



统计学导论：R语言实验04

# 数据存盘、流程控制和函数

主讲人：郑盼盼

# Outline

---

1. 数据存盘与读取
2. 流程控制
3. 函数
4. 实验3
5. 运算优先级和变量作用域\*



## 4.1 数据存盘

---

## 4.1 数据存盘与读取

---

假设我们在 R 中生成了—组员工数据，进行了某些分析后，想将结果**保存**起来，以便其他人可以使用或稍后加载继续分析。

```
# 定义一系列变量
names <- c("Alice", "Bob", "Charlie", "David")
age <- c(25, 30, 35, 40)
salary <- c(50000, 60000, 55000, 62000)
department <- c("HR", "IT", "Finance", "Marketing")

# 使用上面的向量生成一个数据框
employee_data <- data.frame(
  Name = names,
  Age = age,
  Salary = salary,
  Department = department
)
```



# 4.1 数据存盘与读取

假设我们在 R 中生成了—组员工数据，进行了某些分析后，想将结果保存起来，以便其他人可以使用或稍后加载继续分析。

```
# 定义一系列变量
names <- c("Alice", "Bob", "Charlie", "David")
age <- c(25, 30, 35, 40)
salary <- c(50000, 60000, 55000, 62000)
department <- c("HR", "IT", "Finance", "Marketing")

# 使用上面的向量生成一个数据框
employee_data <- data.frame(
  Name = names,
  Age = age,
  Salary = salary,
  Department = department
)
```

Environment | History | Connections | Tutorial

Import Dataset | 16 MiB | List

R | Global Environment

Data

employee\_data

4 obs. of 4 variables

Values

age	num [1:4] 25 30 35 40
department	chr [1:4] "HR" "IT" "Finance" "Marke...
names	chr [1:4] "Alice" "Bob" "Charlie" "D...
salary	num [1:4] 50000 60000 55000 62000

执行完后，我们可以在RStudio的右上角看到我们定义的五  
个变量：左侧为变量名，右侧为变量的具体内容。






## 4.1 数据存盘与读取

---

我们可以通过 `save()` 函数实现对于数据的存盘



```
save(age, employee_data, file="./test.RData")
```

`save(变量名..., file=文件存储路径)`

- 变量名 是我们希望存储的变量，可以包含多个，（若有多个变量名）变量名之间使用 `,` 隔开，例如，如上的代码中我们存储了 `age` 和 `employee_data` 两个变量。
- `file=文件存储路径` 用于指定数据存储位置，如 `file="./test.RData"` 指我们将两个变量存储在当前文件夹的 `test.RData` 文件下。

**注意：**文件存储路径为字符型，且其最好以 `.RData` 结尾，表示其是R语言所产生的数据。



## 4.1 数据存盘与读取

---

完成数据存盘后，我们先删除当前环境中所有的变量：



```
rm(list=ls())
```

- `rm(变量名)` 用于删除当前环境中的变量
  - `ls()` 用于列出当前环境中的所有变量名
  - `rm(list=ls())` 删除当前环境中的所有变量
-

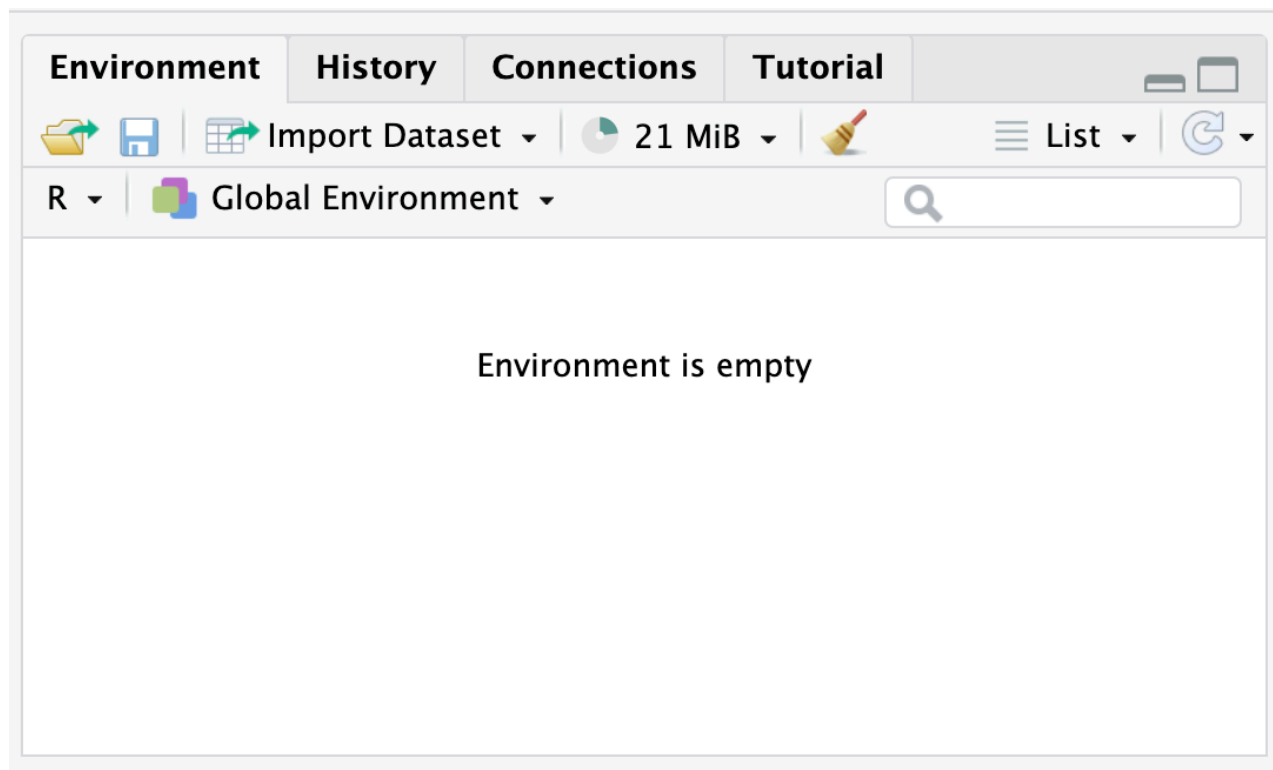


## 4.1 数据存盘与读取

完成数据存盘后，我们先删除当前环境中所有的变量：

```
rm(list=ls())
```

运行完后可以从RStudio的右上角看到，当前环境下没有任何变量






## 4.1 数据存盘与读取

---

然后通过 `load()` 函数进行数据的读取

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains a single line of R code: `load(file="./test.RData")`.

```
load(file="./test.RData")
```

`load(file=文件存储路径)`：导入文件存储路径对应的文件内的所有变量；如 `load(file="./test.RData")` 是导入当前目录下 `test.RData` 文件内所有的变量。



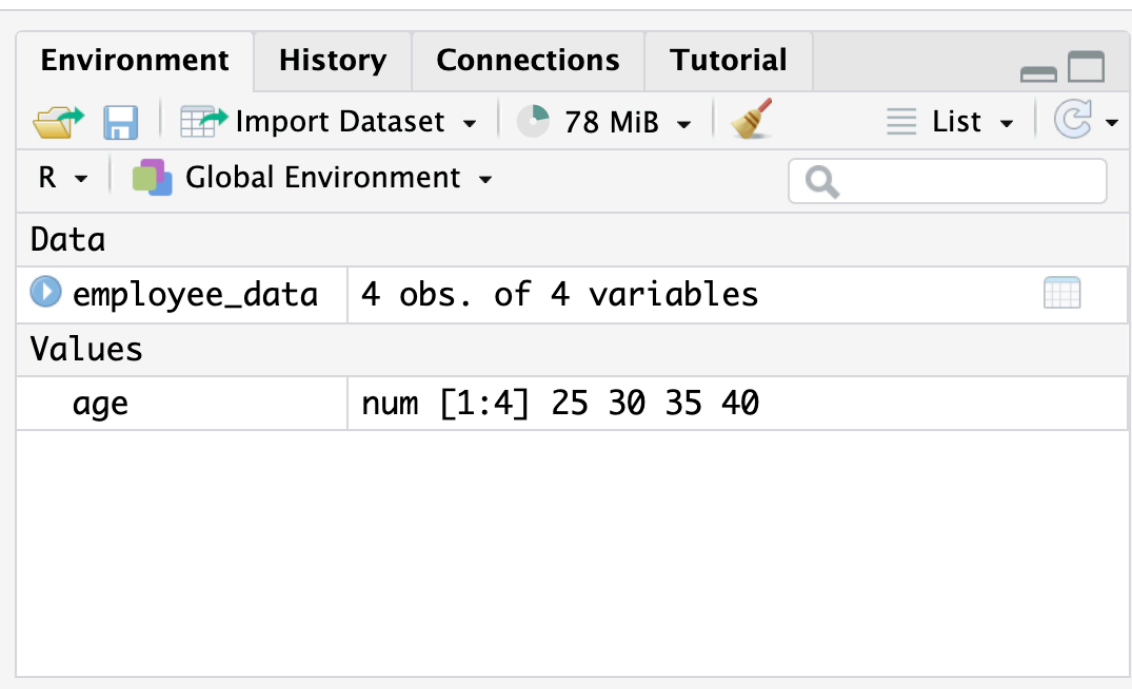


## 4.1 数据存盘与读取

然后通过 `load()` 函数进行数据的读取

```
load(file="./test.RData")
```

读取完成后，我们可以在RStudio右上角看到我们导入的两个变量 `age` 和 `employee_data`





## 4.2 流程控制

---

## 4.2 流程控制

---

1. 关系运算符
2. 逻辑运算符
3. 条件语句
4. 循环语句



## 4.2.1 关系运算符

---

关系运算符用于比较两个值或变量，结果会返回逻辑值 (TRUE 或 FALSE)。这些运算符可以用于数值、字符、逻辑值等的比较。

- **大小比较**：用于比较数值型（逻辑型也是一种特殊的数值型）之间的大小关系。
- **包含**：用于判断一个元素是否属于某一个集合。
- **变量类型判断**：用于判断一个变量是否属于某一数据类型



## 4.2.1 关系运算符

关系运算符用于比较两个值或变量，结果会返回逻辑值 (TRUE 或 FALSE)。这些运算符可以用于数值、字符、逻辑值等的比较。

- 大小比较

- `>`: 左侧是否大于右侧？若成立，返回 T，否则返回 F
- `<`: 左侧是否小于右侧？若成立，返回 T，否则返回 F
- `>=`: 左侧是否大于等于右侧？若成立，返回 T，否则返回 F
- `<=`: 左侧是否小于等于右侧？若成立，返回 T，否则返回 F
- `==`: 左侧是否等于右侧？若成立，返回 T，否则返回 F
- `!=`: 左侧是否不等于右侧？若成立，返回 T，否则返回 F

```
1 < 2      # T
1 > 2      # F
2 < 2      # F
2 <= 2     # T
2 >= 3     # F
2 != 2     # F
2 != 1     # T
2 == 2     # T

c(1,2) < 2 # c(T,F)
2 < c(1,3) # c(F,T)
c(1,2) < c(-1,3) # c(F,T)
c(1,2,3) < c(1,2) # error
```





## 4.2.1 关系运算符

---

关系运算符用于比较两个值或变量，结果会返回逻辑值(TRUE 或 FALSE)。这些运算符可以用于数值、字符、逻辑值等的比较。

- 大小比较
- 包含
  - %in% 判断左侧的变量是否是右侧向量或列表的元素

若左侧为一个向量或列表，则依次判断左侧向量或列表中的每个元素是否属于右侧，返回一个逻辑型向量。

```
1 %in% c(1,2,3,4)      # T
5 %in% c(1,2,3,4)      # F
c(1,5) %in% c(1,2,3,4) # 若左侧为一个
                        # 向量或列表，则依次判断其中的每个元素是否属于
                        # 右侧向量或列表，返回 c(T,T)
c(1,5) %in% list(1,2,3,4) # 右侧也可使
                        # 一个列表
```

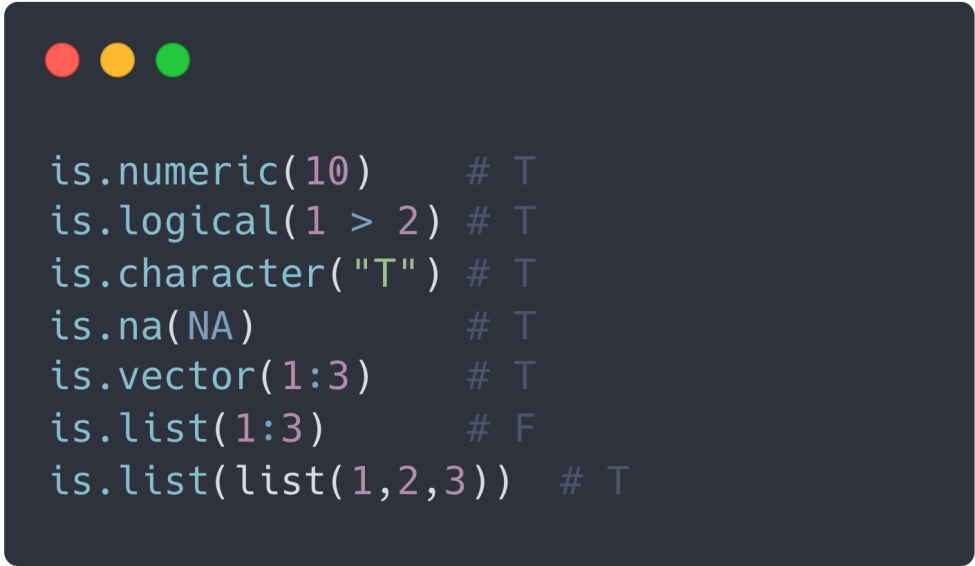


## 4.2.1 关系运算符

---

关系运算符用于比较两个值或变量，结果会返回逻辑值 (TRUE 或 FALSE)。这些运算符可以用于数值、字符、逻辑值等的比较。

- 大小比较
- 包含
- 变量类型判断
  - `is.logical()` 判断变量是否为逻辑型
  - `is.numeric()` 判断变量是否为数值型
  - `is.character()` 判断变量是否为字符型
  - `is.vector()` 判断变量是否为向量
  - `is.list()` 判断变量是否为列表
  - `is.na()` 判断变量是否是缺失值 NA



```
is.numeric(10)      # T
is.logical(1 > 2)    # T
is.character("T")    # T
is.na(NA)            # T
is.vector(1:3)       # T
is.list(1:3)         # F
is.list(list(1,2,3)) # T
```

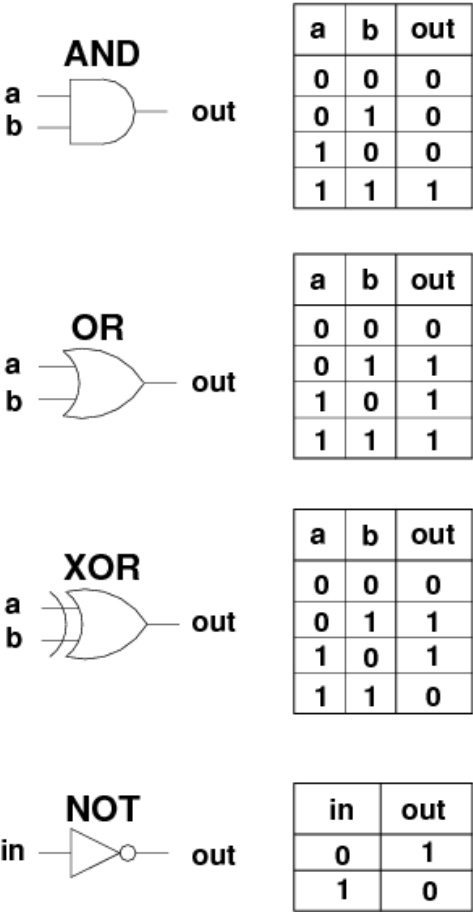


# 4.2.2 逻辑运算符

和数值型的加减乘除一样，逻辑型变量有其自己的运算方式：**与、或、非、异或**

- 与 (AND)：当两个条件都为 TRUE 时，返回 TRUE；否则返回 FALSE。
- 或 (OR)：当至少有一个条件为 TRUE 时，返回 TRUE；如果两个条件都为 FALSE，返回 FALSE。
- 非 (NOT)：用于反转逻辑值。如果条件为 TRUE，则返回 FALSE，反之亦然。
- 异或 (XOR)：两个条件中有且仅有一个为 TRUE 时，返回 TRUE；如果两个条件都为 TRUE 或都为 FALSE，返回 FALSE。

下图中 1 表示 TRUE，0 表示 FALSE





## 4.2.2 逻辑运算符

和数值型的加减乘除一样，逻辑型变量有其自己的运算方式：**与、或、非、异或**

- 与 (AND): `&`, `&&`
  - `&`: 两端既可以是单个变量也可以是一个向量了（或列表）
  - `&&`: 在 R4.3.0 之后，两端只能是单个逻辑值，不然会报错；在之前，若两端输入两个向量，其仅会在两个逻辑型向量的第一个元素之间进行与运算。



```
T & T      #T
T & F      #F
F & T      #F
F & F      #F
T && T     #T
F && T     #F
T && F     #F
F && F     #F

c(T,F) & T #T,F
c(T,F) & c(F,T) #F,F
c(T,F) && T #error
```





## 4.2.2 逻辑运算符

和数值型的加减乘除一样，逻辑型变量有其自己的运算方式：**与、或、非、异或**

- 与 (AND): `&`, `&&`
- 或 (OR): `|`, `||`
  - `|`: 两端既可以是单个变量也可以是一个向量了（或列表）
  - `||`: 在 R4.3.0 之后，两端只能是单个逻辑值，否则会报错；在之前，若两端输入两个向量，其仅会在两个逻辑型向量的第一个元素之间进行或运算。



```
T | T   #T
T | F   #F
F | T   #T
F | F   #F
T || T  #T
F || T  #T
T || F  #T
F || F  #F
```

```
c(T,F) | T #T,T
c(T,F) | c(F,T) #T,T
c(T,F) || T #error
```

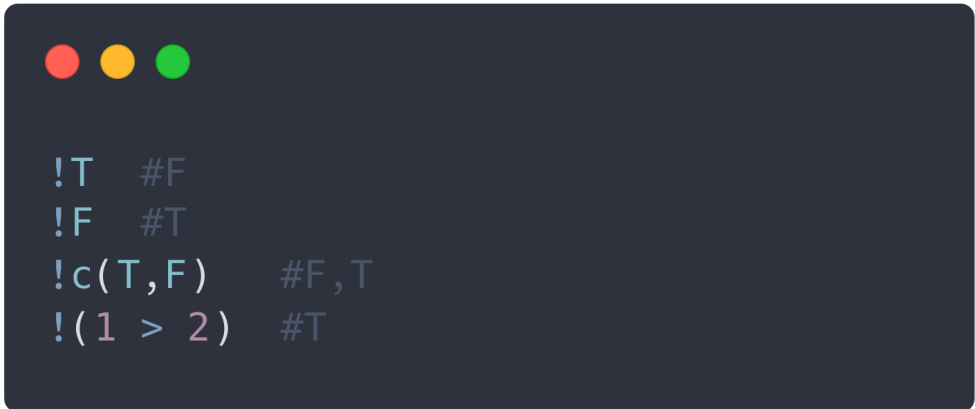


## 4.2.2 逻辑运算符

---

和数值型的加减乘除一样，逻辑型变量有其自己的运算方式：**与、或、非、异或**

- 与 (AND): `&`, `&&`
- 或 (OR): `|`, `||`
- 非 (NOT): `!`



```
!T    #F
!F    #T
!c(T,F)  #F,T
!(1 > 2)  #T
```

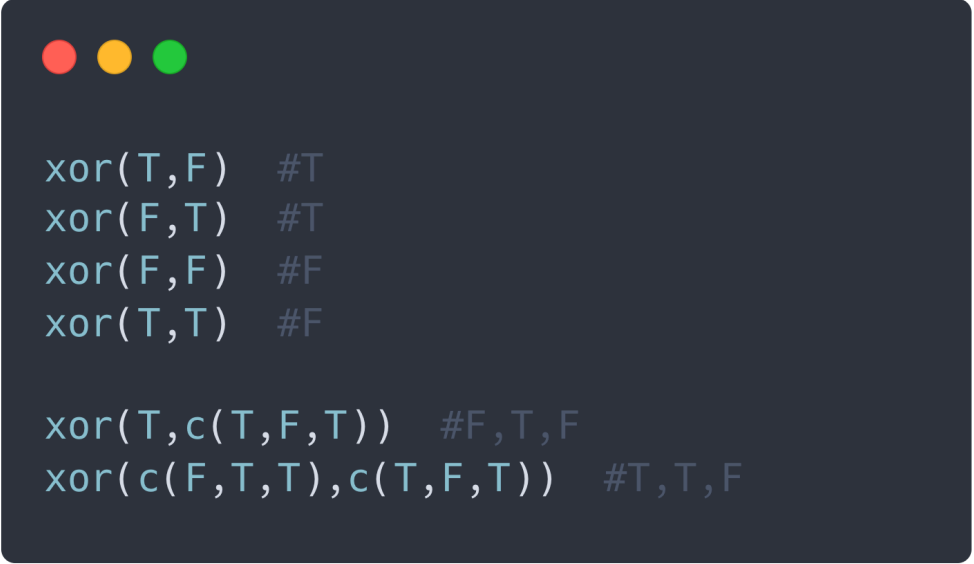


## 4.2.2 逻辑运算符

---

和数值型的加减乘除一样，逻辑型变量有其自己的运算方式：**与、或、非、异或**

- 与 (AND): `&`, `&&`
- 或 (OR): `|`, `||`
- 非 (NOT): `!`
- 异或 (XOR): `xor(a, b)`



```
xor(T,F)    #T
xor(F,T)    #T
xor(F,F)    #F
xor(T,T)    #F

xor(T,c(T,F,T))  #F,T,F
xor(c(F,T,T),c(T,F,T))  #T,T,F
```

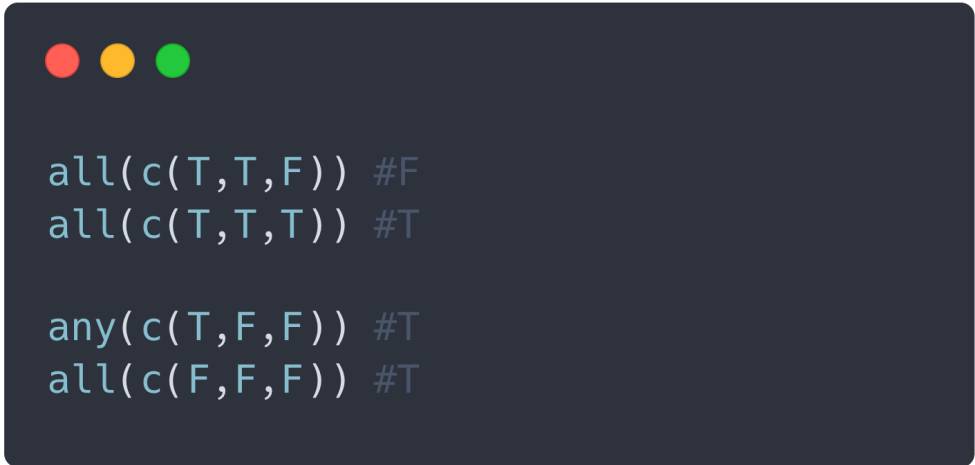


## 4.2.2 逻辑运算符

---

和数值型的加减乘除一样，逻辑型变量有其自己的运算方式：**与、或、非、异或**

- 与 (AND): `&`, `&&`
- 或 (OR): `|`, `||`
- 非 (NOT): `!`
- 异或 (XOR): `xor(a, b)`
- 多个逻辑值的运算:
  - `all(逻辑向量)` 判断一个逻辑型向量（或列表）是否全是 TRUE
  - `any(逻辑向量)` 判断一个逻辑型向量（或列表）是否含有 TRUE



```
all(c(T,T,F)) #F
all(c(T,T,T)) #T

any(c(T,F,F)) #T
all(c(F,F,F)) #F
```





## 4.2.2 逻辑运算符

Questions 使用R语言判断如下问题：

1.  $x \in [-1, 1)$
2.  $x \in (-\infty, -1) \cup [1, +\infty)$
3. 连续投掷 10 次骰子：
  - 判断是否 10 次的结果是否都是 6 点
  - 判断是否 10 次的结果中包含 6 点

```
x <- 10

# Q1
(x >= -1) & (x < 1) # 使用与运算，判断x是否同时满足大于等于-1和小于1的条件

# Q2
(x < -1) | (x >= 1) # 使用或运算，判断x是否小于-1或大于等于1
!((x >= -1) & (x < 1)) # Q1和Q2是对立事件，也可以使用非运算！

dices <- sample(1:6,10,T) # 模拟投掷10次骰子的结果
# Q3
## 判断十次结果是否都为6
all(dices==6)

## 判断十次结果是否包含6
any(dices==6)
6 %in% dices # 也可以通过判断6是否是dices里的元素来判断十次结果中是否包含6
```



## 4.2.3 条件语句

---

程序运行顺序： 从上至下



```
rm(list=ls()) # 清除当前环境中的所有变量
```

```
a <- c(1,2,3)
print(a)
a <- c(a,4)
print(a)
```

```
print(b) # 此前并未定义变量b，会报错，并终止运行
b <- 1
print(b)
```





## 4.2.3 条件语句

---

但很多时候，我们可能需要分情况处理，这时，我们就需要使用条件语句。

- if 语句

```
if (condition) {  
  expr  
}
```

- condition: 判断条件，为一个逻辑值
- expr: 若 condition 满足所执行的代码块

例如：可以用 if 语句判断一个数值为正数或负数

```
x <- 5  
  
if (x > 0) {  
  print("x 是正数")  
}  
  
if (x < 0) {  
  print("x 是负数")  
}
```



## 4.2.3 条件语句

---

但很多时候，我们可能需要分情况处理，这时，我们就需要使用条件语句。

- if 语句
- if else 语句

```
if (condition) {  
  expr1  
} else {  
  expr2  
}
```

- condition: 判断条件，为一个逻辑值
- expr1: 若判断条件满足，所执行的代码块
- expr2: 否则，执行的代码块

例如：使用 if else 语句，判断一个数值是否是正数，若是，返回"x是正数"，否则，返回 x不是正数

```
x <- -5  
  
if (x > 0) {  
  print("x 是正数")  
} else {  
  print("x 不是正数")  
}
```





## 4.2.3 条件语句

---

但很多时候，我们可能需要分情况处理，这时，我们就需要使用条件语句。

- if 语句
- if else 语句
- if else if 语句

例如：使用 if else 语句，判断一个数是正数，负数或0。

```
if (condition1) {  
  expr1  
} else if (condition2) {  
  expr2  
} else if (condition3) {  
  expr3  
} else {  
  expr4  
}
```

```
x <- -5  
  
if (x > 0) {  
  print("x 是正数")  
} else if (x < 0) {  
  print("x 是负数")  
} else if (x == 0) {  
  print("x是0")  
}
```



## 4.2.3 条件语句

---

但很多时候，我们可能需要分情况处理，这时，我们就需要使用条件语句。

- if 语句
- if else 语句
- if else if 语句
- 条件语句的嵌套与多重条件：我们可以在条件语句中嵌套条件语句
  - **嵌套**：先判断一个，再判断一个；
  - **多重条件**：通过逻辑运算符，判断多个条件是否同时成立（或多个条件中某个条件成立）

例如：通过条件语句的嵌套，判断一个数值是否是正整数；当然，也可以通过逻辑运算符来直接判断。

```
x <- 3

# 先判断 x 是否为正数
if (x > 0){
  # 再判断 x 能否被 1 整除
  if (x %% 1 == 0){
    print("x 为正整数")
  }
}

# 利用逻辑运算符：只有当x>0和x能被1整除时，x才是正整数
if (x > 0 & x %% 1 == 0){
  print("x 为正整数")
}
```



## 4.2.4 循环语句

---

反复执行某一段代码，直到特定的条件满足或遍历完某个序列。

- for 循环

```
for (variable in sequence) {  
    expr  
}
```

- variable 为循环变量，其仅在 {} 中有效
- sequence 为被遍历的向量或列表
- expr 称为**循环体**，是那些被不断重复执行的代码块

例如，依次打印向量 c(1,2,3,4,5) 中的每个元素

```
for (i in 1:5){  
    print(i) # 打印结果  
}
```



## 4.2.4 循环语句

反复执行某一段代码，直到特定的条件满足或遍历完某个序列。

- for 循环
- while 循环

```
while (condition) {  
  expr  
}
```

- condition 为每次循环开始时，检查的条件，为一个逻辑型，当条件为 TRUE 执行内部的循环体 expr，否则中止执行。
- expr 称为**循环体**，是那些被不断重复执行的代码块

例如，当  $x \leq 5$  时，打印  $x$  的数值，并让  $x = x+1$ ，一直到  $x > 5$  终止循环

```
x <- 1 # 设置 x 的初始值为 1  
# 条件设置为 x <= 5 时，进行循环，否则循环终止  
while (x <= 5) {  
  print(x) # 打印 x 的值  
  x <- x + 1 # 迭代 x，不然可能会造成死循环!  
}
```



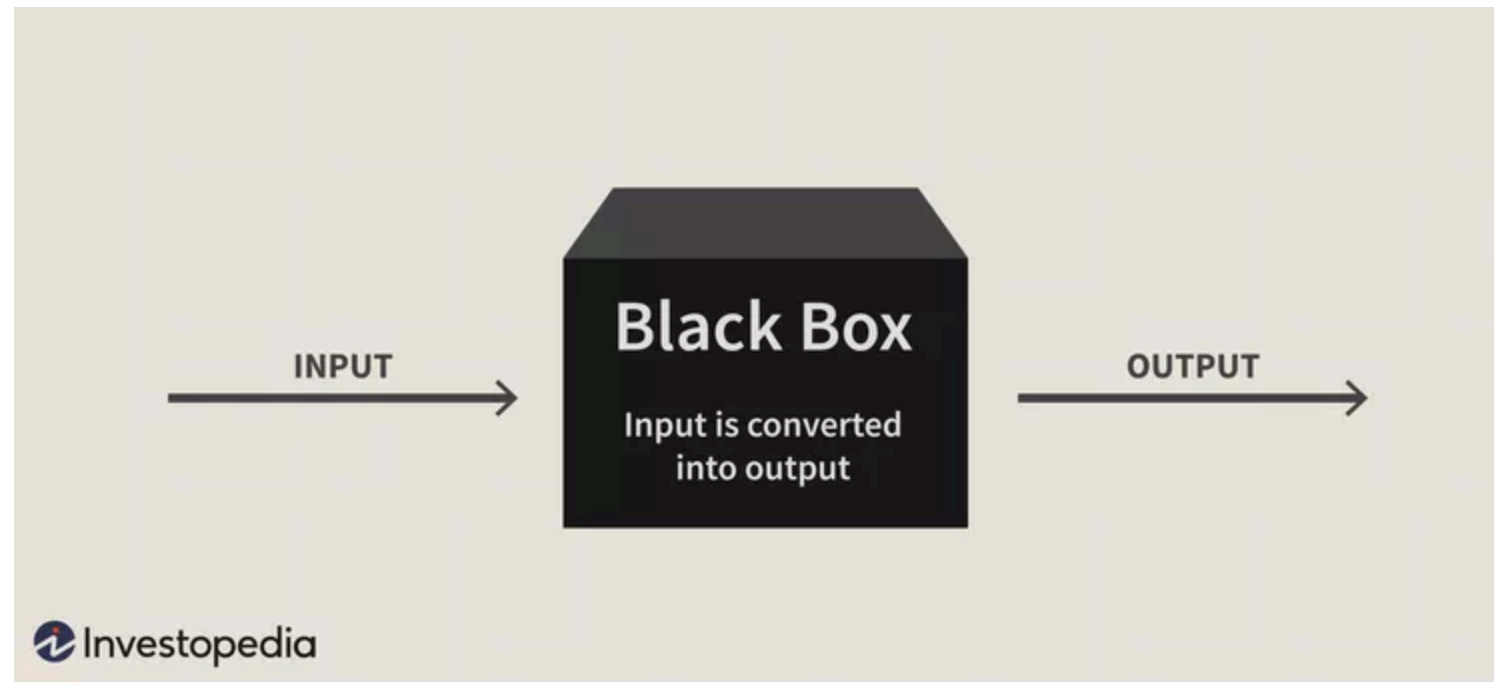


## 4.3 函数

---

## 4.3 函数

---

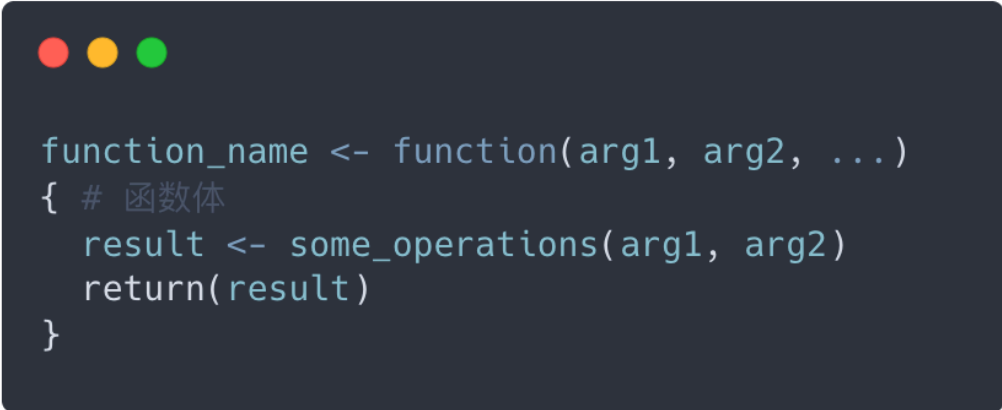




## 4.3 函数

---

- 之前我们所使用的函数都是R内置的函数，但很多时候，我们需要根据自身的需要自行编写函数。



```
function_name <- function(arg1, arg2, ...)  
{ # 函数体  
  result <- some_operations(arg1, arg2)  
  return(result)  
}
```

- `function_name` 为函数名，方便之后的调用。
  - `arg1`, `arg2` 为函数的输入，称为函数的参数 (parameters)，根据实际情况来确定函数的输入参数的个数。
  - `return(result)` 为函数的输出。
-



## 4.3 函数

---

- 例如，我们可以定义一个 add 函数，其会输入两个参数，对两个参数进行相加并返回。

```
add <- function(x,y) {  
  result <- x + y  
  return(result)  
}  
  
add(1,2) # 输出为3
```

- 函数名为 add
  - 输入为 x, y 两个参数
  - 返回值为 result
-





## 4.3 函数

---

- 尝试将我们上节课所使用的一个函数写成R语言的函数形式，函数名为 `f`

$$f(n) = (-1)^{\lfloor \log_2(n) \rfloor}$$




## 4.3 函数

---

- 尝试将我们上节课所使用的一个函数写成R语言的函数形式，函数名为 f

$$f(n) = (-1)^{\lfloor \log_2(n) \rfloor}$$



```
f <- function(x){  
  result <- (-1)^floor(log2(x))  
  return(result)  
}
```



## 4.3 函数

---

- 尝试将我们上节课所使用的一个函数写成R语言的函数形式，函数名为 f

$$f(n) = (-1)^{\lfloor \log_2(n) \rfloor}$$



```
f <- function(x){  
  result <- (-1)^floor(log2(x))  
  return(result)  
}
```

? Q

如果把输入变量 x 全部替换成 n 对于此函数何影响?



## 4.4 实验3

---

## 4.4 实验3

---

1. 从  $1 : 10$  中用取后放回的方法依次任意抽取 8 个数，计算事件  $A = \{1, 2, 3, 4, 5\}$  的频率。





## 4.4 实验3

---

1. 从  $1 : 10$  中用取后**放回**的方法依次任意抽取 8 个数，计算事件  $A = \{1, 2, 3, 4, 5\}$  的频率。
2. 将1.中的有放回的抽取方法改为**不放回**的抽取方法，计算事件  $A = \{1, 2, 3, 4, 5\}$  的频率



## 4.4 实验3

---

1. 从  $1 : 10$  中用取后**放回**的方法依次任意抽取 8 个数，计算事件  $A = \{1, 2, 3, 4, 5\}$  的频率。
2. 将1.中的有放回的抽取方法改为**不放回**的抽取方法，计算事件  $A = \{1, 2, 3, 4, 5\}$  的频率
3. 使用**循环语句**，分别将 1. 和 2. 重复 100 次，分别得到两个含有 100 个频率值的向量：  $\{x_1, x_2, \dots, x_{100}\}$ （对应1.） 和  $\{u_1, u_2, \dots, u_{100}\}$ （对应2.）



## 4.4 实验3

---

1. 从  $1 : 10$  中用取后放回的方法依次任意抽取 8 个数, 计算事件  $A = \{1, 2, 3, 4, 5\}$  的频率。
2. 将1.中的有放回的抽取方法改为不放回的抽取方法, 计算事件  $A = \{1, 2, 3, 4, 5\}$  的频率
3. 使用循环语句, 分别将 1. 和 2. 重复 100 次, 分别得到两个含有 100 个频率值的向量:  $\{x_1, x_2, \dots, x_{100}\}$  (对应1.) 和  $\{u_1, u_2, \dots, u_{100}\}$  (对应2.)
  - 用红色将 100 个点  $\{(x_1, 1), (x_2, 1), \dots, (x_{100}, 1)\}$  绘制在直角坐标系中
  - 用蓝色将 100 个点  $\{(u_1, 2), (u_2, 2), \dots, (u_{100}, 2)\}$  绘制在直角坐标系中



## 4.5 运算优先级和变量作用域\*

---



## 4.5 运算优先级和变量作用域\*

---

R语言中的运算优先级：从上到下，优先级逐渐递减：

1. 括号运算符：() 用于强制改变运算顺序，括号内的会优先运算
2. 函数调用：如 `sqrt()`, `log()`, `abs()`：函数的调用优先于其他运算
3. 指数运算符：`^` 或 `**`
4. 乘除和取余运算：`*`, `/`, `%%`
5. 加减法：`+`, `-`
6. 关系运算符：`>`, `<`, `>=`, `<=`, `==`, `!=`, `%in%`
7. 赋值运算符：`<-`, `=`



Tip

如果有复杂的表达式，建议使用括号来明确运算顺序，避免由于运算符优先级产生的错误。



## 4.5 运算优先级和变量作用域\*

---

R语言中变量的作用域：

1. **全局作用域**：在脚本的顶层环境（global environment）中定义的，通常是直接在 R 控制台或脚本的主代码段中定义的变量。全局变量可以在整个脚本中访问和修改。

```
x <- 10 # 全局变量

my_function <- function() {
  print(x) # 可以访问全局变量 x
}

my_function() # 输出 10
```



## 4.5 运算优先级和变量作用域\*

---

R语言中变量的作用域：

1. **全局作用域**：在脚本的顶层环境（global environment）中定义的，通常是直接在 R 控制台或脚本的主代码段中定义的变量。全局变量可以在整个脚本中访问和修改。
2. **局部作用域**：在函数内部定义的，这些变量只能在该函数内访问，函数外部无法访问这些变量。当函数执行完毕时，局部变量会被销毁，不会影响全局环境中的变量。

```
my_function <- function() {  
  y <- 20 # 局部变量 y  
  print(y) # 输出 20  
}  
  
my_function()  
print(y) # 错误：找不到对象 'y'
```



## 4.5 运算优先级和变量作用域\*

---

R语言中变量的作用域：

1. **全局作用域**：在脚本的顶层环境（global environment）中定义的，通常是直接在 R 控制台或脚本的主代码段中定义的变量。全局变量可以在整个脚本中访问和修改。
2. **局部作用域**：在函数内部定义的，这些变量只能在该函数内访问，函数外部无法访问这些变量。当函数执行完毕时，局部变量会被销毁，不会影响全局环境中的变量。
3. **词法作用域**：函数内的变量首先在局部作用域中查找，如果在局部环境中没有找到变量，则依次向外层环境查找，直到全局环境。

```
x <- 10 # 全局变量

my_function <- function() {
  x <- 20 # 局部变量，遮蔽全局变量
  inner_function <- function() {
    print(x) # 使用局部变量 x
  }
  inner_function()
}

my_function() # 输出 20
```





