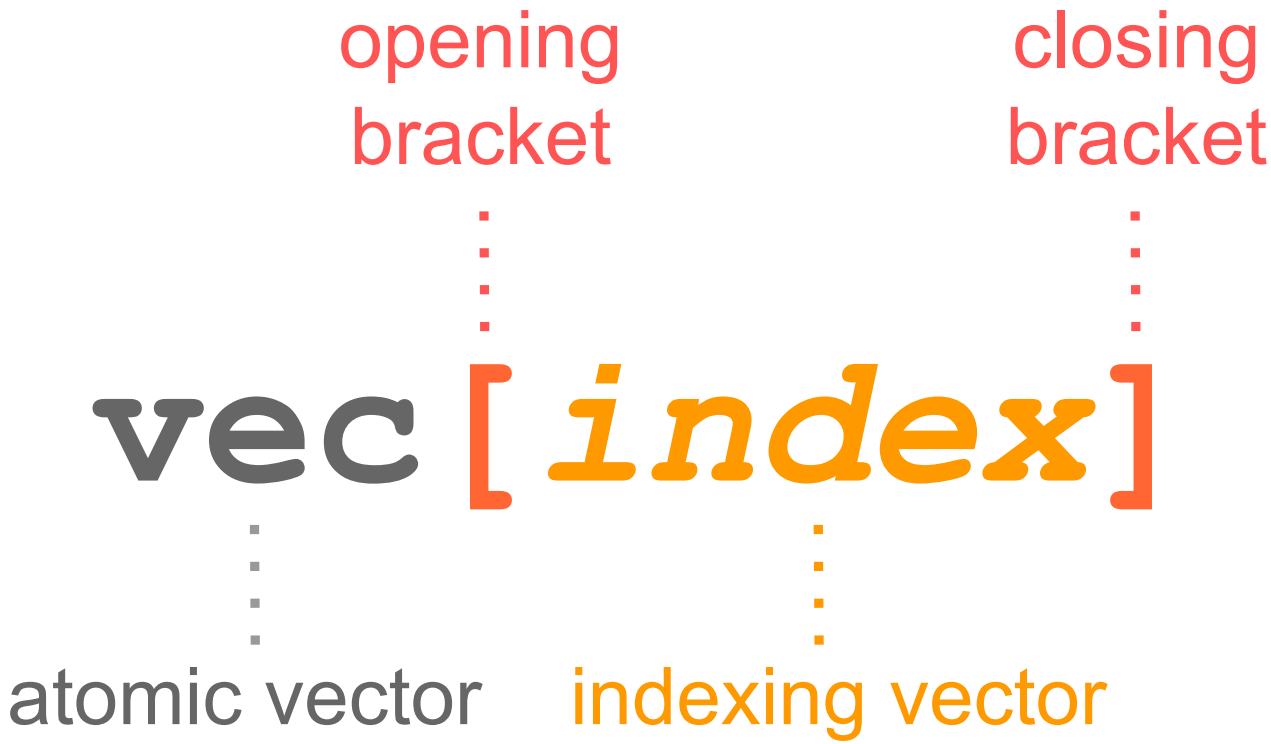
Set Operations

```r
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
```

Vector Manipulation

Subsetting: Bracket Notation



Bracket Notation System

- To extract values from R objects use brackets: `[ ]`
- Inside the brackets specify a vector of indices
- Vector of indices can be of type numeric (integer or double), logical, and sometimes character

Indexing

Consider the following vector `x`

```
# some vector
x <- c(2, 4, 6, 8)

# adding names
names(x) <- letters[1:4]


x
```

```
a b c d
2 4 6 8
```

Numeric Indexing

```
# first element
x[1]
```

```
a
2
```

```
# second element
x[2]
```

```
b
4
```

Numeric Indexing

```
# last element
x[length(x)]
```

```
d
8
```

```
# recall that first position is 1
x[0]
```

```
named numeric(0)
```

Numeric Indexing

```r
# first 3 elements
x[1:3]
```

```
a b c
2 4 6
```

```r
# non-consecutive elements
x[c(1, 3)]
```

```
a c
2 6
```

Numeric Indexing

```r
# different order
x[c(3, 2, 4, 1)]
```

```
c b d a
6 4 8 2
```

```r
# different order (and repetition)
x[c(3, 2, 4, 1, 1, 1)]
```

```
c b d a a a
6 4 8 2 2 2
```

Numeric Indexing

Negative numbers indicate exclusion

```
# exclude 2nd element
x[-2]
```

```
a c d
2 6 8
```

```
# exclude non-consecutive elements
x[-c(1, 3)]
```

```
b d
4 8
```

Logical Indexing (aka logical subsetting)

You can also use `logical` index vectors

```
# first element
x[c(TRUE, FALSE, FALSE, FALSE)]
```

```
a
2
```

```
# exclude first element
x[c(FALSE, TRUE, TRUE, TRUE)]
```

```
b c d
4 6 8
```

Note: you won't typically be typing logical vectors like in these examples; instead you'll be using logical vectors that result from logical operators or from comparison operators

Logical Indexing (aka logical subsetting)

Logical subsetting is extremely powerful

```
# elements equal to 2
x[x == 2]
```

```
a
2
```

```
# elements different to 2
x[x != 2]
```

```
b c d
4 6 8
```

Logical Indexing (aka logical subsetting)

```
# elements greater than 1
x[x > 1]
```

```
a b c d
2 4 6 8
```

```
# greater than 1 and less than or equal to 3
x[x > 1 & x <= 3]
```

```
a
2
```

Logical Indexing (aka logical subsetting)

```
# try this
x[TRUE]

# and this
x[FALSE]

# what about this?
x[as.logical(c(0, 1, pi, -10))]
```

Character Indexing (aka logical subsetting)

You can use a `character` index vector as long as `x` has named elements:

```
# element named "a"
x["a"]
```

```
a
2
```

```
# elements named "b" and "d"
x[c("b", "d")]
```

```
b d
4 8
```

Character Indexing (aka logical subsetting)

You can use a `character` index vector as long as `x` has named elements:

```
# repeated elements
x[rep("a", 5)]
```

```
a a a a a
2 2 2 2 2
```

More Indexing

This is less common but possible:

```
x[-1][2:3]
```

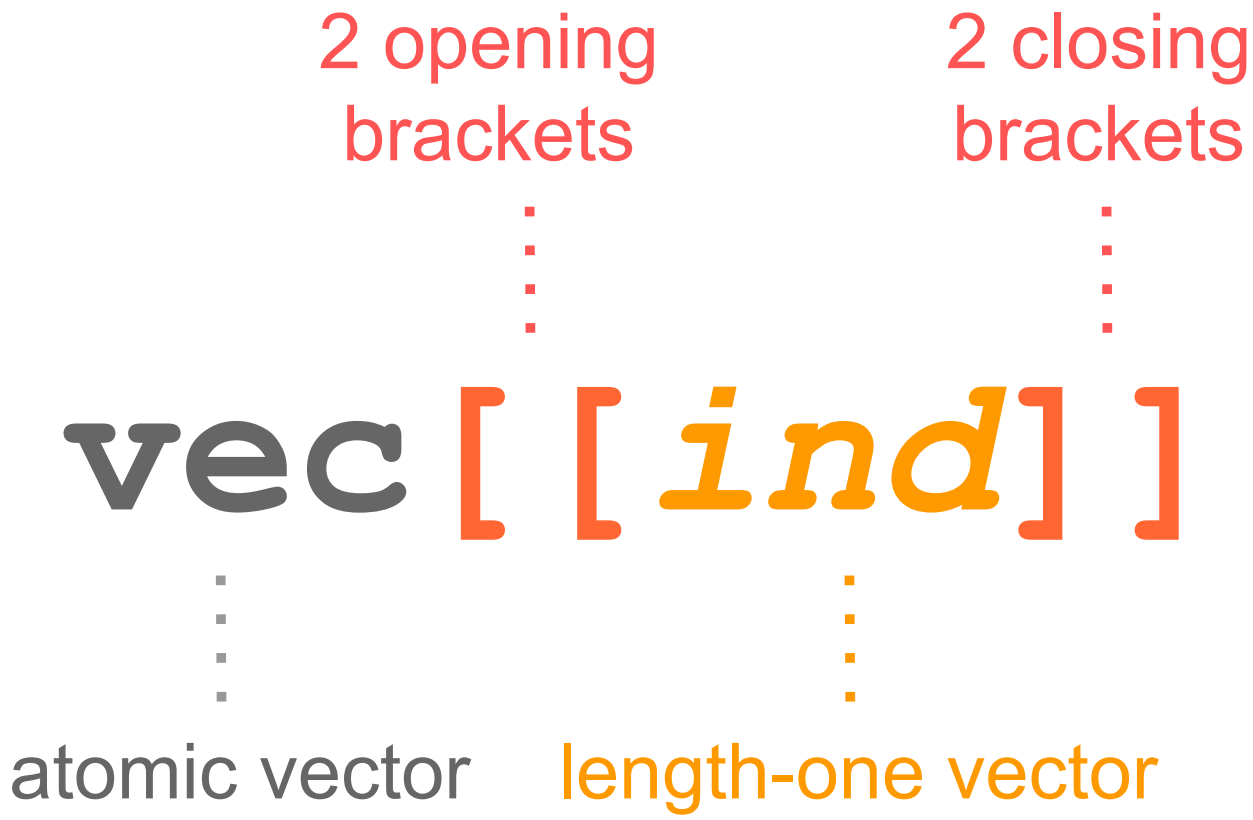
```
c d
6 8
```

```
x[-length(x)][-2][2]
```

```
c
6
```

Double Brackets

Double Bracket Notation



Double Brackets

You can also use double brackets `[[ ]]` but in this case only to extract a **single element**

```
x <- c(a = 2, b = 4, c = 6, d = 8)
x
```

```
a b c d
2 4 6 8
```

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

```
[1] 4
```

```
x[["a"]]
```

```
[1] 2
```

Double Brackets

With double brackets, you **cannot** pass an index vector of length greater than one

```
x[[1:2]]
```

```
Error in x[[1:2]] :
  attempt to select more than one element in vectorIndex
```