Chapter 5
Introduction to R Functions

Arthur Li

R Functions
R Built-in Functions

❖Functions in R are themselves R objects

❖Many things in R are done using function calls

```
plot(height, weight)
```

❑Function name: **`plot`**

❑Actual argument: height, weight

R Built-in Functions

❖Functions in R are themselves R objects

❖Many things in R are done using function calls

```
plot(height, weight)
```

```
height → x
weight → y
```

Positional matching

Formal argument

```
> plot
function (x, y, ...)
{
    if (is.function(x) && is.null(attr(x, "class"))) {
        if (missing(y))
            y <- NULL
        hasylab <- function(...) !all(is.na(pmatch(names(list(...)),
            "ylab")))
        if (hasylab(...))
            plot.function(x, y, ...)
        else plot.function(x, y, ylab = paste(deparse(substitute(x)),
            "(x)"), ...)
    }
    else UseMethod("plot")
}
<environment: namespace:graphics>
```

R Built-in Functions

```
> formals(plot)

$x

$y

$...
```

R Built-in Functions

```
> plot
function (x, y, ...)
{
    if (is.function(x) && is.null(attr(x, "class"))) {
        if (missing(y))
            y <- NULL
        hasylab <- function(...) !all(is.na(pmatch(names(list(...)),
            "ylab")))
        if (hasylab(...))
            plot.function(x, y, ...)
        else plot.function(x, y, ylab = paste(deparse(substitute(x)),
            "(x)"), ...)
    }
    else UseMethod("plot")
}
<environment: namespace:graphics>
```

```
plot(x=height, y=weight)
```

```
plot(y=weight, x=height)
```

keyword matching

Creating Your Own Functions

```
fname = function (arg1 <, arg2, ...>) function.body
```

```
fname = function (arg1 <, arg2, ...>) {
     function.body
}
```

❑**`fname:`** is the name of the function

❑**`function`**: keyword

❑**`arg1, arg2,...:`** formal arguments

❑**`Function.body:`** usually a group of statements

Creating Your Own Functions

- A function call is called by giving its name with argument sequence in parentheses

```
fname(val1 <, y=val2, ... >)
```

Creating Your Own Functions

❑ Conduct a two-sample t-test

```
> set.seed(2)
> var1 = rnorm(50, 3, 2)
> var2 = rnorm(60, 5, 3)
> result = t.test(var1, var2)
> result
Welch Two Sample t-test
data: var1 and var2
t = -3.0751, df = 101.474, p-value = 0.002704
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-2.8246762 -0.6094406
sample estimates:
mean of x mean of y
3.138276 4.855334
> mode(result)
[1] "list"
```

Creating Your Own Functions

```
> str(result)
List of 9
 $ statistic  : Named num -3.08
  ..- attr(*, "names")= chr "t"
 $ parameter  : Named num 101
  ..- attr(*, "names")= chr "df"
 $ p.value    : num 0.0027
 $ conf.int   : atomic [1:2] -2.82 -0.61
  ..- attr(*, "conf.level")= num 0.95
 $ estimate   : Named num [1:2] 3.14 4.86
  ..- attr(*, "names")= chr [1:2] "mean of x" "mean of y"
 $ null.value : Named num 0
  ..- attr(*, "names")= chr "difference in means"
 $ alternative: chr "two.sided"
 $ method     : chr "Welch Two Sample t-test"
 $ data.name  : chr "var1 and var2"
 - attr(*, "class")= chr "htest"
```

```
> print(result$statistic)
t
-3.075058
```

Creating Your Own Functions

```
> str(result)
List of 9
 $ statistic  : Named num -3.08
  ..- attr(*, "names")= chr "t"
 $ parameter  : Named num 101
  ..- attr(*, "names")= chr "df"
 $ p.value    : num 0.0027
 $ conf.int   : atomic [1:2] -2.82 -0.61
  ..- attr(*, "conf.level")= num 0.95
 $ estimate   : Named num [1:2] 3.14 4.86
  ..- attr(*, "names")= chr [1:2] "mean of x" "mean of y"
 $ null.value : Named num 0
  ..- attr(*, "names")= chr "difference in means"
 $ alternative: chr "two.sided"
 $ method     : chr "Welch Two Sample t-test"
 $ data.name  : chr "var1 and var2"
 - attr(*, "class")= chr "htest"
```

```
> print(paste("t", round(result$statistic, 3), sep = "= "))
[1] "t= -3.075"
```

Creating Your Own Functions

```
> str(result)
List of 9
 $ statistic  : Named num -3.08
  ..- attr(*, "names")= chr "t"
 $ parameter  : Named num 101
  ..- attr(*, "names")= chr "df"
 $ p.value    : num 0.0027
 $ conf.int   : atomic [1:2] -2.82 -0.61
  ..- attr(*, "conf.level")= num 0.95
 $ estimate   : Named num [1:2] 3.14 4.86
  ..- attr(*, "names")= chr [1:2] "mean of x" "mean of y"
 $ null.value : Named num 0
  ..- attr(*, "names")= chr "difference in means"
 $ alternative: chr "two.sided"
 $ method     : chr "Welch Two Sample t-test"
 $ data.name  : chr "var1 and var2"
 - attr(*, "class")= chr "htest"
```

- Write a function that used to print t-statistics, degree of freedom and p value

Creating Your Own Functions

```
printT = function(tValue) {
    print(paste("t", round(tValue$statistic, 3), sep = "= "))
    print(paste("DF", round(tValue$parameter, 3), sep = "= "))
    print(paste("p", round(tValue$p.value, 3), sep = "= "))
}
```

```
> printT(result)
[1] "t= -3.075"
[1] "DF= 101.474"
[1] "p= 0.003"
```

Creating Your Own Functions

- Create another function that
  - performs the two-sample t-test
  - print the three statistics
  - return its p-value only

```
myTtest = function(x1, x2) {
    result = t.test(x1, x2)
    printT(result)
    return(result$p.value)
}
```

```
> getP = myTtest(var1, var2)
[1] "t= -3.075"
[1] "DF= 101.474"
[1] "p= 0.003"
> getP
[1] 0.002704407
```

Creating Your Own Functions

❖ If one of the arguments is empty, return NA

```
myTtest1 = function(x1, x2){
    if (length(x1) == 0 | length(x2) ==0) return (NA)
    result = t.test(x1, x2)
    printT(result)
    result$p.value
}
```

```
> getP1 = myTtest1(numeric(0), var2)
> getP1
[1] NA
```

Creating Your Own Functions

- If you want to return several values, you combine them into a list

```
> myTtest2
function(x1, x2){
    if (length(x1) == 0 | length(x2) ==0) return (NA)
    result = t.test(x1, x2)
    printT(result)
    list(method=result$method, t=result$statistic,
        df=result$parameter, p=result$p.value)
}
```

```
> result2 = myTtest2(var1, var2)
[1] "t= -3.075"
[1] "DF= 101.474"
[1] "p= 0.003"
```

Creating Your Own Functions

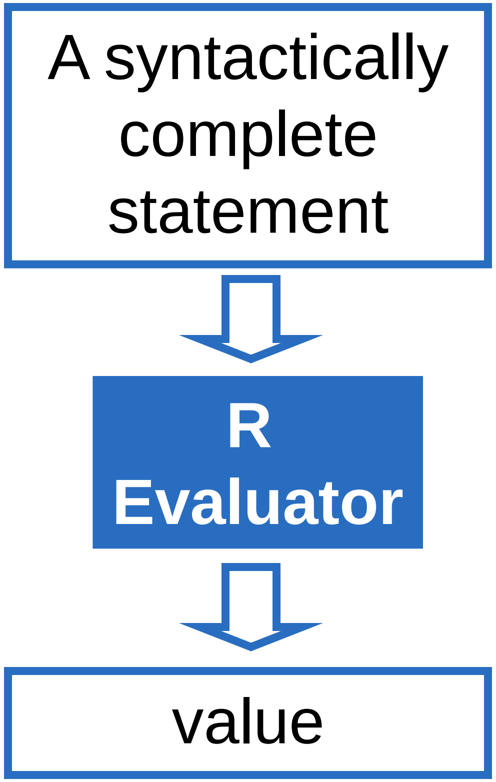
- If you want to return several values, you combine them into a list

```
> myTtest2
function(x1, x2){
    if (length(x1) == 0 | length(x2) ==0) return (NA)
    result = t.test(x1, x2)
    printT(result)
    list(method=result$method, t=result$statistic,
        df=result$parameter, p=result$p.value)
}
```

```
> result2
$method
[1] "Welch Two Sample t-test"
$t
t
-3.075058
$df
df
101.4736
$p
[1] 0.002704407
```

Conditional Execution

Single Statements



A syntactically complete statement

R Evaluator

value

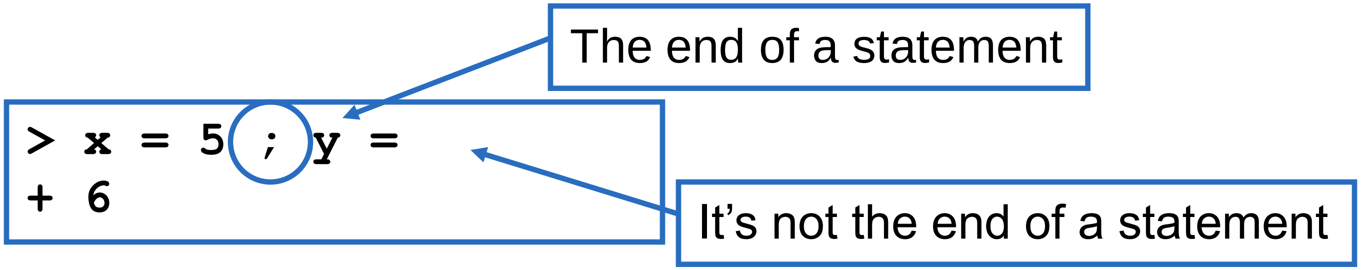
```
> 3 * 2 + 5
[1] 11
```

Single Statements

- ❖Statements can be separated by
  - ❑a semicolon
  - ❑a new line

❖If the current statement is not syntactically complete, new lines are simply ignored by the evaluator

❖If the session is interactive, the prompt changes from > to +



The end of a statement

```
> x = 5 ; y =
+ 6
```

It’s not the end of a statement

Blocks

- ❖ Statements can be grouped together by **{** and **}** → block

- ❖ Blocks will only be evaluated until a new line is entered after **}**

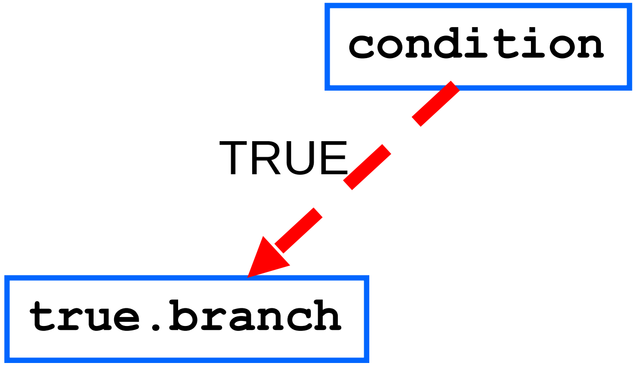
```
> {
+    x <- 0
+    x + 5
+ }
[1] 5
```

End of a statement

The `if` Statement

- The if statement has the following form

```
if (condition) true.branch else false.branch
```



condition

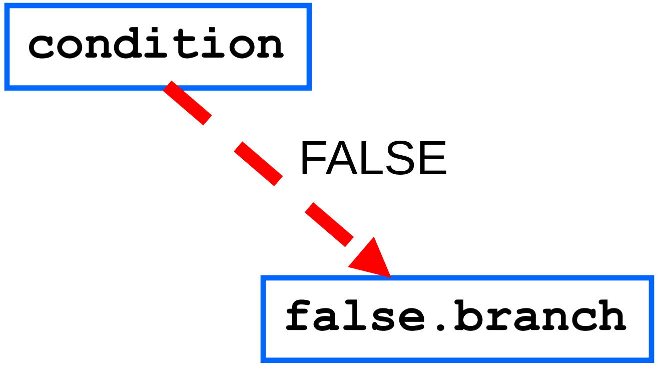
TRUE

true.branch

The `if` Statement

- The if statement has the following form

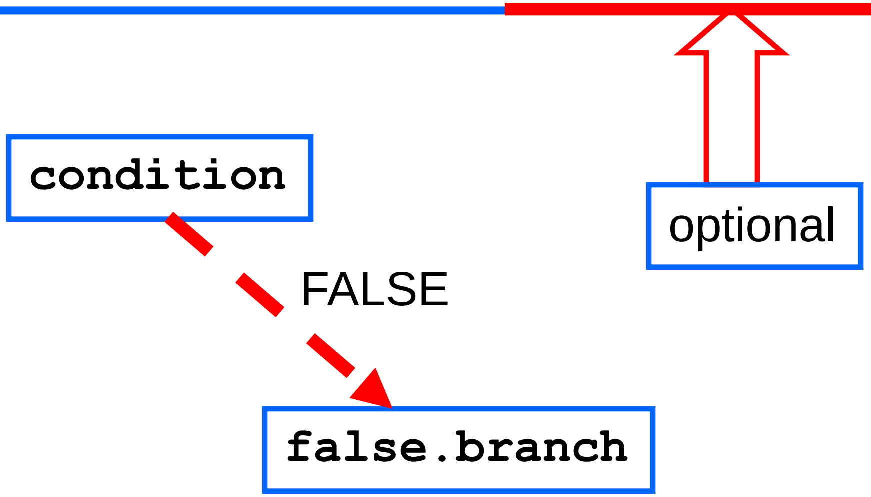
```
if (condition) true.branch else false.branch
```



The `if` Statement

❖ The if statement has the following form

```
if (condition) true.branch else NULL
```



condition

FALSE

false.branch

optional

The `if` Statement

- The if statement has the following form

```
if (condition) true.branch else false.branch
```

If it is a vector, only the first component is used

The `if` Statement

```
> x = rnorm(10)
> x
 [1]  1.83549568 -1.51726496  1.36570069 -0.04439632  1.06512840
 [6] -0.96186290 -1.83487466  0.48290005 -2.84671068 -0.42084838
> if (mean(x) > median(x)) print ("Mean > Median") else print   +
("Mean < Median")
[1] "Mean < Median"
```

The `if` Statement

❖ The **`if`** statement can be extended to several lines…

```
if (condition) {
    true.branch.1
    true.branch.2
    ...
} else {
    false.branch.1
    false.branch.2
    ...
}
```

The `if` Statement

- ❖ When the **`if`** statement is not in a block …

```
if (condition) else {
     false.branch.1
     false.branch.2
     ...
}
```

`else` must appear on the same line of the **`if`** statement

The `if` Statement

- When the **`if`** statement is in a block …

```
if (condition) {
    true.branch.1
    true.branch.2
    ...
} else {
    false.branch.1
    false.branch.2
    ...
}
```

`else` must appear on the same line of the closing bracket **`}`**

The `if` Statement

❖ If **`if`** statement is in a function → **`else`** can be placed in a new line

❖ Multiple cases: use the **`if ... else if...`** structure

```
if (condition) {
  ...
    ...
} else if {
  ...
} else if {
  ...
} else{
  ...
}
```

The `if` Statement

- You can assign the value of **`if/else`** statements to a variable

- The following two statements are equivalent

```
> if (any (x <= 0)) y = log(10+x) else y = log(x)
> y
 [1] 2.471103 2.138033 2.430600 2.298136 2.403799 2.201453
 [7] 2.099872 2.349745 1.967572 2.259589
> y = if (any (x <= 0)) log(10+x) else log(x)
> y
 [1] 2.471103 2.138033 2.430600 2.298136 2.403799 2.201453
 [7] 2.099872 2.349745 1.967572 2.259589
```

The Difference Between &, | and &&, ||

❖ Two additional logical operators, && and ||, are useful with the if statement

❖ **&**, **&&:** logical AND; **|**, **||**: logical OR

❖ **&** and **|** perform element-wise comparisons, while **&&** and **||** do not

❖ **&&** and **||** evaluate left to right examining only the first element of each vector

❖ With **&&**, the RH expression is only evaluated if the LH one is true, and with **||**, only if it is false

The Difference Between &, | and &&, ||

- Situations when either `&` or `&&` generates the same results:

```
> a = 3
> b = 3
> a == 3 & b == 3
[1] TRUE
> a == 3 && b == 3
[1] TRUE
```

- Both `a == 3` and `b == 3` return TRUE, which is a logical vector with length equaling 1

The Difference Between &, | and &&, ||

❖ Situations when using **`&`** - the goal of the calculation is for the element-wise comparison

```
> x = c(T, F, T)
> y = c(T, T, T)
> x & y
[1] TRUE FALSE TRUE
> x && y
[1] TRUE
```

```
> y1 = c(F, T, T)
> x && y1
[1] FALSE
```

The Difference Between &, | and &&, ||

- Situations when using &&:

- Suppose that you would like to test matrix y to see
  - if the `nrow(y) > 1`, and
  - if it does, you would like to check if `y[2, 1] == 2`

```
> y = matrix(1:2, 1)
> y
    [,1] [,2]
[1,]   1    2
> nrow(y) > 1 && y[2, 1] == 2
[1] FALSE
```

```
> nrow(y) > 1 & y[2, 1] == 2
Error: subscript out of bounds
```

Calculate the Square Root and Log of a Vector

```
sqrtAndLog = function(x){
    if (is.numeric(x) && min(x) > 0){
        x.sqrt <- sqrt(x)
        x.log <- log(x)
    } else stop ("x must be numeric and all components positive")
    return (list(x.sqrt, x.log))
}
```

```
> sqrtAndLog(c(2,4,3))
[[1]]
[1] 1.414214 2.000000 1.732051

[[2]]
[1] 0.6931472 1.3862944 1.0986123
> sqrtAndLog(c(2,4,-33))
Error in sqrtAndLog(c(2, 4, -33)) :
  x must be numeric and all components positive
```

Calculate the Central Tendency

❖ Example: calculate the central tendency with choices of arithmetic average, harmonic mean, or median

```
central = function (y, measure){
    if (measure == "mean") return (mean(y))
    else if (measure == "harmonic") return (1/mean(1/y))
    else if (measure == "median") return (median(y))
    else stop ("Your measure is not supported")
}
```

```
> z <- rnorm(10, mean=2, sd=3)
> central (z, "mean")
[1] 1.560157
> central (z, "harmonic")
[1] -1.634218
> central (z, "median")
[1] 0.73387
```

Test the Equality of Variance For a Two-sample T-test

- Write a function, *myTtest*, to perform a two-sample t-test
  - ❑ Test the equality of variance
  - ❑ If the variance are equal, set the **`var.equal = T`**
  - ❑ Print the t-statistics, degrees of freedom, and p-value
  - ❑ Return the result from the t-test

Test the Equality of Variance For a Two-sample T-test

```
myTtest = function(x, y) {
    vt.p = var.test(x, y)$p.value
    if (vt.p > 0.05) {
        print("Variance are equal")
        result = t.test(x, y, var.equal = T)
    } else {
        print("Variance are not equal")
        result = t.test(x, y)
    }
    print(paste("t: ", round(result$statistic, 2), sep = ""))
    print(paste("DF: ", result$parameter, sep = ""))
    print(paste("p: ", round(result$p.value, 2), sep = ""))
    return(result)
}
```

Test the Equality of Variance For a Two-sample T-test

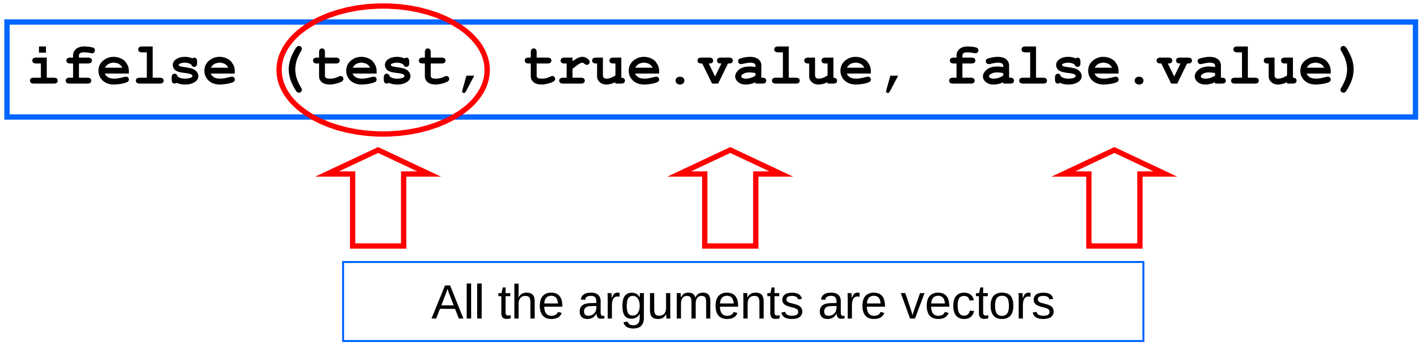
```
> set.seed(3)
> a = rnorm(10, 2, 5)
> b = rnorm(8, 3, 2)
> c = rnorm(9, 5, 5)
> r1 = myTtest(a, b)
[1] "Variance are not equal"
[1] "t: -0.22"
[1] "DF: 10.1891228068484"
[1] "p: 0.83"
> r1
Welch Two Sample t-test
data: x and y
t = -0.2203, df = 10.189, p-value = 0.83
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-3.454233 2.831165
sample estimates:
mean of x mean of y
1.664322 1.975856
```

Test the Equality of Variance For a Two-sample T-test

```
> r2 = myTtest(a, c)
[1] "Variance are equal"
[1] "t: -1.06"
[1] "DF: 17"
[1] "p: 0.3"
> r2
Two Sample t-test
data: x and y
t = -1.0587, df = 17, p-value = 0.3046
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-6.605105 2.191275
sample estimates:
mean of x mean of y
1.664322 3.871237
```

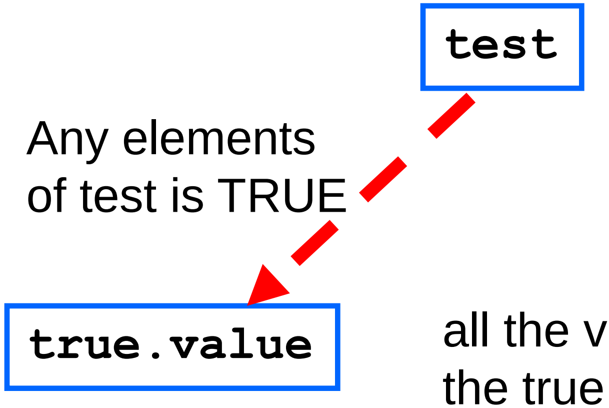
The ifelse and switch Functions
The ifelse Function

- The **ifelse** function has the following form:



```
ifelse (test, true.value, false.value)
```

All the arguments are vectors



test

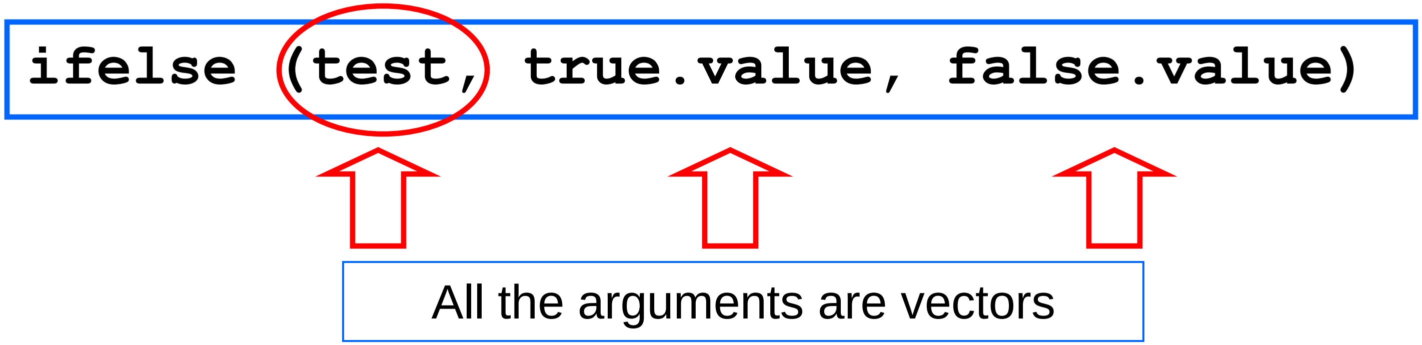
Any elements of test is TRUE

true.value

all the values in the true.value vector will need to be evaluated

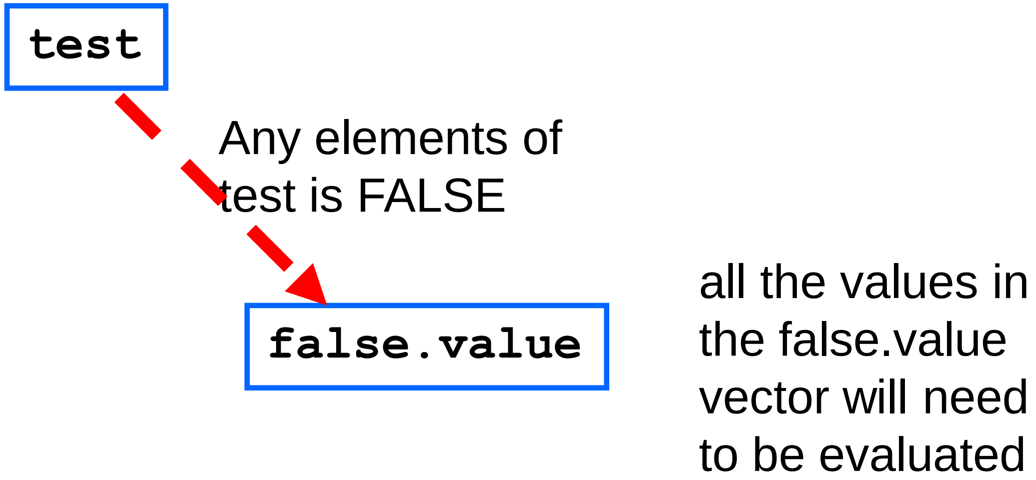
The `ifelse` Function

- The **`ifelse`** function has the following form:



```
ifelse (test, true.value, false.value)
```

All the arguments are vectors



`test`

Any elements of test is FALSE

`false.value`

all the values in the false.value vector will need to be evaluated

The `ifelse` Function

- Example: create an indicator variable

```
> treatment = c(rep("case", 3), rep("control", 2))
> treat.ind = ifelse(treatment == "case", 1, 0)
> treat.ind
[1] 1 1 1 0 0
```

The `ifelse` Function

❖ Example: Taking square root of a negative number

```
> x = -3:5
> sqrt(x)
[1]      NaN      NaN      NaN 0.000000 1.000000 1.414214
1.732051
[8] 2.000000 2.236068
Warning message:
In sqrt(x) : NaNs produced
```

⚐ Use the **`ifelse`** function to avoid the warning message

```
> sqrt(ifelse(x >= 0, x, NA))
[1]       NA       NA       NA 0.000000 1.000000 1.414214
1.732051
[8] 2.000000 2.236068
```

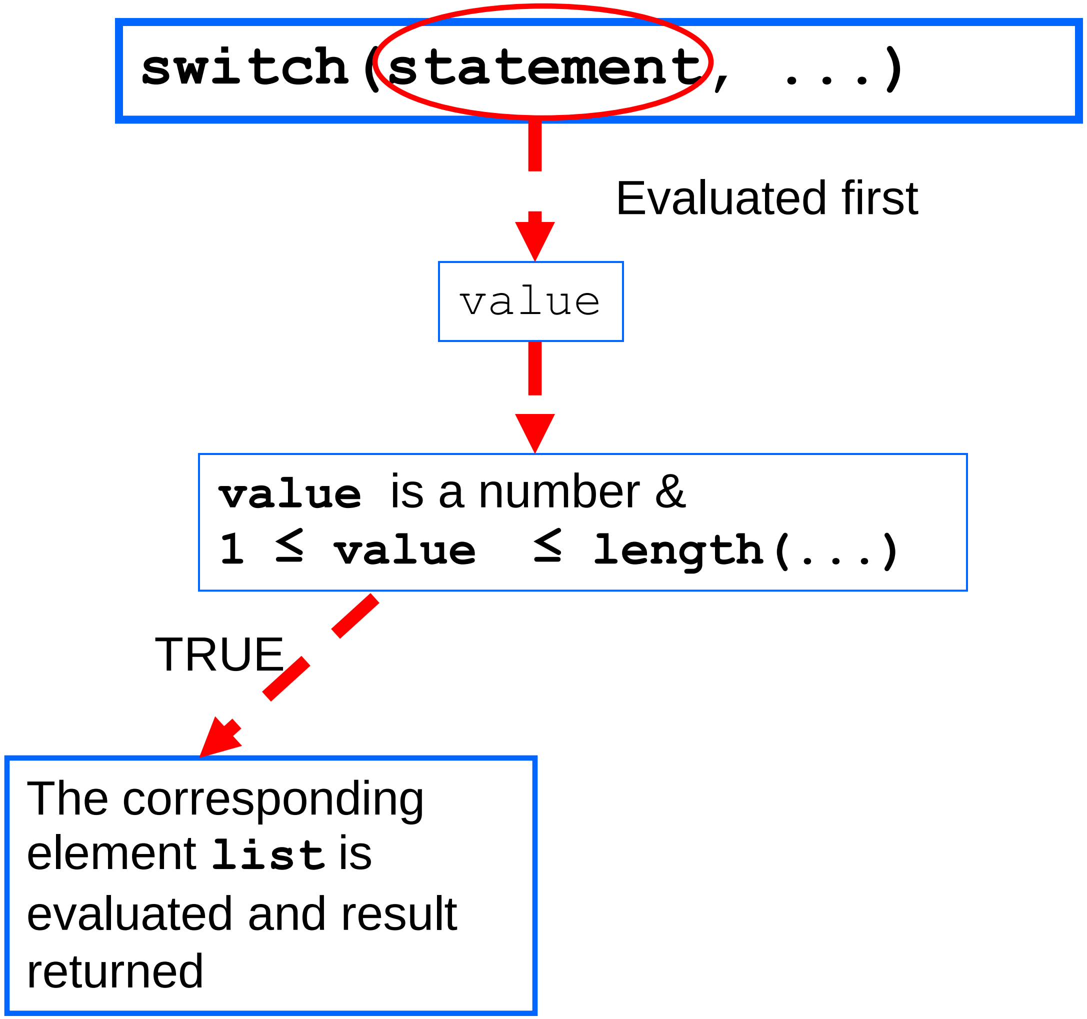
The `ifelse` Function

- The following code will still generate warning message since all the arguments in the **`ifelse`** function are evaluated

```
> ifelse(x >=0, sqrt(x), NA)
[1]       NA       NA       NA 0.000000 1.000000 1.414214
1.732051
[8] 2.000000 2.236068
Warning message:
In sqrt(x) : NaNs produced
```

The `switch` Function

- The **`switch`** function has the following form



`switch(statement, ...)`

Evaluated first

`value`

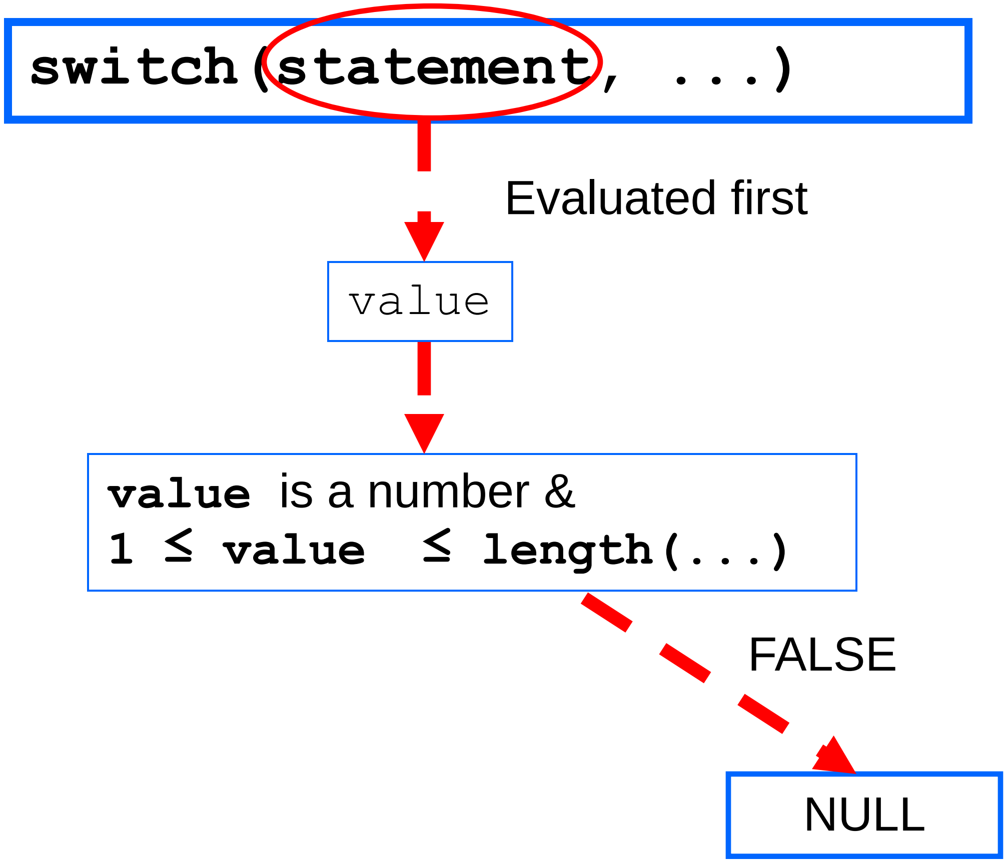
`value` is a number &
`1 ≤ value ≤ length(...)`

TRUE

The corresponding element **`list`** is evaluated and result returned

The switch Function

- The **switch** function has the following form



switch(statement, ...)

Evaluated first

value

value is a number &

1 ≤ value ≤ length(...)

FALSE

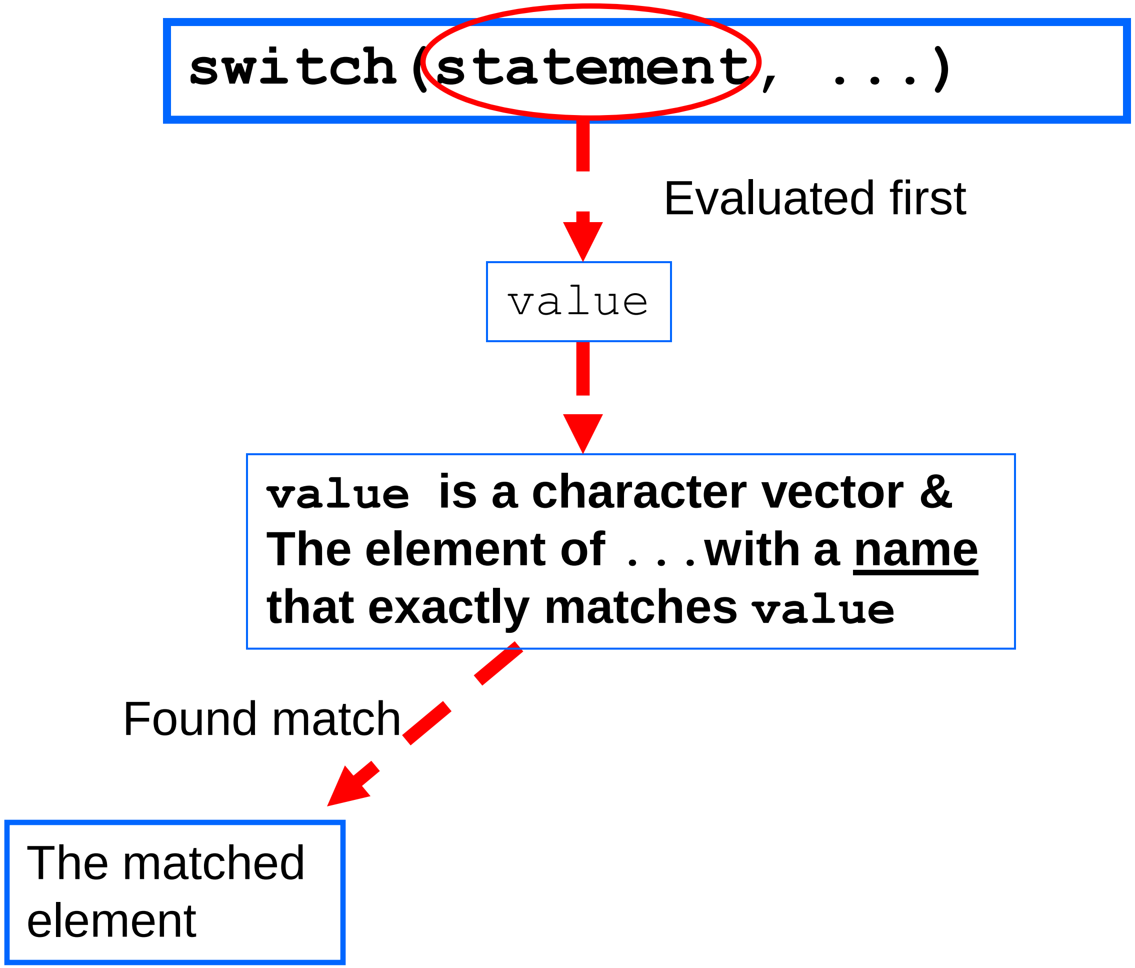
NULL

The `switch` Function

```
> x = 3
> switch(x, 2+2, mean(1:10), rnorm(5))
[1]  0.9319961  0.6316102  1.3671494 -2.3049011 -1.9256352
> switch(2, 2+2, mean(1:10), rnorm(5))
[1] 5.5
> foo = switch(6, 2+2, mean(1:10), rnorm(5))
> foo
NULL
```

The `switch` Function

- The **`switch`** function has the following form



```
switch(statement, ...)
```

Evaluated first

`value`

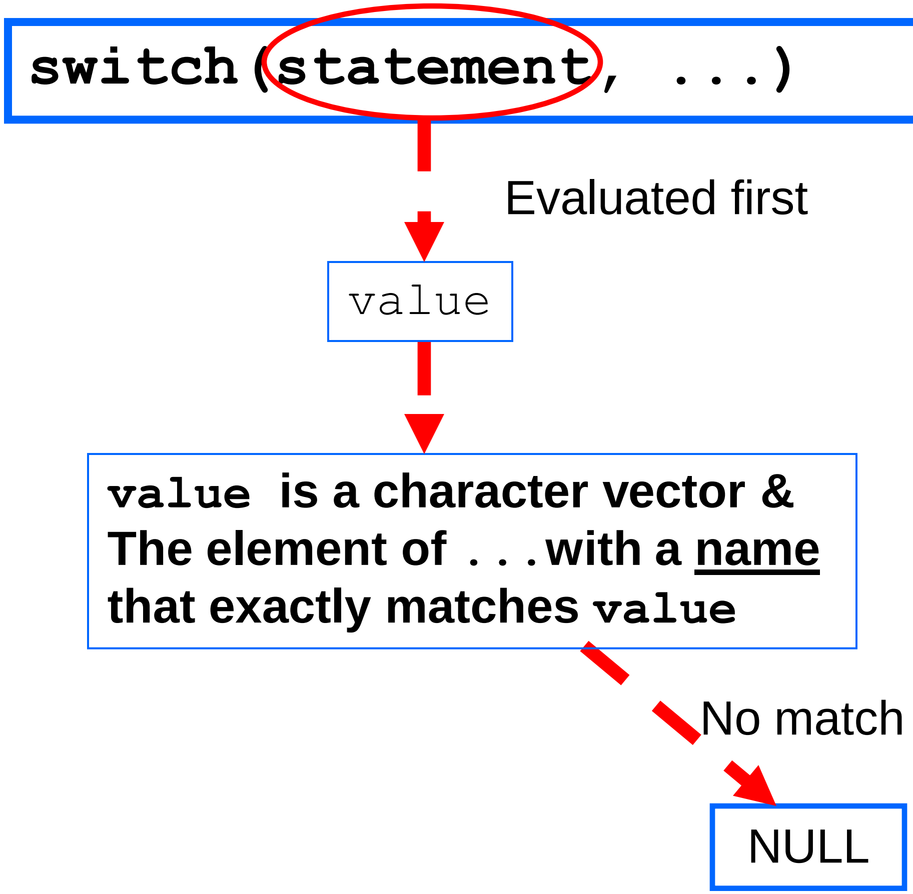
`value` is a character vector & The element of `...` with a <u>name</u> that exactly matches `value`

Found match

The matched element

The `switch` Function

❖ The **`switch`** function has the following form



```
switch(statement, ...)
```

Evaluated first

`value`

`value` is a character vector & The element of `...` with a <u>name</u> that exactly matches `value`

No match

NULL

The `switch` Function

```
> y = "fruit"
> switch(y, fruit = "banana", veggi = "broccoli", meat =
+ "beaf")
[1] "banana"
```

The switch Function

❖ Make a selection according to the character value of one of the arguments to a function

```
central = function(y, measure){
    switch(measure, mean = return(mean(y)),
                    harmonic = return(1/mean(1/y)),
                    median = return(median(y)),
                    stop ("Your measure is not supported"))
}
```

```
> z = rnorm(10, mean = 2, sd = 3)
> central(z, "mean")
[1] 1.570131
```

Looping

- ❖ Two types of loop: implicit and explicit

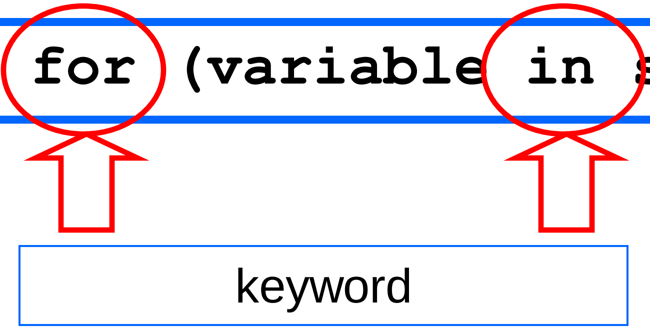
- ❖ Three types of explicit loop: **`for`**, **`while`**, and **`repeat`**

- ❖ **`break`**, and **`next`** can be used explicitly to control looping
  - ❑**`break`** causes an exist from the innermost loop
  - ❑**`next`** causes control to return the start of the loop

The `for` statement

❖ **`for`** loop has the following form:

❖ **`for (variable in sequence) statement`**

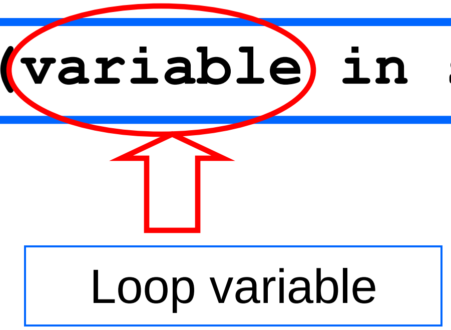


keyword

The `for` statement

- **`for`** loop has the following form:

- `for (variable in sequence) statement`



Loop variable

The `for` statement

- **`for`** loop has the following form:

```
for (variable in sequence) statement
```

vector of values

The `for` statement

- ❖ **`for`** loop has the following form:

- ❖ **`for (variable in sequence) statement`**

For a grouped statement, enclosed within { }

The `for` statement

- **`for`** loop has the following form:

- **`for (variable in sequence) statement`**

```
> ss <- 0
> total <- 0
> for (i in c(20, 30, 25, 40)){
+      total <- total + i
+      ss <- ss + i^2
+ }
> total
[1] 115
> ss
[1] 3525
```

The `for` statement

❖ Previous example is equivalent to …

```
> i <- c(20, 30, 25, 40)
> total.1 <- sum(i)
> ss.1 <- sum(i^2)
> total.1
[1] 115
> ss.1
[1] 3525
```

The **while** statement

- The **while** loop has the following form

```
while (condition) statement
```

If **condition** is **TRUE**, **statement** is evaluated
This process continues until **statement** is evaluates to **FALSE**

```
> x <- 0
> test <- 1
> while (test > 0) {
+     x <- x + 1
+     print (x^2)
+     test <- x < 6
+ }
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

The `repeat` statement

- The **`repeat`** loop has the following form

```
Repeat {statement}
```

`statement` must be a block statement

```
> x <- 0
> repeat {
+       x <- x + 1
+       print (x^2)
+       if (x == 6) break
+ }
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```