W8

Irlanda Ayon-Moreno

4/22/2018

Introduction to loops

For Loops

The anatomy of a `for` loop is as follows:

```
for (iterator in times) {
  do_something
}
```

`for()` takes an **iterator** variable and a vector of **times** to iterate through.

```
value <- 2

for (i in 1:5) {
  value <- value * 2
  print(value)
}
```

```
## [1] 4
## [1] 8
## [1] 16
## [1] 32
## [1] 64
```

The vector of *times* does NOT have to be a numeric vector; it can be any vector

For Loops and Next statement

Sometimes we need to skip a loop iteration if a given condition is met, this can be done with a next statement

```
for (iterator in times) {
  expr1
  expr2
  if (condition) {
    next
  }
  expr3
  expr4
}
```

Example:

```r
x <- 2

for (i in 1:5) {
  y <- x * i
  if (y == 8) {
    next
  }
  print(y)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 10
```

Nested Loops

```r
for (iterator1 in times1) {
  for (iterator2 in times2) {
    expr1
    expr2
    ...
  }
}
```

Example: Nested loops

```r
# some matrix
A <- matrix(1:12, nrow = 3, ncol = 4)

A
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

Example: Nested Loops

```r
# reciprocal of values less than 6
for (i in 1:nrow(A)) {
  for (j in 1:ncol(A)) {
    if (A[i,j] < 6) A[i,j] <- 1 / A[i,j]
  }
}

A
```

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

```
## [1,] 1.0000000 0.25    7   10
## [2,] 0.5000000 0.20    8   11
## [3,] 0.3333333 6.00    9   12
```

Repeat Loop

`repeat` executes the same code over and over until a stop condition is met:

```
repeat {
  # keep
  # doing
  # something
  if (stop_condition) {
    break
  }
}
```

The `break` statement stops the loops. If you enter an infinite loop, you can manually break it by pressing the `ESC` key.

```
value <- 2

repeat {
  value <- value * 2
  print(value)
  if (value >= 40) {
    break
  }
}
```

While Loops

It can also be useful to repeat a computation until a condition is false. A `while` loop provides this form of control flow.

```
while (condition) {
  # keep
  # doing
  # something
  # until
  # condition is FALSE
}
```

Repeat, While, For

- If you don't know the number of times something will be done, you can use either `repeat` or `while`

- `while` evaluates the condition at the beginning
- `repeat` executes operations until a stop condition is met
- If you know the number of times that something will be done, use `for`
- `for` needs an *iterator* and a vector of *times*

More about functions

Function Arguments

If you specify an argument with no default value, you must give it a value everytime you call the function, otherwise you'll get an error:

```
sqr <- function(x) {
  y <- x^2
  return(y)
}

# be careful
sqr()
```

```
## Error in sqr(): argument "x" is missing, with no default
```

Sometimes you don't want to give default values, but you also don't want to cause an error. We can use `missing()` to see if an argument is missing:

```
abc <- function(a, b, c = 3) {
  if (missing(b)) {
    result <- a * 2 + c
  } else {
    result <- a * b + c
  }
  return(result)
}
```

You can also set an argument value to NULL if you don't want to specify a default value:

```
abcd <- function(a, b = 2, c = 3, d = NULL) {
  if (is.null(d)) {
    result <- a * b + c
  } else {
    result <- a * b + c * d
  }
  return(result)
}
```

- named (default) arguments are created by naming the argument inside `function()`
- unmaned arguments are positional arguments
- Arguments can be matched positionally or by name

Messages

There are two main functions for generating warnings and errors:

- `stop()`
- `warning()`
- There's also the `stopifnot()` function

Stop Execution

Use `stop()` to stop the execution (this will raise an error)

```
meansd <- function(x, na.rm = FALSE) {
  if (!is.numeric(x)) {
    stop("x is not numeric")
  }
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}
```

Warning Messages

Use `warning()` to show a warning message

```
meansd <- function(x, na.rm = FALSE) {
  if (!is.numeric(x)) {
    warning("non-numeric input coerced to numeric")
    x <- as.numeric(x)
  }
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}
```

A warning is useful when you don't want to stop the execution, but you still want to show potential problems

Function `stopifnot()`

`stopifnot()` ensures the truth of expressions:

```
meansd <- function(x, na.rm = FALSE) {
  stopifnot(is.numeric(x))
  # output
  c(mean = mean(x, na.rm = na.rm),
    sd = sd(x, na.rm = na.rm))
}
```

```
meansd('hello')
```

```
## Error: is.numeric(x) is not TRUE
```

Iteration

Map Functions

The pattern of looping over a vector, doing something to each element and saving the results is so common that the purrr package provides a family of functions to do it for you. There is one function for each type of output:

`map()` makes a list.
`map_lgl()` makes a logical vector.
`map_int()` makes an integer vector.
`map_dbl()` makes a double vector.
`map_chr()` makes a character vector.

Each function takes a vector as input, applies a function to each piece, and then returns a new vector that's the same length (and has the same names) as the input. The type of the vector is determined by the suffix to the map function.

Example:

```
map_dbl(df, mean)
```

```
   a       b       c       d
```

0.2026 -0.2068 0.1275 -0.0917