Chapter 2
Vectors, Matrices, and Arrays

Arthur Li

Vectors
Types of Vectors

❖**5** types:

logical | integer | real | complex | character

❖ cannot mixed type

❖ character strings – entered with “ ” or ‘ ’

❖ logical values: TRUE, FALSE, T, F

❖ missing value = NA

❖The simplest way to create a vector is to use the **c** function

Types of Vectors

```
> mydata = c(2.9, 3.5, 4.5, NA, 3, 2.4)
> mydata
[1] 2.9 3.5 4.5 NA 3.0 2.4
> mode(mydata)
[1] "numeric"
> class(mydata)
[1] "numeric"
```

```
> single = 5
> length(5)
[1] 1
```

Types of Vectors

- Numbers in R are stored in double precision real numbers

- To create an integer vector explicitly, use the L suffix.

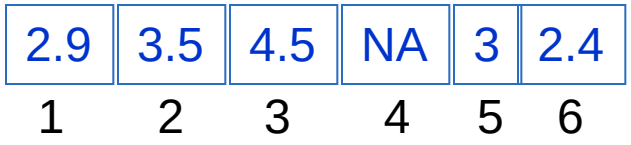
```
> int = c(1L, 3L, 10L)
> int
[1] 1 3 10
> mode(int)
[1] "numeric"
> class(int)
[1] "integer"
```

Types of Vectors

```
> colors = c("red", "green", "blue", "yellow", NA, "purple")
> colors
[1] "red" "green" "blue" "yellow" NA "purple"
>
> newLogic = c(TRUE, NA, T, F)
> newLogic
[1] TRUE NA TRUE FALSE
```

Accessing a Vector

```
> mydata = c(2.9, 3.5, 4.5, NA, 3, 2.4)
> mydata
[1] 2.9 3.5 4.5  NA 3.0 2.4
```

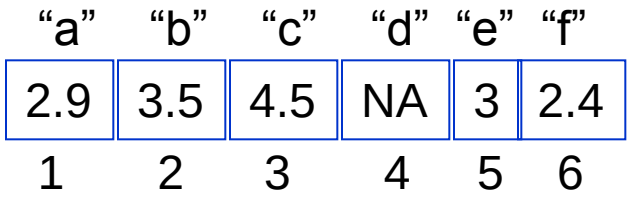



```
> mydata[3]
[1] 4.5
>
> mydata[2:5]
[1] 3.5 4.5  NA 3.0
>
> mydata[c(2,4,6)]
[1] 3.5  NA 2.4
```

The `names` Attribute of the Vector

❖ A vector can also be named and accessed by names

```
> names(mydata) = c("a", "b", "c", "d", "e", "f")
> mydata
  a   b   c   d   e   f
2.9 3.5 4.5  NA 3.0 2.4
```



❖ This vector has the **`name`** attibute

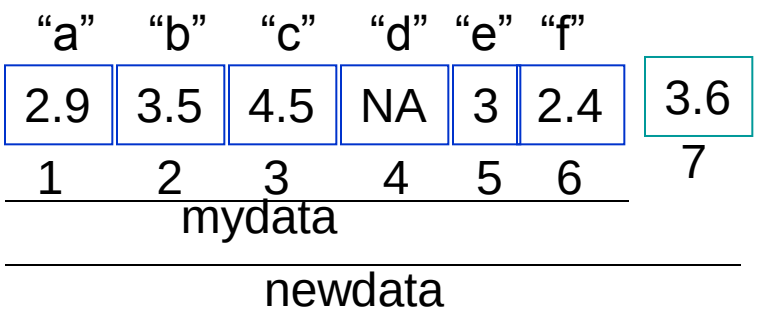
```
> names(mydata)
[1] "a" "b" "c" "d" "e" "f"
> mydata["a"]
  a
2.9
> mydata[c("a", "d")]
  a   d
2.9  NA
```

```
> letters[1:3]
[1] "a" "b" "c"
> mydata[letters[1:3]]
  a   b   c
2.9 3.5 4.5
```

Concatenating a Vector

♪ To add a component to a vector, we can use the “c” function

```
> newdata = c(mydata, 3.6)
> newdata
   a    b    c    d    e    f
2.9  3.5  4.5   NA  3.0  2.4  3.6
```

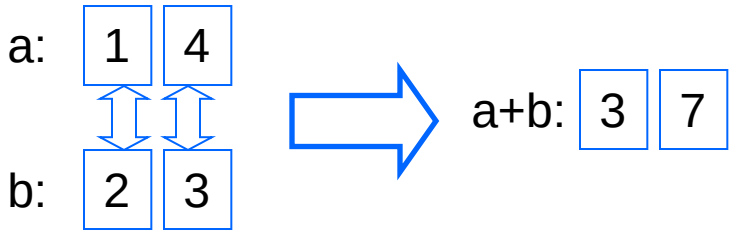



```
> newdata2 = c(mydata, newdata)
> newdata2
   a    b    c    d    e    f    a    b    c    d    e    f
2.9  3.5  4.5   NA  3.0  2.4  2.9  3.5  4.5   NA  3.0  2.4  3.6
```

The Recycling Rule

❖ Arithmetical operations performed on vectors, element by element

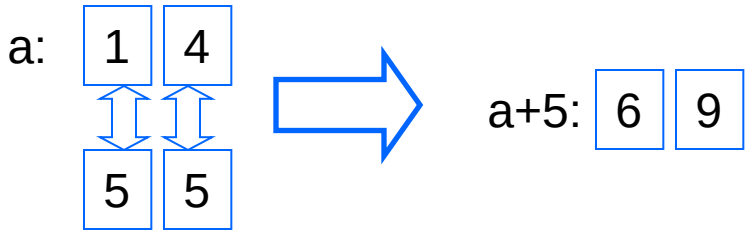
```
> a = c(1, 4)
> b = c(2, 3)
> a + b
[1] 3 7
```



❖ Applying an operator = calling a function: 3 + 2 = ‘+’ (3, 2)

❖ 2 vector of different length → shorter one is recycled

```
> a + 5
[1] 6 9
```



The Recycling Rule

```
> a + c(1, 2, 4)
[1] 2 6 5
Warning message:
In a + c(1, 2, 4) :
  longer object length is not a multiple of
shorter object length
```

Numeric Vectors
Generating Sequences By Using the `seq` Function

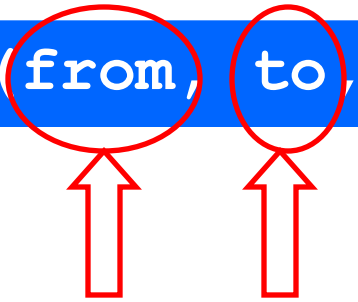
- Generating a sequence of number – using ":" operator

```
> 2:15
 [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
```

Generating Sequences By Using the `seq` Function

❖ Alternatively -- the **`seq`** function: takes 5 arguments

```
seq(from, to, by, length, along)
```



Beginning of the sequence

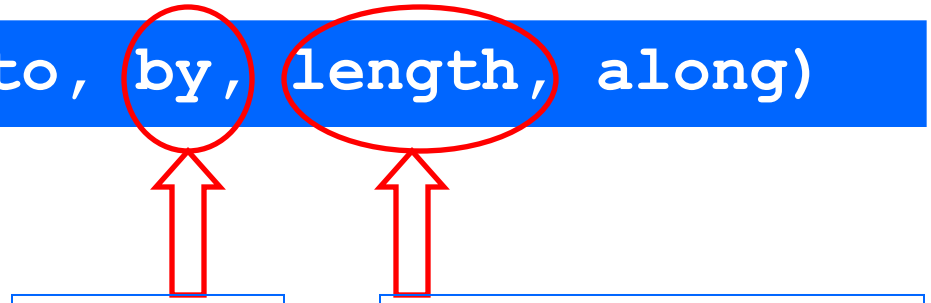
End of the sequence

```
> seq(2,10)
[1]  2  3  4  5  6  7  8  9 10
```

Generating Sequences By Using the `seq` Function

❖ Alternatively -- the **`seq`** function: takes 5 arguments

```
seq(from, to, by, length, along)
```



Can not use the same time

```
> seq(2, 20, by=3)
[1]  2  5  8 11 14 17 20
```

=

```
> seq(2, 20, length=7)
[1]  2  5  8 11 14 17 20
```

Generating Sequences By Using the `seq` Function

❖ Alternatively -- the **`seq`** function: takes 5 arguments

```
seq(from, to, by, length, along)
```

Argument: a vector

❖ Use **`along`** instead of **`to`** or **`length`** →

`length(vector) = length(result)`

```
> x = c(3, 5, 8, 10, 0.3, 4, 5)
> length(x)
[1] 7
> seq(2, along = x)
[1] 2 3 4 5 6 7 8
> seq(2, 20, along = x)
[1]  2  5  8 11 14 17 20
```

Generating Sequences By Using the `seq` Function

❖ Alternatively -- the **`seq`** function: takes 5 arguments

```
seq(from, to, by, length, along)
```

❖ **`seq`** has 1 unnamed argument, with length = 1 → **`from = 1`**

```
> seq(6)
[1] 1 2 3 4 5 6
```

❖ **`seq`** has 1 unnamed argument, with length > 1 → **`along =x`**

```
> x
[1]  3.0  5.0  8.0 10.0  0.3  4.0  5.0
> seq(x)
[1] 1 2 3 4 5 6 7
```

Generating Sequences By Using the `rep` Function

- The **`rep`** function: repeat an object

- 2 important arguments: **`x`** (a vector) & **`times`** (# of times)

- If **`times`** is an integer
  → repeat **`x`** # of **`times`**

```
> i = rep(2, 4)
> i
[1] 2 2 2 2
> x = 1:4
> rep(x, 2)
[1] 1 2 3 4 1 2 3 4
```

Generating Sequences By Using the `rep` Function

- The **`rep`** function: repeat an object

- 2 important arguments: **`x`** (a vector) & **`times`** (# of times)

- If **`times`** is an integer
  → repeat **`x`** # of **`times`**

```
> i = rep(2, 4)
> i
[1] 2 2 2 2
> x = 1:4
> rep(x, 2)
[1] 1 2 3 4 1 2 3 4
```

- If **`times`** is a vector, & **`length(times)= length(x)`**
  →repeat **`x[i] times[i]`** times

```
> rep(x, i)
[1] 1 1 2 2 3 3 4 4
> rep(x, x)
 [1] 1 2 2 3 3 3 4 4 4 4
```

Generating a Sequence of Number By Using the Random Number Generator Functions

- All the random number generator functions start **`r`**
- **`runif(n, min=0, max=1)`: Uniform**
- **`rnorm(n, mean=0, sd=1)` : Normal**
- Generate 100 number ~ Normal (2, 3)

```
> normData = rnorm(100, 2, 3)
> mean(normData)
[1] 2.042951
> sd(normData)
[1] 2.941787
```

The `numeric` Function

- **`numeric`**: create a double-precision vector of the specified length with each element equal to 0

```
> numeric(5)
[1] 0 0 0 0 0
> empty = numeric(0)
> empty
numeric(0)
> length(empty)
[1] 0

> character(5)
[1] "" "" "" "" ""
> character(0)
character(0)
> logical(5)
[1] FALSE FALSE FALSE FALSE FALSE
> logical(0)
logical(0)
```

Generating Logical Vectors

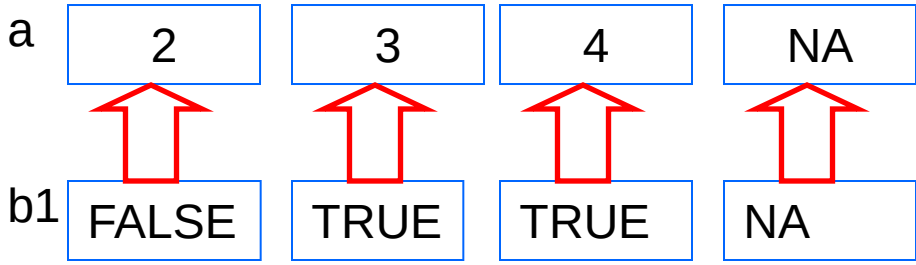
- To create a logical vector, we can use the **c** function

```
> z = c(TRUE, FALSE, T, F)
```

- Most of the time, it is generated by condition
- Relational operators: **<**, **<=**, **>**, **>=**, **==**, and **!=**
- Create a logical vector: indicating if **a > 3**

```
> a = c(seq(2, 4), NA)
```

```
> b1 = a > 2
> b1
[1] FALSE TRUE TRUE  NA
```



Generating Logical Vectors

- We can combine/negate conditions by using logical operators
- Logical operators: **&**, **|**, **!**
- The recycling rules applies to these operations

```
> b2 = a < 4
> b2
[1] TRUE TRUE FALSE NA
> b1 & b2
[1] FALSE TRUE FALSE NA
> b1 | b2
[1] TRUE TRUE TRUE NA
> !b1
[1] TRUE FALSE FALSE NA
```

Generating Logical Vectors

- **`is.na:`** examine if the element of a given vector is missing

- **`is.na(a) ≠ a == NA`**

- **`a == NA`** → creates a vector of the same length as **`a`** with values **`NA`**

```
> is.na(a)
[1] FALSE FALSE FALSE TRUE
> a == NA
[1] NA NA NA NA
```

Using Logical Vectors For Calculations

- In calculation: logical vector → numeric vector
  TRUE → 1, FALSE → 0

- Example: count # of elements in a vector > the mean

```
> g <- c(seq(1, 6, by = 0.5), 10)
> g
 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 10.0
> sum( g > mean(g) )
[1] 5
```

The `all` And `any` Functions

- **`any`**: test whether **at least** one element of a vector is TRUE

- **`all`**: test whether **all** the elements of a vector is TRUE

```
> f = c(3.01, 3.001, 3.0001, 3.00001)
> any(f == 3.001)
[1] TRUE
> all(f > 3)
[1] TRUE
> all(f > 3.001)
[1] FALSE
```

== and `identical` function

- **`identical`:** compare 2 objects, return 1 TRUR/FALSE

- == or !=: if arguments with length > 1, return a value with length > 1

```
> identical(3, c(3, 4))
[1] FALSE
> 3 == c(3, 4)
[1]  TRUE FALSE
```

Factors
Creating a Factor

- R has a factor class to store categorical data
- When stored values as a factor class, it requires much less storage space since R only stores each unique level once
- To create a factor, you can use the factor function

```
> countyVector = c("la", "sb", "la", "oc", "oc", "sb")
> countyVector
[1] "la" "sb" "la" "oc" "oc" "sb"
> county = factor(countyVector)
> county
[1] la sb la oc oc sb
Levels: la oc sb
```

A factor is also printed without quotes

Creating a Factor

```
> attributes(county)
$levels
[1] "la" "oc" "sb"
$class
[1] "factor"
```

Subsetting a Factor

```
> county1 = county[1:3]
> county1
[1] la sb la
Levels: la oc sb
```

```
> nlevels(county1)
[1] 3
```

Subsetting a Factor

- To eliminate the “oc” level, …

```
> county1 = factor(county[1:3])
> county1
[1] la sb la
Levels: la sb
```

```
> county[1:3, drop = T]
[1] la sb la
Levels: la sb
```

The Order of the Factor

- Internally, the factor is stored as a set of codes

```
> print.default(county)
[1] 1 3 1 2 2 3
```

la stored as 1 | oc stored as 2 | sb stored as 3

Alphabetically

```
> mode(county)
[1] "numeric"
```

The Order of the Factor

- Some statistical functions give the 1st level a special status
- To specify the levels explicitly, …

```
> county2 = factor(countyVector, levels = c("oc", "sb",
"la"))
> county2
[1] la sb la oc oc sb
Levels: oc sb la
```

Numeric Factors

❖Creating a factor based on numerical values

```
> numFactor = factor(seq(1, 10, by = 2))
> numFactor
[1] 1 3 5 7 9
Levels: 1 3 5 7 9
```

❖To convert numFactor to a numeric vector, ...

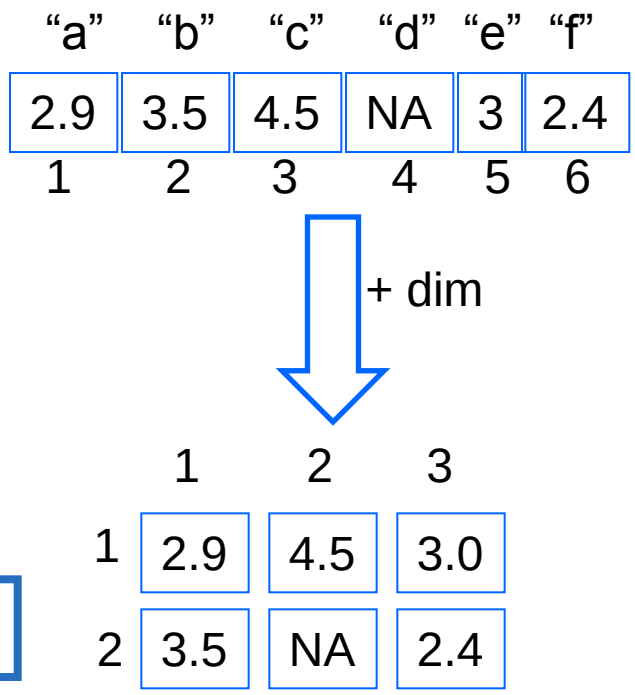
```
> as.numeric(numFactor)
[1] 1 2 3 4 5
```

```
> as.numeric(as.character(numFactor))
[1] 1 3 5 7 9
```

Matrices and Arrays
Creating a Matrix By Using the `dim` Attribute

❖ A vector + **`dim`** attribute → a matrix

```
> mydata
  a   b   c  d   e   f
2.9 3.5 4.5 NA 3.0 2.4
> dim(mydata) = c(2,3)
> mydata
     [,1] [,2] [,3]
[1,]  2.9  4.5  3.0
[2,]  3.5   NA  2.4
> names(mydata)
NULL
```



❖ The name attribute has been removed

❖ To store a matrix → a vector

```
> dim(mydata) = NULL
> mydata
[1] 2.9 3.5 4.5  NA 3.0 2.4
```

❖ The mode of a matrix is simply the mode of its element

Creating a Matrix By Using the `matrix` Function

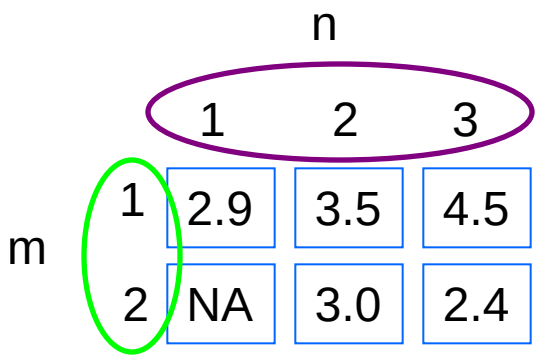
❖ Use the **`matrix`** function

```
> mydata1 = matrix(mydata, 2, 3)
> mydata1
     [,1] [,2] [,3]
[1,]  2.9  4.5  3.0
[2,]  3.5   NA  2.4
```

❖ To fill the matrix by row …

```
> mydata2 = matrix(mydata, 2, 3, byrow = TRUE)
> mydata2
     [,1] [,2] [,3]
[1,]  2.9  3.5  4.5
[2,]   NA  3.0  2.4
```

Accessing the Element of a Matrix



mat[m, n]

```
> mydata2[2,3]
[1] 2.4
> mydata2[2,]
[1]  NA 3.0 2.4
> mydata2[,3]
[1] 4.5 2.4
```

The dim, nrow, and ncol Functions

	1	2	3
1	2.9	3.5	4.5
2	NA	3.0	2.4

nrow = # of row

ncol = # of column

dim = dimension

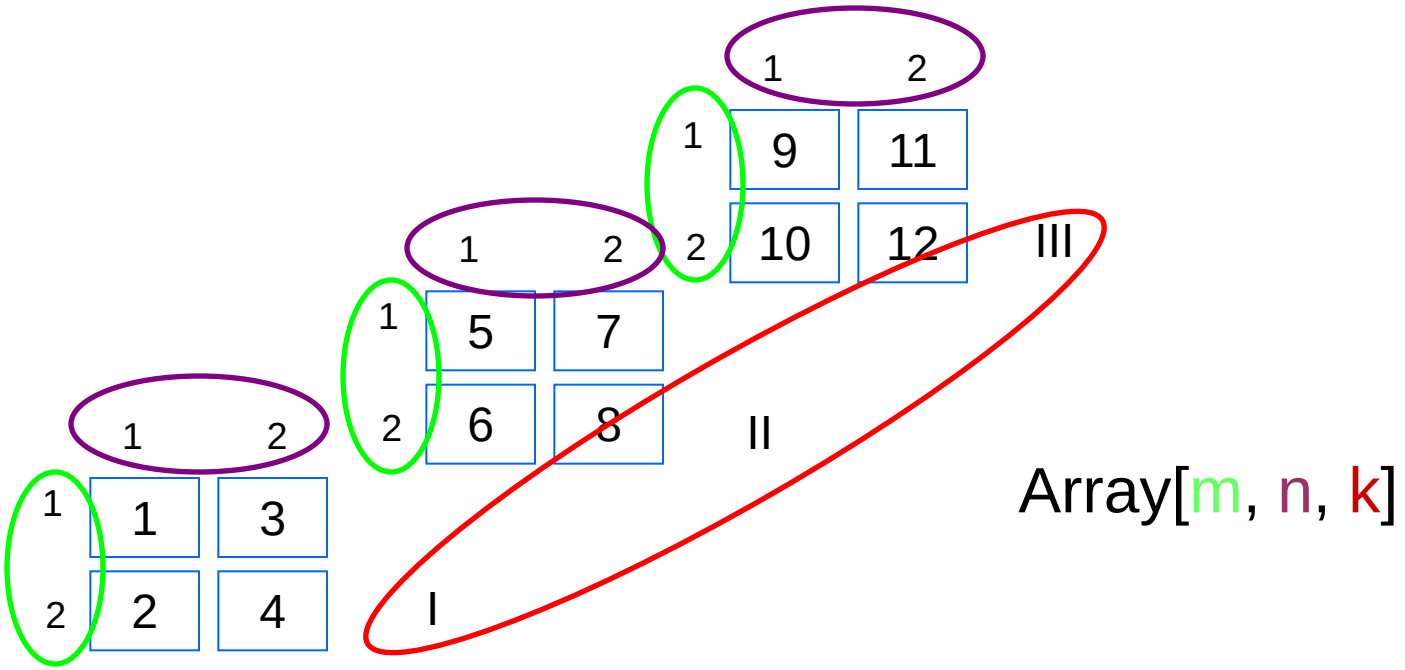
```
> dim(mydata2)
[1] 2 3
> nrow(mydata2)
[1] 2
> ncol(mydata2)
[1] 3
```

`dim(mydata.2)[1]`

`dim(mydata.2)[2]`

Arrays

- A vector is an array ↔ it has **`dim`** attribute/dimension vector
- The dim attribute: a vector of positive integer with length ≥1
- If the length (dim vector) = k, array is k-dimensional



Arrays

❖You can create an array by adding the dim attribute of a vector

```
> x = c(1:20, rep(NA,4))
> dim(x) = c(2,3,4)
> x
, , 1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,] 2 4 6
, , 2
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
, , 3
     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18
, , 4
     [,1] [,2] [,3]
[1,]   19   NA   NA
[2,]   20   NA   NA
```

Arrays

❖You can also create this array by using the array function

```
x = array(c(1:20, rep(NA,4)), c(2,3,4))
```

Arrays

❖ Each dimension can be given a set of names → **dimnames**

```
> dimnames(x) =  list(d1 = c("i", "ii"), d2 = c("I", "II", "III"), d3 =
+ letters[1:4])
> x
, , d3 = a

    d2
d1   I II III
  i  1  3   5
  ii 2  4   6

, , d3 = b

    d2
d1   I II III
  i  7  9  11
  ii 8 10  12
...
...
```

NULL

- ❖ NULL is used to indicate an object is absent
- ❖ NULL object should not be confused with a vector of zero length
- ❖ NULL has no type and no modifiable properties
- ❖ We cannot set attributes on NULL

```
> mode(NULL)
[1] "NULL"
> class(NULL)
[1] "NULL"
> length(NULL)
[1] 0
> identical(NULL, c())
[1] TRUE
```