# Lecture 3. Vectors

R and Data Visualization

BIG2006, Hanyang University, Fall 2022

# Scalars, Vectors, Arrays, and Matrices

R variables types are called $modes$.

All elements in a vector must have the same mode, which can be integer, numeric (floating-point number), character (string), logical (Boolean), complex, and so on.

**Note:** If you need your program code to check the mode of a variable x, you can query it by the call `typeof(x)`.

## Adding and Deleting Vector Elements

- Vectors are stored like arrays in C, so you cannot insert or delete element.
- The size of vector is determined at its creation.

  $\Longrightarrow$ To add or delete elements, reassign the vector!

```r
x <- c(1983, 5, 15, 2022)
x <- c(x[1:3],9999,x[4])
x
```

```
## [1] 1983    5   15 9999 2022
```

## Obtaining the Length of a Vector, `length()`

```r
x <- c(1,2,10,5000)
length(x)
```

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

- We often need to use `length()`.
- Now we want to have a function that determines the index of the first "1" value in the function's vector argument (assuming we are sure there is such a value).
- How do you write the code?

```r
first1 <- function(x){
  for (i in 1:length(x)){
    if (x[i]==1) break # break out of loop
  }
  return(i)
}
```

**Note:** Without the `length()` function, we would have needed to add a second argument to `first1()`, say naminig in n, to specify the length of x.

## Matrices and Arrays as Vectors

- Arrays and matrices are actually vectors too.

```
m <- matrix(1:4,nrow=2,ncol=2,byrow=TRUE)
m
```

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

```
m + 10:13 # (1,3,2,4) + (10,11,12,13) = (11,14,14,17)
```

```
##      [,1] [,2]
## [1,]   11   14
## [2,]   14   17
```

# Declarations

Compiled languages requires that you *declare* variables.

C example:

```
int x;
int y[3];
```

In R, you do not declare variables.

```
z <- 3
y <- vector(length=2)
y[1] <- 5
y[2] <- 12 # or y <- c(5,12)
```

# Recycling

When applying an operation to two vectors that requires them to be the same length, R automatically recycles or repeats, the shorter one.

```
> c(6,0,9,20,22) + c(1,2,4)
[1]  7  2 13 21 24
Warning message:
In c(6, 0, 9, 20, 22) + c(1, 2, 4) :
  longer object length is not a multiple of shorter
object length
```

# Common Vector Operations

We will cover artithmetic and logical operations, vector indexing, and some useful ways to create vectors.

## Vector Arithmetic and Logical Operations

- Scalars are actually one-elements vectors.
- $+$, $-$, $*$, $/$, and $\%\%$ operations will be applied element-wise.

```r
2+3
```

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

```r
"+"(2,3)
```

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

```r
x <- c(1,5,9)
x + c(5,-5,0)
```

```
## [1] 6 0 9
```

```r
x * c(5,0,-1)
```

```
## [1]  5  0 -9
```

```r
x / c(5,4,-1)
```

```
## [1]  0.20  1.25 -9.00
```

```r
x %% c(5,4,-1)
```

```
## [1] 1 1 0
```

## Vector Indexing

- Form a subvector by picking elements of the given vector for specific indices

```r
y <- c(1.2, 3.9, 0.4, 0.12)
y[c(1,3)] # extract elements 1 and 3 of y
```

```
## [1] 1.2 0.4
```

```r
v <- 2:3
y[v]
```

```
## [1] 3.9 0.4
```

- Negative subscript: exclude the given elements in the output

```r
z <- c(5,112,500)
z[-1] # exclude element 1
```

```
## [1] 112 500
```

```r
z[1:length(z)-1] # 1:(length(z)-1) ?
```

```
## [1]   5 112
```

```r
z[-length(z)]
```

```
## [1]   5 112
```

## Generating Useful Vectors with the : Operator

- The colon operator : produces a vector consisting of a range of numbers.

```
c(5:8, "||", 5:1)
```

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

```
i <- 3
c(1:i-1, "||", 1:(i-1))
```

```
## [1] "0"  "1"  "2"  "||" "1"  "2"
```

## Generating Vector Sequences with seq()

- seq() is a generalization of : and generates a sequence in arithmetic progression.

```
seq(from=-10,to=10,by=2)
```

```
##  [1] -10  -8  -6  -4  -2   0   2   4   6   8  10
```

```
seq(from=0.1,to=2,length=5)
```

```
## [1] 0.100 0.575 1.050 1.525 2.000
```

```r
x <- c(9,15,2022)
seq(x)
```

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

```r
x <- NULL
x
```

```
## NULL
```

```r
seq(x)
```

```
## integer(0)
```

**Note:** `seq(x)` gives us the same results as `1 : length(x)` if x is not empty, but it correctly evaluates to NULL if x is empty.

## Repeating Vector Constants with rep()

- rep() allows us to put the same constant into long vectors.

```
x <- rep(7,5)
x
```

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

```
rep(c(5,15,2019),3)
```

```
## [1]    5   15 2019    5   15 2019    5   15 2019
```

```
rep(c(3:1,2))
```

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

# Using all() and any()

The `any()` and `all()` functions are handy shortcuts.

They report whether any or all of their arugments are TRUE.

```r
x <- 1:15
c(any(x>8), any(x>20))
```

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

```r
c(all(x>8), all(x>0))
```

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

# Vectorized Operations

Suppose we have a function `f()` that we wish to apply to all elements of a vector x.

In many cases, we can accomplish this by simply calling `f()` on x itself.

This can really simplify our code and give us a dramatic performance increase of hundredfold or more.

One of the most effective ways to achieve speed in R code is to use operations that are *vectorized*, meaning that a function applied to a vector is actually applied individually to each element.

## Vector In, Vector Out

- Vectorized operations ($+$, $*$, $>$) enable a potential speedup.

```r
u <- c(5,2,8)
v <- c(-1,3,9)
u > v
```

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

```r
w <- function(x) return(x+1)
w(u)
```

```
## [1] 6 3 9
```

- Even the transcendental functions, square roots, logs, trig functions, and so on, are vectorized.

```
sqrt(1:5)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

```
y <- c(1.2,3.9,0.4)
z <- round(y)
z
```

```
## [1] 1 4 0
```

```r
f <- function(x,c) return((x+c)^2)
f(1:3,0)
```

```
## [1] 1 4 9
```

```r
f(1:3,2)
```

```
## [1]  9 16 25
```

## Vector In, Matrix Out

- Consider the function itself is vector-valued, as `z12()` is here:

```
z12 <- function(z) return(c(z,z^2))
x <- 1:4
z12(x)
```

```
## [1]  1  2  3  4  1  4  9 16
```

⇒ More natural to have these arranged as an 9-by-2 matrix, which we can do with the `matrix` function.

```r
matrix(z12(x),ncol=2)
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    4
## [3,]    3    9
## [4,]    4   16
```

- We can streamline things using `sapply()` (simplify apply).
- The call `sapply(x, f)` applies the function `f()` to each element of `x` and then converts the result to a matrix.

```
z12 <- function(z) return(c(z,z^2))
sapply(1:9,z12)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    2    3    4    5    6    7    8    9
## [2,]    1    4    9   16   25   36   49   64   81
```

# NA and NULL Values

In statistical data sets, we often encounter missing data, which we represent in R with the value NA.

NULL, on the other hand, represents that the value in question simply doesn't exist, rather than being existent but unknown.

# Using NA

```r
x <- c(15, NA, 99, 2019)
x
```

```
## [1]   15   NA   99 2019
```

```r
c(mean(x), mean(x, na.rm=T)) # instruction to skip NAs
```

```
## [1]  NA 711
```

```r
x <- c(15, NULL, 99, 2019)
mean(x)
```

```
## [1] 711
```

## Using NULL

```r
# built up a vector of the even numbers in 1:20
z <- NULL
for (i in 1:20) if (i %% 2 == 0) z <- c(z,i)
z
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```r
z <- NA
for (i in 1:20) if (i %% 2 == 0) z <- c(z,i)
z
```

```
##  [1] NA  2  4  6  8 10 12 14 16 18 20
```

**Note:** NULL values really are counted as nonexistent and NULL is a special R object with no mode.

# Filtering

Another feature reflecting the functional programming nature of R is *filtering*.

This allows us to extract a vector's elements that satisfy certain conditions.

## Generating Filtering Indices

- Example 1

```
z <- c(5, 2, -3, 8)
w <- z[z*z > 8]
w
```

```
## [1]  5 -3  8
```

**Note:** Evaluation of the expression $\mathtt{z} * \mathtt{z} > 8$ gives us a vector of Boolean values!

- Example 2

```r
z <- c(5, 2, -3, 8)
y <- c(1, 2, 30, 5)
y[z*z > 8]
```

```
## [1]  1 30  5
```

- Example 3 (involving assignment)

```r
x <- c(1, 3, 8, 2, 20)
x[x > 3] <- 0
x
```

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

# Filtering with the subset() Function

- The difference between `subset()` and ordinary filtering lies in the manner in which NA values are handled.

```r
x <- c(6, 1:3, NA, 12)
x
```

```
## [1]  6  1  2  3 NA 12
```

```r
x[x > 5]
```

```
## [1]  6 NA 12
```

```r
subset(x, x > 5)
```

```
## [1]  6 12
```

## The Selection Function `which()`

- `which()` can extract the positions of elements in the vector at which the condition occurs.

```
z <- c(5, 2, -3, 8)
which(z*z < 8)
```

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

**Note:** $\mathtt{z} * \mathtt{z} < 8$ is evaluated to $(\mathtt{FALSE}, \mathtt{TRUE}, \mathtt{FALSE}, \mathtt{FALSE})$.

# A Vectorized if-then-else: The ifelse() Function

```
x <- 1:10
y <- ifelse(x %% 2 == 0, 5, 12) # %% is the mod operator
y
```

```
##  [1] 12  5 12  5 12  5 12  5 12  5
```

Here, we wish to produce a vector in which there is a 5 wherever x is even or a 12 wherever x is odd.

```
x <- c(5,2,9,12)
y <- ifelse(x > 6, 2*x, 3*x) # %% is the mod operator
y
```

```
## [1] 15  6 18 24
```

# Testing Vector Equality

Suppose we want to test whether two vectors are equal.

```
x <- 1:3
y <- c(1,3,4)
x == y
```

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

**Note:** In fact, $==$ is a vectorized function. The expression `x` $==$ `y` applies the function $==()$ to the elements of $x$ and $y$, yielding a vector of Boolean values.

```r
"=="(3,2)
```

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

```r
i <- 2
"=="(i,2)
```

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

```r
all(x == y)
```

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

```r
identical(x,y)
```

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

# Vector Element Names

The elements of a vector can optionally be given names.

```r
x <- c(5,15,2019)
names(x)
```

```
## NULL
```

```r
names(x) <- c("day","month","year")
names(x)
```

```
## [1] "day"   "month" "year"
```

```r
x
```

```
##   day month  year
##     5    15  2019
```

```r
x["month"]
```

```
## month
##    15
```

```r
names(x) <- NULL
x
```

```
## [1]    5   15 2019
```

# More on c()

If the arguments you pass to c() are of differing modes, they will be reduced to a type that is the lowest common denominator, as follows:

```
c(5,2,"hanyang")
```

```
## [1] "5"       "2"       "hanyang"
```

R consider the list mode to be of lower precedence in mixed expressions.

```
c(5,2,list(math=3,stat=7))
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] 2
##
## $math
## [1] 3
##
## $stat
## [1] 7
```

Another point to keep in mind is that c() has a flattening effect for vectors.

```r
c(5,2,c(1.8,3.5))
```

```
## [1] 5.0 2.0 1.8 3.5
```

# Reference


- Matloff, N. The Art of R Programming: A Tour of Statistical Software Design. No Starch Press. Chapter 2.