Itérations (`for`)

Algo & Prog avec R

A. Malapert, B. Martin, M. Pelleau, et J.-P. Roy

11 septembre 2021

Université Côte d'Azur, CNRS, I3S, France
`firstname.lastname@univ-cotedazur.fr`

La boucle `for`

En théorie, la boucle `while` permet de réaliser toutes les boucles que l'on veut. Toutefois, les boucles `for` sont très utilisées.

```
S <- function(n) {
  i <- 0
  acc <- 0
  while(i < n) {
    i <- i + 1
    acc <- acc + i
  }
  return(acc)
}
```

```
S <- function(n) {
  acc <- 0
  for(i in 1:n) {
    acc <- acc +i
  }
  return(acc)
}
```

- Souvent, on utilisera une boucle for pour incrémenter ou décrémenter un compteur.
- La boucle `for` est plus pratique ici, mais pas pour l'épluchage des entiers.

Échappement d'une boucle `for`

Lorsque l'instruction `return` ou `break` est placée à l'intérieur de la boucle `for`, cela signifie :

Je parcours a priori *tout* une séquence,
mais je me réserve la possibilité de *m'échapper* en cours de route !

Exemple : comparaison d'une valeur à une séquence
Est-ce que la valeur `x` est plus grande ou égale aux valeurs d'une séquence ?

```
geq <- function(x, values) {
  for(v in values) {
    if(x < v) return(FALSE)
  }
  return(TRUE)
}
```

```
> values <- c(2, 4, 1, 7, 5)
> geq(0, values)
[1] FALSE
> geq(1, values)
[1] FALSE
> geq(7, values)
[1] TRUE
> geq(8, values)
[1] TRUE
```

Une séquence ?

Nous commençons à pénétrer dans un continent qu'il faudra tôt ou tard aborder : celui des VECTEURS et LISTES. Nous garderons pour l'instant une idée naïve de ce qu'est un vecteur, une séquence d'éléments.

La fonction `seq` : générer une séquence régulière

`seq(from, to, by, length.out, ...)`

```
> seq(5) #ou mieux seq_len(5)
[1] 1 2 3 4 5
> seq(0,5)
[1] 0 1 2 3 4 5
> seq(from=0,to=5)
[1] 0 1 2 3 4 5
> seq(from=0,to=5,by=2)
[1] 0 2 4
> seq(from=0,to=5, by=1.25)
[1] 0.00 1.25 2.50 3.75 5.00
> seq(from=0,to=5, length.out=5)
[1] 0.00 1.25 2.50 3.75 5.00
```

En version courte pour générer des entiers consécutifs :

```
> 1:5
[1] 1 2 3 4 5
> 1:0 # /!\ ATTENTION !
[1] 1 0
```

La boucle for parcourt un objet itérable

Nous avons utilisé jusqu'à présent la boucle `for` sur une séquence d'entiers. Or `seq` ne construit pas forcément l'intervalle, mais un itérateur sur un vecteur virtuel. Heureusement d'ailleurs :

```
for(i in seq(1,10**7, by=2)) {
  print(i)
  if(i > 10) break
}
```

```
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
[1] 11
```

Les séquences (vector, list) sont des objets itérables .

Syntaxe

Syntaxe de la boucle `for`

```
for (x in sequence) {
  ...
}
```

La fonction `sample` : générer une séquence aléatoire

On va choisir aléatoirement `size` éléments du vecteur `x` avec ou sans remise (`replace`) et avec ou sans biais (`prob`).

```
sample(x, size, replace = FALSE, prob = NULL)
```

Tirage sans remise (`replace=FALSE`)
La variante `sample.int` est un raccourci pour choisir dans `1:n`

```
> sample.int(10) # permutation aléatoire de [1, 10]
[1]  8  6 10  7  5  4  3  1  2  9
> sample.int(n = 90,  size = 6) # tirage du loto
[1]  5 26 11 77 70 32
> sample.int(n = 10,  size = 11) # Choisir 11 parmi 10
Error in sample.int(10, 11) :
  impossible de prendre un échantillon plus grand que la
      population lorsque 'replace = FALSE'
```

La fonction `sample` ...

On va choisir aléatoirement `size` éléments du vecteur `x` avec ou sans remise (`replace`) et avec ou sans biais (`prob`).

```
sample(x, size, replace = FALSE, prob = NULL)
```

Tirage avec remise (`replace=TRUE`)

```
>  sample(c('pile', 'face'),  size = 5, replace = TRUE)
[1] "face" "pile" "pile" "pile" "pile"
```

Tirage biaisé avec remise (`prob`)

```
> sample(c('pile', 'face'),  size = 10, replace = TRUE, prob
    = c(5, 1))
 [1] "pile" "pile" "pile" "face" "face" "pile" "pile" "pile"
     "pile" "pile"
```

Générer des nombres approchés aléatoires

On va tirer des `n` nombres approchés aléatoires compris entre `min` et `max` avec une probabilité uniforme.

`runif(n, min = 0, max = 1)`

```
> runif(5)
[1] 0.36051021 0.96824951 0.08495143 0.87527313 0.16520820
> runif(n = 5, min = 0, max = 10)
[1] 7.945856 8.398957 2.757909 6.035876 1.205764
```

Autres distributions
`rnorm`, `rpois`, `rgamma` ...

Mais, il faut éviter les boucles !

Quand c'est possible, il faut mieux utiliser une fonction prédéfinie.

```
SP <- function(n) {
  if(n <= 0) return(0)
  return(sum(1:n))
}
```

```
SR <- function(n) {
  if(n > 0) {
    return(n+SR(n-1))
  }
  else return(0)
}
```

```
SW <- function(n) {
  i <- 0
  acc <- 0
  while(i < n) {
    i <- i + 1
    acc <- acc +i
  }
  return(acc)
}
```

```
SF <- function(n) {
  if(n <= 0) return(0)
  acc <- 0
  for(i in 1:n) {
    acc <- acc +i
  }
  return(acc)
}
```

La vectorisation est plus efficace !

```
> system.time(replicate(10**4, invisible(SP(10**6))))
utilisateur     système      écoulé
      0.041       0.000       0.042
```

	$n = 10^3$ temps (s)	$n = 10^4$ temps (s)
SR	3.200	overflow
SW	0.547	4.664
SF	0.260	2.455
SP	0.011	0.011

- La suppression des boucles s'appelle la vectorisation.
- Nous découvrirons plus tard autre famille de boucles : `apply`.

Questions?

Retrouvez ce cours sur le site web

`www.i3s.unice.fr/~malapert/R`