Basic data structures

Statistical Computing, STA3005

Jan 8, 2024

Why statisticians learn to program

- **Independence**: otherwise, you rely on someone else giving you exactly the right tool
- **Honesty**: otherwise, you end up distorting your problem to match the tools you have
- **Clarity**: often, turning your ideas into something a machine can do refines your thinking
- **Fun**: these were the best of times (the worst of times)

How this class will work

- Instructor: Fangda Song
- TA: Jiasheng Li
- Prerequisite: Some statistics knowledge assumed, but no real programming knowledge assumed
- Focus is R programming language
- Lecture: Twice a week, each 90 mins
- Tutorial: Weekly and starting from the third week, each 45 mins
- Assignment: Biweekly, complied R markdown files needed
- Final project: Develop your own R package, a group of at most three members, posted around Week 8
- Class will be cumulative, so keep up with the material and assignments!

- Assessment scheme:
  - Assignments: 50%
  - Project: 30%
  - Final: 20%

R, RStudio, R Markdown

- R is a programming language for statistical computing
- RStudio is an integrated development environment for R programming
- R Markdown is a markup language for combining R code with text

All three are free, and all three will be used extensively in this course

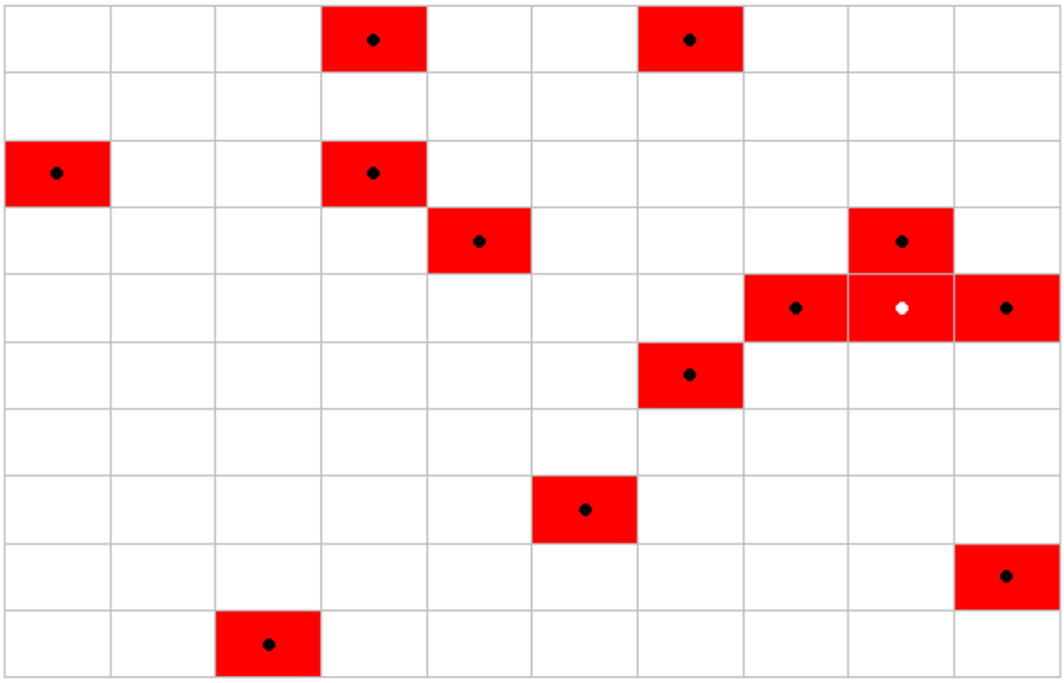
Cool example: Catching an intruder

There is a $n \times n$ matrix, and the probability of a black point emerging in a cell of the matrix is p. Moreover, we know that one of these black points is an intruder. As time passes, the intruder moves one step in any of four directions or stay steady at the previous position, while normal points randomly appear in each cell with probability p. How long can we identify the intruder for different values of n and p?

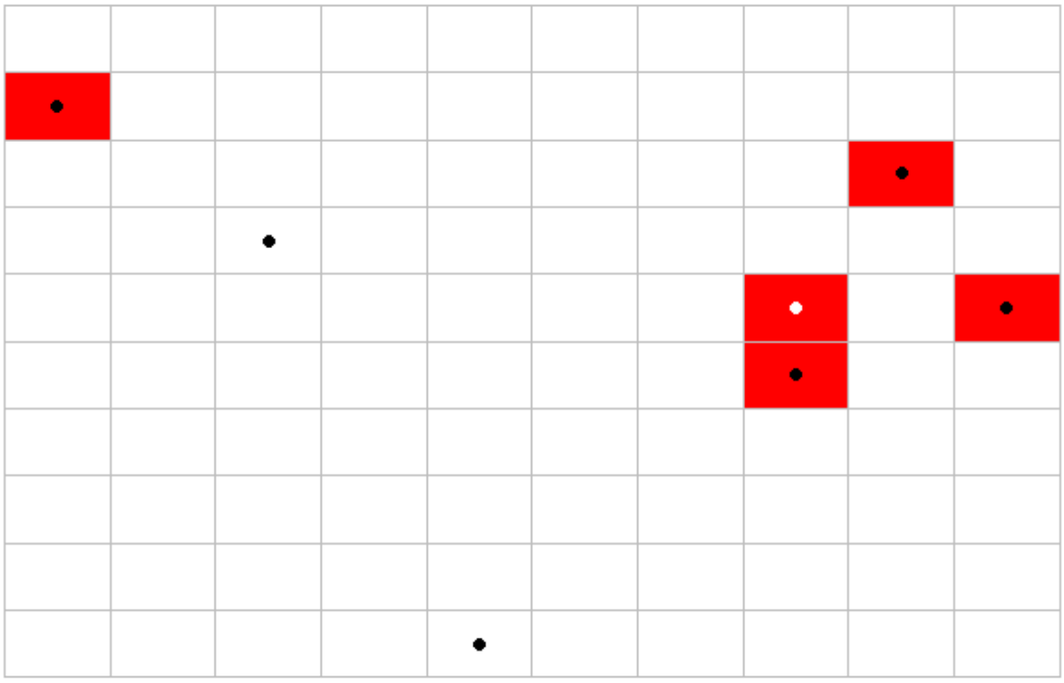
- Guess:
  - Large n $\longrightarrow$ longer
  - Large p $\longrightarrow$ longer
- Task:
  - Gamer to do simulation: A sequence of random matrices
  - Player to identify the intruder: Filter candidates

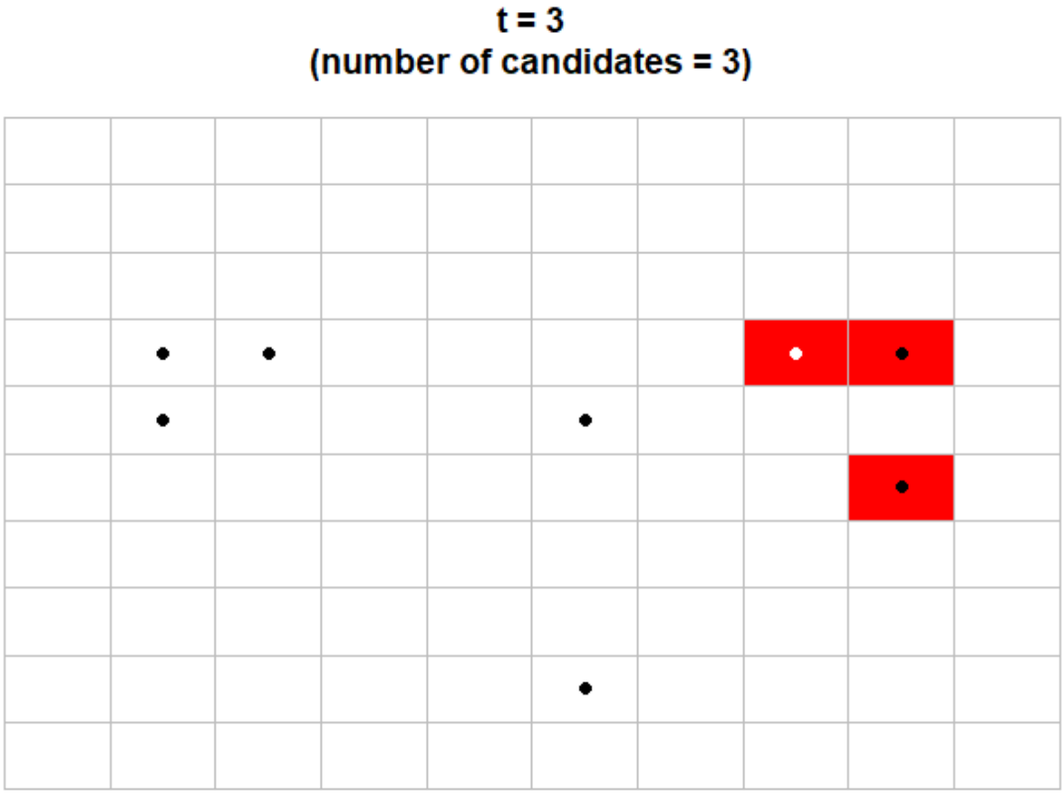
```
source("intruder.R")
set.seed(1234) # Fix the initial state of random number gene
intruder.sim(n=10, p=0.1)
```

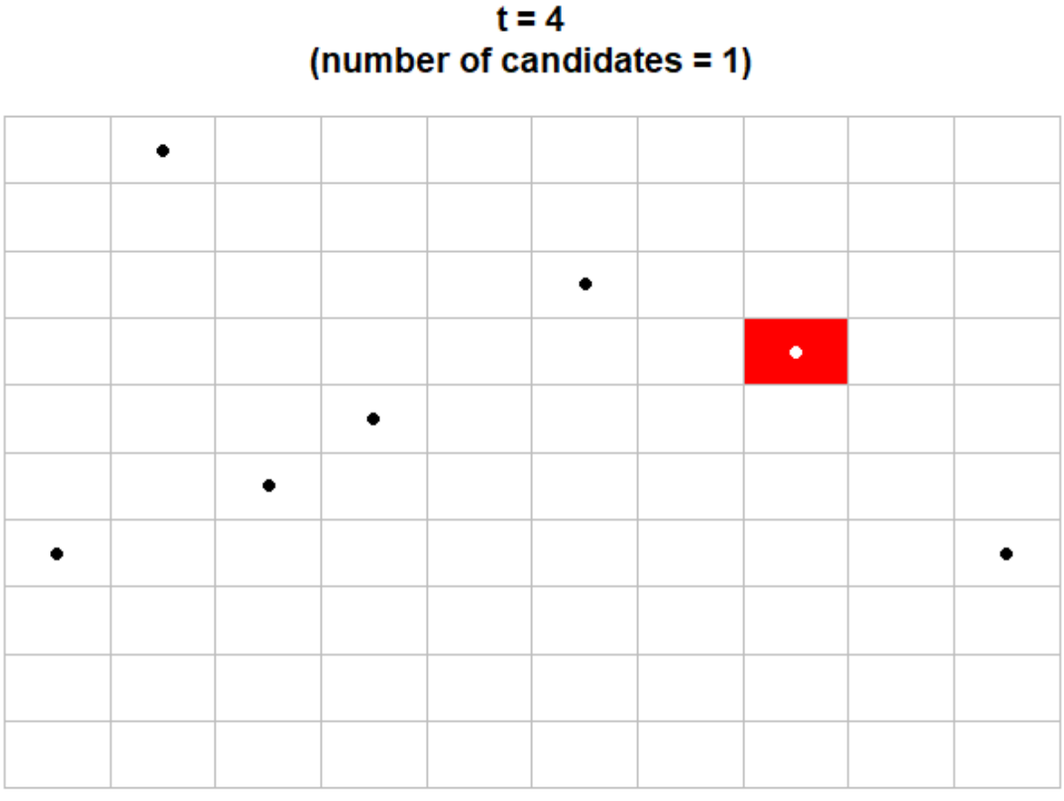
t = 1
(number of candidates = 13)



t = 2
(number of candidates = 5)








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

Part I: Data types, operators, variables

This class in a nutshell: functional programming

1.组成成分有哪些

Two basic types of things/objects: **data** and **functions**

- **Data**: things like 7, "seven", 7.000, and $\begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$
- **Functions**: things like `log`, `+` (takes two arguments), `<` (two), `%%` (two), and `mean` (one)

2.function 是什么

> A function is a machine which turns input objects, or **arguments**, into an output object, or a **return value** (possibly with side effects), according to a definite rule

- Programming is writing functions to transform inputs into outputs
- Good programming ensures the transformation is done easily and correctly
- Machines are made out of machines; functions are made out of functions, like $f(a, b) = a^2 + b^2$

3. 怎样写出好的function

> The trick to good programming is to take a big transformation and **break it down** into smaller ones, and then break those down, until you come to tasks which are easy (using built-in functions)

Before functions, data

1.有哪些数据类型

At base level, all data can represented in binary format, by **bits** (i.e., TRUE/FALSE, YES/NO, 1/0). Basic data types:

- **Booleans**: Direct binary values: `TRUE` or `FALSE` in R
- **Integers**: whole numbers (positive, negative or zero), represented by a fixed-length block of bits (R没有明确区分 integer 和 double)
- **Characters**: fixed-length blocks of bits, with special coding; **strings**: sequences of characters
- **Floating point numbers**: an integer times a positive integer to the power of an integer, as in 3×10^6 or 1×3^{-1}
- **Missing or ill-defined values**: `NA`, `NaN`, etc.

* NA: not available
NaN: not a (valid) number (或其他类型)

Operators

1.运算符的类型

- **Unary**: take just one argument. E.g., `-` for arithmetic negation, `!` for Boolean negation

- **Binary**: take two arguments. E.g., `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division). Also, `%/%` (integer division), `%%` (remainder), and `^` (power)

整数除法 取模

```
-7
```

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

```
!TRUE
```

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

```
7 + 5
```

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

```
7 - 5
```

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

```
7 * 5
```

```
## [1] 35
```

```
7 ^ 5
```

```
## [1] 16807
```

```
7 / 5
```

```
## [1] 1.4
```

```
7 %/% 5
```

(整数除法)

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

```
7 %% 5
```

(取模)

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

Comparison operators

These are also binary operators; they take two objects, and return a Boolean

```
7 > 5
```

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

```
7 < 5
```

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

```
7 >= 7
```

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

```
7 <= 5
```

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

```
7 == 5
```

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

```
7 != 5
```

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

Warning: `==` is a comparison operator, `=` is not!

Logical operators

These basic ones are `&` (and) and `|` (or)

```
(5 > 7) & (6 * 7 == 42)
```

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

```
(5 > 7) | (6 * 7 == 42)
```

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

```
(5 > 7) | (6 * 7 == 42) & (0 != 0)
```

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

```
(5 > 7) | (6 * 7 == 42) & (0 != 0) | (9 - 8 >= 0)
```

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

Note: The double forms `&&` and `||` are different! We'll see them later

```r
#短路原则
#若单独执行a会报错，因为a还没被定义
TRUE || a
FALSE && a
#但是上面两个运算不会报错，因为不需要执行a就可以确定结果
```

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

```r
x = ((-2:2) >= 0)
print(x)
y = ((-2:2) <= 0)
print(y)

x & y
x && y
x | y
x || y
```

```
[1] FALSE FALSE  TRUE  TRUE  TRUE
[1]  TRUE  TRUE  TRUE FALSE FALSE
[1] FALSE FALSE  TRUE FALSE FALSE
Warning: 'length(x) = 5 > 1' in coercion to 'logical(1)'[1] FALSE
[1] TRUE TRUE TRUE TRUE TRUE
Warning: 'length(x) = 5 > 1' in coercion to 'logical(1)'Warning:
'length(x) = 5 > 1' in coercion to 'logical(1)'[1] TRUE
```

① double forms 有短路原则，single forms 没有
② 对于 vector inputs，single forms 会返回一个 vector，double forms 只会比较第一个元素

More types

- The `typeof()` function returns the data type

- `is.foo()` functions return Booleans for whether the argument is of type *foo*
- `as.foo()` (tries to) "cast" its argument to type *foo*, to translate it sensibly into such a value

```
typeof(7)
```

```
## [1] "double"
```

```
is.numeric(7)
```

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

```
is.na(7)
```

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

```
is.na(7/0)
```

= Inf 注: is.nan(7/0) ## FALSE

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

(Inf is available & is a number)

```
is.na(0/0)
```

注: is.nan(0/0) ## TRUE

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

(0/0 is not available & is not a number)

```
is.character(7)
```

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

```
is.character("7")
```

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

```
is.character("seven")
```

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

```
is.na("seven")
```

注：is.nan("seven") # FALSE

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

```
as.character(5/6)
```

```
## [1] "0.833333333333333"
```

```
as.numeric(as.character(5/6))
```

```
## [1] 0.8333333
```

(和5/6有误差，仅保留了特定位数)

```
typeof(as.character(5/6))
```

```
## [1] "character"
```

```
typeof(as.numeric(as.character(5/6)))
```

```
## [1] "double"
```

```
6 * as.numeric(as.character(5/6))
```

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

```
5/6 == as.numeric(as.character(5/6))
```

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

```
5/6 - as.numeric(as.character(5/6))
```

```
## [1] 3.330669e-16
```

Data can have names

1.内置变量名

We can give names to data objects; these give us **variables**. Some variables are built-in:

```
pi
```

```
## [1] 3.141593
```

2.变量运算

Variables can be arguments to functions or operators, just like constants:

```
pi * 10
```

```
## [1] 31.41593
```

```
cos(pi)
```

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

3.用赋值运算符给变量赋值

We create variables with the **assignment operator**, `<-` or `=`

```
approx.pi = 22/7
approx.pi
```

```
## [1] 3.142857
```

```
diameter = 10
approx.pi * diameter
```

```
## [1] 31.42857
```

4.用赋值运算符改变变量的值

The assignment operator also changes values:

```
circumference = approx.pi * diameter
circumference
```

```
## [1] 31.42857
```

```
circumference = 30
circumference
```

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

- The code you write will be made of variables, with descriptive names
- Easier to design, easier to debug, easier to improve, and easier for others to read
- Avoid “magic constants”; instead use named variables
- Named variables are a first step towards **abstraction**

R workspace

1.查看定义了哪些变量

What variables have you defined?

```
ls()
```

```
## [1] "approx.pi"       "check.neighbors" "circumference"
## [5] "find.candidates" "intruder.sim"    "plot.onetime"
```

2.删除单个变量

Getting rid of variables:

```
rm("circumference")
ls()
```

```
## [1] "approx.pi"       "check.neighbors" "diameter"
## [5] "intruder.sim"    "plot.onetime"    "random.step"
```

3.删除所有变量

```
rm(list=ls()) # Be warned! This erases everything
ls()
```

```
## character(0)
```

Part II: Data structures

First data structure: vectors

1.创建一个vector

- A **data structure** is a grouping of related data values into an object
- A **vector** is a sequence of values, all of the same type

```
x = c(7, 8, 10, 45)
x
```

```
## [1]  7  8 10 45
```

```
is.vector(x)
```

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

- The `c()` function returns a vector containing all its arguments in specified order
- `1:5` is shorthand for `c(1,2,3,4,5)`, and so on
- `x[1]` would be the first element, `x[4]` the fourth element, and `x[-4]` is a vector containing *all but* the fourth element

(除掉第4个后的所有元素)

2.创建一个空vector，随后进行赋值

`vector(length=n)` returns an empty vector of length *n*; helpful for filling things up later

```
weekly.hours = vector(length=5)
weekly.hours
```

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

```
typeof(weekly.hours)
```

```
## [1] "logical"
```

(initialize 成 logical type)

```
weekly.hours[5] = 8
weekly.hours
```

```
## [1] 0 0 0 0 8
```

(一旦有一个元素变为了 double，其他元素也会被转化为 double)

"coercion"

```
typeof(weekly.hours)
```

```
## [1] "double"
```

注："coercion" 的级别：

logical / boolean
numerical
character

Vector arithmetic

Arithmetic operator apply to vectors in a "component-wise" fashion

```
y = c(-7, -8, -10, -45)
x + y
```

[x = c(7, 8, 10, 45)]

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

```
x * y
```

```
## [1]   -49   -64  -100 -2025
```

Recycling

注：longer vector 的元素数必须是 shorter vector 的倍数

Recycling repeat elements in shorter vector when combined with a longer one [x=c(7,8,10,45)]

```
x + c(-7,-8)
```

等价于 x+ rep(c(-7,-8),2)

```
## [1]  0  0  3 37
```

```
x^c(1,0,-1,0.5)
```

```
## [1] 7.000000 1.000000 0.100000 6.708204
```

Single numbers are vectors of length 1 for purposes of recycling:

```
2 * x
```

```
## [1] 14 16 20 90
```

Can do componentwise comparisons with vectors:

```
x > 9
```

(会返回一个向量)

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

Logical operators also work elementwise:

```
(x > 9) & (x < 20)
```

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

To compare whole vectors, best to use `identical()` or `all.equal()`:

```
x == -y
```

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

```
identical(x, -y)
```

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

```
identical(c(0.5-0.3,0.3-0.1), c(0.3-0.1,0.5-0.3))
```

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

注：identical 检验两个 arguments 是否 exactly equal

```
all.equal(c(0.5-0.3,0.3-0.1), c(0.3-0.1,0.5-0.3))
```

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

注：all.equal 检验两个 arguments 是否 nearly equal

tolerance 默认为 $\sqrt{\text{machine number}} \approx 1.5\times10^{-8}$

注：all.equal(5/6, as.numerical(as.character(5/6))) ## TRUE

Note: these functions are slightly different; we'll see more later

注：== 会判断是否 exactly equal，因此，c(0.5-0.3, 0.3-0.1) == c(0.3-0.1, 0.5-0.3) 会返回 (False, False)

Functions on vectors

Many functions can take vectors as arguments:

- `mean()`, `median()`, `sd()`, `var()`, `max()`, `min()`, `length()`, and `sum()` return single numbers
- `sort()` returns a new vector
- `hist()` takes a vector of numbers and produces a histogram, a highly structured object, with the side effect of making a plot
- `ecdf()` similarly produces a cumulative-density-function object
- `summary()` gives a five-number summary of numerical vectors
- `any()` and `all()` are useful on Boolean vectors

min; 1st qu; median; 3rd.qu; max

Indexing vectors

```
x
```

```
## [1]  7  8 10 45
```

Vector of indices:

```
x[c(2,4)]
```

```
## [1]  8 45
```

Vector of negative indices:

```
x[c(-1,-3)]
```
（除掉第1,3个后的所有元素）

```
## [1]  8 45
```

Boolean vector:

```
x[x > 9]
```
注：x>9被称为on-the-fly boolean vector（不用额外创建变量）

```
## [1] 10 45
```

```
y[x > 9]
```

```
## [1] -10 -45
```

`which()` gives the indices of a Boolean vector that are `TRUE`:

```
places = which(x > 9)
places
```

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

```
y[places]
```

```
## [1] -10 -45
```

Named components

We can give names to elements/components of vectors, and index vectors accordingly

```
names(x) = c("v1","v2","v3","fred")
names(x)
```

```
## [1] "v1"   "v2"   "v3"   "fred"
```

```
x[c("fred","v1")]
```

```
## fred   v1
##   45    7
```

注：这不是一个矩阵，只是把 names 一并显示出来了

Note: here R is printing the labels, these are not additional components of `x`

`names()` returns another vector (of characters):

```
names(y) = names(x)
sort(names(x)) # sort in alphabetical order of the first cha
```

注：sort( ) 不会改变 argument 的顺序

```
## [1] "fred" "v1"   "v2"   "v3"
```

```
which(names(x) == "fred")
```

（注意不是返回 1）

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

Second data structure: arrays

An **array** is a multi-dimensional generalization of vectors

```
x = c(7, 8, 10, 45)
x.arr = array(x, dim=c(2,2))
x.arr
```

(默认为 column-major order)

注: p.arr = array(x, dim=2): [1] 1 2

```
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   45
```

q.arr = array(x, dim=c(2,1)):

[,1]
[1,] 1
[2,] 2

	is.vector()	is.array()	is.matrix()
p.arr	X	✓	X
q.arr	X	✓	✓

- `dim` says how many rows and columns; filled by columns
- Can have 3d arrays, 4d arrays, etc.; `dim` is vector of arbitrary length

(在matrix中才可选择 by columns / by rows)

Some properties of our array:

```
dim(x.arr)
```

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

```
is.vector(x.arr)
```

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

```
is.array(x.arr)
```

注: is.matrix(x.arr) ## TRUE

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

```
typeof(x.arr)
```

(返回 elements 的 type)

```
## [1] "double"
```

Indexing arrays

(使用一对index) (使用一个index)

Can access a 2d array either by pairs of indices or by the underlying vector (column-major order):

```
x.arr[1,2]
```

```
## [1] 10
```

```
x.arr[3]
```

```
## [1] 10
```

Omitting an index means “all of it”:

```
x.arr[c(1,2),2]
```

```
## [1] 10 45
```

```
x.arr[,2]
```

（默认返回 vector）

```
## [1] 10 45
```

```
x.arr[,2,drop=FALSE]
```

```
##      [,1]
## [1,]   10
## [2,]   45
```

```
is.array(x.arr[,2])
```

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

```
is.array(x.arr[,2,drop=FALSE])
```

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

Note: the optional third argument `drop=FALSE` ensures that the result is still an array, not a vector

Functions on arrays

实际作用于 array 底层的 vector 上

Many functions applied to an array will just boil things down to the underlying vector:

```
which(x.arr > 9)
```

（先求出 booleans array，再转换为 vector，再执行 which）

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

This happens unless the function is set up to handle arrays specifically

And there are several functions/operators that *do* preserve array structure:

```
y = -x
y.arr = array(y, dim=c(2,2))
y.arr + x.arr
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
```

2d arrays: matrices

A **matrix** is a specialization of a 2d array

```
z.mat = matrix(c(40,1,60,3), nrow=2)
z.mat
```

```
##      [,1] [,2]
## [1,]   40   60
## [2,]    1    3
```

```
is.array(z.mat)
```

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

```
is.matrix(z.mat)
```

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

- Could also specify `ncol` for the number of columns
- To fill by rows, use `byrow=TRUE`
- Elementwise operations with the usual arithmetic and comparison operators (e.g., `z.mat/3`)

Matrix multiplication

Matrices have its own special multiplication operator, written `%*%`:

```
six.sevens = matrix(rep(7,6), ncol=3)
six.sevens
```

```
##      [,1] [,2] [,3]
## [1,]    7    7    7
## [2,]    7    7    7
```

```
z.mat %*% six.sevens # [2x2] * [2x3]
```

```
##      [,1] [,2] [,3]
## [1,]  700  700  700
## [2,]   28   28   28
```

改成t(six.sevens)则不会报错

```
six.sevens %*% z.mat # [2x3] * [2x2]
```

```
## Error in six.sevens %*% z.mat: non-conformable arguments
```

Can also multiply a matrix and a vector

Row/column manipulations

Row/column sums, or row/column means:

```
rowSums(z.mat)
```

```
## [1] 100   4
```

```
colSums(z.mat)
```

```
## [1] 41 63
```

```
rowMeans(z.mat)
```

```
## [1] 50  2
```

```
colMeans(z.mat)
```

```
## [1] 20.5 31.5
```

Matrix diagonal

The `diag()` function can be used to extract the diagonal entries of a matrix:

```
diag(z.mat)
```

```
## [1] 40  3
```

It can also be used to change the diagonal:

```
diag(z.mat) = c(35,4)
z.mat
```

```
##      [,1] [,2]
## [1,]   35   60
## [2,]    1    4
```

Creating a diagonal matrix

Finally, `diag()` can be used to create a diagonal matrix:

```
diag(c(3,4))
```

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

```
diag(2)
```

（默认为 identity matrix）

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Other matrix operators

Transpose:

```
t(z.mat)
```

```
##      [,1] [,2]
## [1,]   35    1
## [2,]   60    4
```

Determinant:

```
det(z.mat)
```

```
## [1] 80
```

Inverse:

```
solve(z.mat)
```

```
##         [,1]    [,2]
## [1,]  0.0500 -0.7500
## [2,] -0.0125  0.4375
```

```
z.mat %*% solve(z.mat)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Names in matrices

- We can name either rows or columns or both, with `rownames()` and `colnames()`
- These are just character vectors, and we use them just like we do `names()` for vectors
- Names help us understand what we're working with

Third data structure: lists

A **list** is sequence of values, but not necessarily all of the same type

```
my.dist = list("exponential", 7, FALSE)
my.dist
```

```
## [[1]]
## [1] "exponential"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] FALSE
```

Most of what you can do with vectors you can also do with lists

Accessing pieces of lists

- Can use `[ ]` as with vectors
- Or use `[[ ]]`, but only with a single index `[[ ]]` drops names and structures, `[ ]` does not

返回一个 list

返回 index 对应的元素

```
my.dist[2]
```

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

```
my.dist[[2]]
```

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

```
my.dist[[2]]^2
```

```
## [1] 49
```

Expanding and contracting lists

Add to lists with `c()` (also works with vectors):

```
my.dist = c(my.dist, 9)
my.dist
```

```
## [[1]]
## [1] "exponential"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] FALSE
##
```

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

Chop off the end of a list by setting the length to something smaller (also works with vectors):

```
length(my.dist)
```

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

```
length(my.dist) = 3
my.dist
```

```
## [[1]]
## [1] "exponential"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] FALSE
```

Keep all but exclude one piece of a list by negative indices (also works with vectors):

```
my.dist[-2]
```

```
## [[1]]
## [1] "exponential"
##
## [[2]]
## [1] FALSE
```

Names in lists

We can name some or all of the elements of a list:

```
names(my.dist) = c("family","mean","is.symmetric")
my.dist
```

```
## $family
## [1] "exponential"
##
## $mean
## [1] 7
##
## $is.symmetric
## [1] FALSE
```

```
my.dist[["family"]]
```

```
## [1] "exponential"
```

```
my.dist["family"]
```

```
## $family
## [1] "exponential"
```

Lists have a special shortcut way of using names, with `$`:

```
my.dist[["family"]]
```

```
## [1] "exponential"
```

```
my.dist$family
```

```
## [1] "exponential"
```

Creating a list with names:

```
another.dist = list(family="gaussian", mean=7, sd=1, is.symmetric = TRUE )
```

Adding named elements:

```
my.dist$was.estimated = FALSE
my.dist[["last.updated"]] = "2022-01-04"
```

Removing a named list element, by assigning it the value `NULL`:

```
my.dist$was.estimated = NULL
```

Key-value pairs

- Lists give us a natural way to store and look up data by *name*, rather than by *position*
- A really useful programming concept with many names: **key-value pairs**, i.e., **dictionaries**, or **associative arrays**
- If all our distributions have components named `family`, we can look that up by name, without caring where it is (in what position it lies) in the list

Data frames

- The classic data table, n rows for cases, p columns for variables
- Lots of the really-statistical parts of R presume data frames
- Not just a matrix because *columns can have different types*
- A special case of list. Each column of data frames is a list element, and the length of each element is the same
- Many matrix functions also work for data frames (e.g., `rowSums()`, `summary()`, `apply()`)

```
a.mat = matrix(c(35,8,10,4), nrow=2)
colnames(a.mat) = c("v1","v2")
a.mat
```

```
##      v1 v2
## [1,] 35 10
## [2,]  8  4
```

```
a.mat[,"v1"]
```

```
## [1] 35  8
```

```
a.mat$v1
```

(matrix 没有 shortcut way $)

```
## Error in a.mat$v1: $ operator is invalid for atomic vecto
```

```
a.df = data.frame(a.mat,logicals=c(TRUE,FALSE))
a.df
```

```
##   v1 v2 logicals
## 1 35 10     TRUE
## 2  8  4    FALSE
```

a.df[1] 是一个 list / dataframe

```
a.df[[1]]
```

(相当于检索 list 的第一个元素，返回第一列)

```
## [1] 35  8
```

(是一个 vector)

```
a.df$v1
```

```
## [1] 35  8
```

(是一个 vector)

```
a.df[,"v1"]
```

```
## [1] 35  8
```

(是一个 vector)

```
a.df[1,]
```

```
##   v1 v2 logicals
## 1 35 10     TRUE
```

(是一个 list & data.frame)

是 vector

```
colMeans(a.df)
```

注：此处发生了 coercion：TRUE→1，FALSE→0

```
##       v1       v2 logicals
##     21.5      7.0      0.5
```

Adding rows and columns

We can add rows or columns to an array or data frame with `rbind()` and `cbind()`, but be careful about forced type conversions (保证 df 每列的 type 相同)

```
rbind(a.df,list(v1=-3,v2=-5,logicals=TRUE))
```

```
##   v1 v2 logicals
## 1 35 10     TRUE
## 2  8  4    FALSE
## 3 -3 -5     TRUE
```

```
rbind(a.df,c(3,4,6))
```

```
##   v1 v2 logicals
## 1 35 10        1
## 2  8  4        0
## 3  3  4        6
```

被 cast 为 numerics

Much more on data frames a bit later in the course.

Structures of structures

So far, every list element has been a single data value. List elements can be other data structures, e.g., vectors and matrices, even other lists:

```
my.list = list(z.mat=z.mat, my.lucky.num=13, my.dist=my.dist
my.list
```

```
## $z.mat
##      [,1] [,2]
```

```
## [1,]   35   60
## [2,]    1    4
##
## $my.lucky.num
## [1] 13
##
## $my.dist
## $my.dist$family
## [1] "exponential"
##
## $my.dist$mean
## [1] 7
##
## $my.dist$is.symmetric
## [1] FALSE
##
## $my.dist$last.updated
## [1] "2022-01-04"
```

Summary

- We write programs by composing functions to manipulate data
- The basic data types let us represent Booleans, numbers, and characters
- Data structures let us group together related values
- Vectors let us group values of the same type
- Arrays add multi-dimensional structure to vectors
- Matrices act like you'd hope they would
- Lists let us combine different types of data
- Data frames are a special case of lists, allowing each column to have a different data type