1_2_cours

October 15, 2018

Table of Contents

1 les opérateurs de base

R est d'abord une calculatrice mathématique:

1.1 Arithmetic

D'abord créons quelques variables;

```
In [1]: x <- 12
        y <- 10
        z<--1 #ou  z <- -1
```

```
In [2]: x+y+z
```

21

```
In [3]: x-y-z
```

3

1.2 multiplication / division

```
In [4]: x*y
```

120

```
In [5]: x/y
```

1.2

1.3 Puissance

```
In [6]: x^2
```

144

```
In [7]: x**2
```

144

1.4 Arrondir

1.4.1 round()

```
In [8]: pi
```

3.14159265358979

```
In [9]: round(pi,2)
```

3.14

1.4.2 floor

```
In [10]: floor(pi)
```

3

1.4.3 ceiling

```
In [11]: ceiling(pi)
```

4

1.4.4 valeur absolue

```
In [12]: z
```

-1

```
In [13]: abs(z)
```

1

On peut aussi effectuer un calcul à l'intérieur d'un autre calcul. Essayons de calculer la racine carrée de z

```
In [14]: # sqrt(z)

In [15]: sqrt(abs(z))
```

1

1.4.5 les parenthèses

```
In [16]: x^(1/2)
```

3.46410161513775

```
In [17]: x^1/2
```

6

1.5 logarithme

```
In [18]: log(x)
```

2.484906649788

```
In [19]: log10(x)
```

1.07918124604762

```
In [20]: exp(1)
```

2.71828182845905

1.6 Factoriel

```
In [21]: factorial(4)
```

24

```
In [22]: 4*3*2*1
```

24

2 Les vecteurs

Nous avons vu que nous avons créé des variables, nous leur avons donnée des valeurs et nous avons effectué des opérations mathématiques sur ces variables. Nous avons vu que nous avons chargé des *packages*, nous avons également appelé des fonctions des données venant de *packages* de base ou incluses dans des *packages* que nous avons chargés.

On commence à comprendre alors que dans R, tous les éléments sont des **objets**. Ces derniers sont créés et interagissent entre elles par des instructions (des lignes de code) du programmeur. Par exemple si nous créons une variable de x de valeur 21, nous venons alors de créer un seul objet x ayant une valeur 21

```
In [23]: x<-21
```

Nous pouvons en tout moment faire appel à ce vecteur par son attribut;

```
In [24]: x
```

21

On peut aussi créer une collection de valeurs à l'intérieur d'**un seul** objet. C'est ce que nous appelons **un vecteur**

```
In [25]: vecteur <-c(1,2,3,4,5)
```

Notre vecteur est un ensemble d'entiers

$$vecteur = \{1, 2, 3, 4, 5\}$$

Regardons la valeur de cet objet que R a gardé en mémoire;

```
In [26]: vecteur
```

1. 1 2. 2 3. 3 4. 4 5. 5

Regardons maintenant le type de cet objet;

```
In [27]: class(vecteur)
```

'numeric'

Tel qu'attendu, R a automatiquement reconnu le type de ce vecteur sans avoir à le définir. Contrairement à d'autres langages de programmation, tel que `C++`, R a donné le type `numeric` à cet objet.

2.1 autres façons de créer des vecteurs

```
In [28]: vecteur <-1:5
```

```
In [29]: vecteur
```

1. 1 2. 2 3. 3 4. 4 5. 5

```
In [30]: vecteur <-10:17
```

```
In [31]: vecteur
```

1. 10 2. 11 3. 12 4. 13 5. 14 6. 15 7. 16 8. 17

Ceci fonctionne aussi pour les décimales

```
In [32]: vecteur <- 17.5:22.5
```

```
In [33]: vecteur
```

1. 17.5 2. 18.5 3. 19.5 4. 20.5 5. 21.5 6. 22.5

On remarque que les deux façons qu'on vient de voir créent une suite de données avec un saut de 1 entre chaque valeur. On peut utiliser la fonction `seq(from = ..., to = ..., by = ...)` afin de changer le saut.

```
In [34]: # ?seq
```

```
In [35]: vecteur <-seq(from = 17.5,to = 29.5,by = 2)
```

```
In [36]: vecteur
```

1. 17.5 2. 19.5 3. 21.5 4. 23.5 5. 25.5 6. 27.5 7. 29.5

2.2 Extraction des valeurs

Supposons qu'on voudrait extraire la première valeur se trouvant dans le vecteur;

```
In [37]: vecteur # ce vecteur avait comme valeurs
```

1. 17.5 2. 19.5 3. 21.5 4. 23.5 5. 25.5 6. 27.5 7. 29.5

```
In [38]: vecteur[1] #le premier objet
```

17.5

Le troisième objet maintenant;

```
In [39]: vecteur[3]
```

21.5

```
In [40]: vecteur[1:3] # du premier au troisième objet
```

1. 17.5 2. 19.5 3. 21.5

Si l'on voulait tous sauf le nième élément, on utilise le signe (-)

```
In [41]: vecteur [-1] # tout sauf le premier
```

1. 19.5 2. 21.5 3. 23.5 4. 25.5 5. 27.5 6. 29.5

```
In [42]: vecteur[-(1:3)] # tous sauf les trois premiers objets
```

1. 23.5 2. 25.5 3. 27.5 4. 29.5

2.3 Combinaison de vecteurs

Puisque les vecteurs sont des objets ou une collection d'objets, on peut combiner plusieurs vecteurs et effectuer des opérations sur ces derniers.

Créons trois vecteurs;

```
In [43]: x<-c(1,2,3)
         y <- c(.17, .66, .44)
         z<-11
```

```
In [44]: vecteur <-c(x,y,z)
```

```
In [45]: vecteur
```

1. 1 2. 2 3. 3 4. 0.17 5. 0.66 6. 0.44 7. 11

```
In [46]: vecteur <-c(x,y,z, -123:-126)
```

```
In [47]: vecteur
```

1. 1 2. 2 3. 3 4. 0.17 5. 0.66 6. 0.44 7. 11 8. -123 9. -124 10. -125 11. -126

2.4 Taille des vecteurs

```
In [48]: length(vecteur)
```

11

2.5 Plusieurs types

Lorsque nous créons un vecteur, R se charge de donner un type à ce dernier. Toutefois, il est possible de mélanger les données de types différents.

2.5.1 Numeric

Nous avons vu plusieurs exemples de ce type auparavant

2.5.2 Character

```
In [49]: vect_stri <-c("bonjour", "Hi", Buongiorno")
```

```
In [50]: vect_stri
```

1. 'bonjour' 2. 'Hi'

Regardons maintenant la `class` de cette variable;

```
In [51]: class(vect_stri)
```

'character'

2.5.3 logical

```
In [52]: vect_bool <- c(T,F,F,F,T,F)
```

```
In [53]: vect_bool
```

1. TRUE 2. FALSE 3. FALSE 4. FALSE 5. TRUE 6. FALSE

```
In [54]: class(vect_bool)
```

'logical'

2.5.4 Mélange

On peut aussi avoir de multiples class dans un seul vecteur;

```
In [55]: vect_melange_type<-c(123,"bonjour", T)
```

```
In [56]: vect_melange_type
```

1. '123' 2. 'bonjour' 3. 'TRUE'

On remarque que R s'est chargé de tout transformer en type caractère, car "bonjour" ne peut pas prendre le numérique

```
In [57]: class(vect_melange_type)
```

'character'

2.6 Opértation sur les vecteurs

Créons notre vecteur qui nous sert d'exemple'

```
In [58]: vecteur <- c(1:5)
         vecteur
```

1. 1 2. 2 3. 3 4. 4 5. 5

Multiplions ce vecteur par deux, donc toutes les valeurs contenues à l'intérieur de ce dernier;

```
In [59]: vecteur*2
```

1. 2 2. 4 3. 6 4. 8 5. 10

Ou appliquons une simple adition à lui même;

```
In [60]: vecteur+vecteur
```

1. 2 2. 4 3. 6 4. 8 5. 10

Si l'on ajoute 5 au vecteur, R se charge d'ajouter 5 à chaque élément du vecteur

```
In [61]: vecteur+5
```

1. 6 2. 7 3. 8 4. 9 5. 10

```
In [62]: vecteur**2
```

1. 1 2. 4 3. 9 4. 16 5. 25

```
In [63]: vecteur^2
```

1. 1 2. 4 3. 9 4. 16 5. 25

Un autre façon de créer le même vecteur;

```
In [64]: vecteur2 <- 2*(1:5)
         vecteur2
```

1. 2 2. 4 3. 6 4. 8 5. 10

```
In [65]: (vecteur*3+vecteur2^2+17)/2
```

1. 12 2. 19.5 3. 31 4. 46.5 5. 66

2.6.1 Attention!

Note Ces opérations fonctionnent sur des vecteurs de même taille. Essayons par exemple d'additionner deux vecteurs de différente taille; le premier contient cinq éléments et le second en contient six.

```
In [66]: vecteur5 <-c(1:5)
         vecteur6 <-c(1:6)
```

```
In [67]: vecteur5+vecteur6
```

```
Warning message in vecteur5 + vecteur6:
''longer object length is not a multiple of shorter object length''
```

1. 2 2. 4 3. 6 4. 8 5. 10 6. 7

Nous avons quand même un résultat, mais nous avons un avertissement que la taille des deux vecteurs est différente.

On voit que le résultat contient six éléments, `R` s'est chargé de réappliquer le même calcul (adition) entre le sixième élément du vecteur6 et le premier élément du vecteur5 puisque le sixième de ce dernier est manquant.

3 Sequences

```
In [68]: ?seq
```

Cette fonction sert à générer une série de séquence régulière. Les arguments sont; * `from`: qui est le début de la séquence qu'on voudrait générer. * `to`: la fin de la séquence * `by`: l'incrément avec lequel on veut augmenter notre séquence

```
In [69]: seq(from = 1, to = 5,by=.3)
```

1. 1 2. 1.3 3. 1.6 4. 1.9 5. 2.2 6. 2.5 7. 2.8 8. 3.1 9. 3.4 10. 3.7 11. 4 12. 4.3 13. 4.6 14. 4.9

Un autre exemple où l'on crée un incrément de `pi` dans une série de 1 à 10;

```
In [70]: seq(1, 10, pi)
```

1. 1 2. 4.14159265358979 3. 7.28318530717959

Si l'on omet de mettre tous les arguments, l'incrément devient alors 1 par défaut.

```
In [71]: seq(5)
```

1. 1 2. 2 3. 3 4. 4 5. 5

Cela fonctionne aussi si l'on veut générer des nombres négatifs

```
In [72]: seq(-1,-5, -.3)
```

1. -1 2. -1.3 3. -1.6 4. -1.9 5. -2.2 6. -2.5 7. -2.8 8. -3.1 9. -3.4 10. -3.7 11. -4 12. -4.3 13. -4.6 14. -4.9

Cette fonction nous sert plus souvent lorsqu'on veut générer des vecteurs. D'ailleurs, vérifions si une séquence générée possède les mêmes valeurs qu'un vecteur.

```
In [73]: seq(5)
```

1. 1 2. 2 3. 3 4. 4 5. 5

```
In [74]: 1:5
```

1. 1 2. 2 3. 3 4. 4 5. 5

```
In [75]: seq(5)==1:5
```

1. TRUE 2. TRUE 3. TRUE 4. TRUE 5. TRUE

Dans la ligne de code ci-haut on s'aperçoit que chaque valeur de la séquence est égale à chaque valeur du vecteur. Si l'on veut savoir par une seule réponse booléenne si toutes les valeurs du vecteur sont égales.

```
In [76]: all(seq(5)==1:5)
```

TRUE

Ou le contraire, est-ce toutes les valeurs sont différentes?

```
In [77]: !all(seq(5)==1:5)
```

FALSE

4 Plus de resources

- **The R Project for Statistical Computing**: (http://www.r-project.org/) Premier lieu où.
- **The Comprehensive R Archive Network**: (http://cran.r-project.org/) C,est là ou se trouve le logiciel R, avec des miliers de *packages*, il s,y trouve aussi des exemples et même des livres!
- **R-Forge**: (http://r-forge.r-project.org/) Une autre place où des *packages* sont sauvegardé, on y trouve aussi des *packages* tout récemment développés
- **Rlanguage reddit**: (https://www.reddit.com/r/Rlanguage) On y trouve toutes sortes d'informations ou question exemple