Programmation orientée objet en R

Sophie Baillargeon, Université Laval

2020-03-13

Table des matières

La programmation orientée objet est un paradigme de programmation basé sur le concept d'objets, qui peuvent contenir des données et métadonnées (attributs), et qui sont associés à des procédures, souvent appelées méthodes. R propose plusieurs systèmes orientés objet en R :

- **S3** : Il s'agit du système le moins formel, mais le plus utilisé, en particulier dans les packages `base` et `stats`. Ces notes traitent principalement de ce système.
- **S4** : Ce système fonctionne de façon similaire au système S3, mais il est plus formel. La majorité des packages sur Bioconductor utilisent ce système. Nous verrons ici comment manipuler des objets dont la classe est de type S4, mais pas comment en créer.
- Autres : Parmi les autres systèmes orientés objet en R, il y a eu **RC** (aussi nommé **R5**), le système « Reference Classes ». Celui-ci n'a cependant jamais été très utilisé. Récemment, le système **R6** a vu le jour, afin de remplacer le système RC. Comparativement aux systèmes S3 et S4, ce système se rapproche davantage du paradigme orienté objet des langages informatiques Python et Java notamment. Comparativement au système RC, il est plus simple et plus rapide. Ces autres systèmes ne seront pas approfondis dans ce cours.

Système S3

À chaque fois que nous avons effleuré le sujet de la programmation orientée objet dans ce cours, nous parlions toujours du système S3. Nous l'avons mentionné dans les notes sur :

- les concepts de base, à la section « Est-ce que de la programmation orientée objet est possible en R ? » ;

- les graphiques, à la section « Fonction générique plot » ;
- les calculs statistiques et mathématiques, à la section« Manipulation de la sortie d'une fonction d'ajustement de modèle ».

Le fonctionnement de ce système est très simple. Il est possible d'attribuer des *classes* de type S3 aux objets R. Ces classes déterminent comment les *fonctions génériques* se comportent en recevant en entrée un certain objet. Une fonction générique est dite *polymorphe*. Elle possède plusieurs définitions, appelées *méthodes*, pour des objets de différentes classes. Techniquement, une fonction générique R ne fait que rediriger les arguments qui lui sont fournis en entrée vers la méthode associée à la classe de l'objet assigné à son premier argument.

Voyons plus en détail le fonctionnement du système S3, puis apprenons comment créer une nouvelle méthode pour une fonction générique existante. Nous ne verrons pas comment créer de nouvelles fonctions génériques.

Fonctionnement du système S3

Nous avons déjà mentionné que les fonctions suivantes sont génériques : `plot`, `print`, `summary`, `coef` et plusieurs autres fonctions d'extraction d'informations de la sortie d'une fonction d'ajustement de modèle. La fonction `print` est probablement la fonction de cette liste que nous utilisons le plus souvent, puisqu'elle est appelée à chaque fois que nous soumettons dans la console une commande contenant uniquement le nom d'un objet. La définition de cette fonction est la suivante :

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x000000000f51f880>
## <environment: namespace:base>
```

Il n'y a qu'une seule instruction dans le corps de cette fonction : un appel à la fonction `UseMethod`. Toutes les fonctions génériques ont cette forme. De plus, la plupart d'entre elles possèdent un argument « `...` ».

Lorsqu'une fonction générique est appelée, la fonction `UseMethod` vérifie d'abord la classe de l'objet fourni au premier argument, ici `x`. Ensuite, elle appelle la méthode correspondant à la classe obtenue, si elle existe, en lui fournissant en entrée les arguments qui ont été fournis dans l'appel à la fonction générique.

Par exemple, considérons le data frame suivant (il s'agit d'un jeu de données du package `datasets`).

```
str(women)
```

```
## 'data.frame':    15 obs. of  2 variables:
##  $ height: num  58 59 60 61 62 63 64 65 66 67 ...
##  $ weight: num  115 117 120 123 126 129 132 135 139 142 ...
```

Les data frames possèdent un attribut `class`.

```
attributes(women)
```

```
## $names
## [1] "height" "weight"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Lors de l'évaluation de l'appel à la fonction `print` suivant :

```
print(women)
```

R va d'abord vérifier la classe de l'objet assigné au premier argument dans l'appel de la fonction :

```
class(women)
```

```
## [1] "data.frame"
```

Puis R va vérifier si une méthode `print` est définie pour cette classe. La fonction `methods` nous permet de connaître toutes les méthodes définies pour une fonction générique présente dans le chemin de recherche de notre session R.

```
methods(print)
```

Le résultat n'est pas affiché en entier, car le nombre de méthodes pour la fonction générique `print` est trop grand. Voici cependant un extrait de la sortie que j'obtiens.

```
## . . .
## [67] print.condition                                        print.connection
## [69] print.CRAN_package_reverse_dependencies_and_views* print.data.frame
## [71] print.Date                                             print.default
## [73] print.dendrogram*                                      print.density*
## . . .
## see '?methods' for accessing help and source code
```

Pour une classe spécifique, une méthode de type S3 portera le nom : `nom_fonction_generique.nom_classe`. Nous voyons ici que la méthode `print.data.frame` existe. Celle-ci est une fonction, dont la définition est la suivante :

```
print.data.frame
```

```
## function (x, ..., digits = NULL, quote = FALSE, right = TRUE,
##     row.names = TRUE, max = NULL)
## {
##     n <- length(row.names(x))
##     if (length(x) == 0L) {
##         cat(sprintf(ngettext(n, "data frame with 0 columns and %d row",
##             "data frame with 0 columns and %d rows"), n), "\n",
##             sep = "")
##     }
##     else if (n == 0L) {
##         print.default(names(x), quote = FALSE)
##         cat(gettext("<0 rows> (or 0-length row.names)\n"))
##     }
##     else {
##         if (is.null(max))
##             max <- getOption("max.print", 99999L)
##         if (!is.finite(max))
##             stop("invalid 'max' / getOption(\"max.print\"): ",
##                 max)
##         omit <- (n0 <- max%/%length(x)) < n
##         m <- as.matrix(format.data.frame(if (omit)
##             x[seq_len(n0), , drop = FALSE]
##         else x, digits = digits, na.encode = FALSE))
##         if (!isTRUE(row.names))
##             dimnames(m)[[1L]] <- if (isFALSE(row.names))
##                 rep.int("", if (omit)
##                   n0
##                 else n)
##             else row.names
##         print(m, ..., quote = quote, right = right, max = max)
```

```
##         if (omit)
##             cat(" [ reached 'max' / getOption(\"max.print\") -- omitted",
##                 n - n0, "rows ]\n")
##     }
##     invisible(x)
## }
## <bytecode: 0x000000000f741138>
## <environment: namespace:base>
```

R va donc finalement appeler cette fonction.

Donc, la commande

```
women
```

qui revient en fait à la commande

```
print(women)
```

cache l'évaluation de la commande suivante

```
print.data.frame(women)
```

qui produit le résultat suivant.

```
##    height weight
## 1      58    115
## 2      59    117
## 3      60    120
## 4      61    123
## 5      62    126
## 6      63    129
## 7      64    132
## 8      65    135
## 9      66    139
## 10     67    142
## 11     68    146
## 12     69    150
## 13     70    154
## 14     71    159
## 15     72    164
```

Situation particulière 1 : l'objet assigné au premier argument n'a pas d'attribut `"class"`

Dans l'exemple précédent, l'objet fourni en entrée au premier argument de la fonction générique possédait un argument nommé `"class"`.

```
attr(women, which = "class")
```

```
## [1] "data.frame"
```

Qu'arrive-t-il si l'objet assigné au premier argument n'a pas d'attribut `"class"` ?

En toute circonstance, la fonction `class` doit retourner une classe pour permettre en système S3 de bien fonctionner. Alors les objets R ont tous une classe implicite.

Par exemple, créons une copie de `women` pour laquelle nous allons retirer l'attribut `class` avec la fonction `unclass`.

```
women_2 <- unclass(women)
attr(women_2, which = "class")
```

```
## NULL
```

L'objet `women_2` n'a pas d'attribut `class`, mais il possède tout de même une classe implicite.

```
class(women_2)
```

```
## [1] "list"
```

L'objet `women_2` possède la classe implicite `"list"`. Ce résultat est cohérent avec le fait qu'un data frame est un type particulier de liste.

Bref, la fonction `class` fonctionne comme suit. Elle vérifie d'abord si l'objet qu'elle reçoit en entrée possède une classe explicite, soit un attribut nommé `"class"`. Si c'est le cas, elle retourne cet attribut. Sinon, elle retourne la classe implicite de l'objet.

Remarque

Notons que, alors que les listes ont la classe implicite `list`, les arrays ont la classe implicite `array`, les matrices ont la classe implicite `matrix` et les vecteurs ont une classe implicite correspondant au type de données qu'ils contiennent, par exemple :

```
x <- 1:5
x
```

```
## [1] 1 2 3 4 5
```

```
class(x)
```

```
## [1] "integer"
```

Les facteurs, pour leur part, ont un attribut `class`, tout comme les data frames.

```
x_factor <- factor(x)
x_factor
```

```
## [1] 1 2 3 4 5
## Levels: 1 2 3 4 5
```

```
attributes(x_factor)
```

```
## $levels
## [1] "1" "2" "3" "4" "5"
##
## $class
## [1] "factor"
```

```
class(x_factor)
```

```
## [1] "factor"
```

Situation particulière 2 : la méthode `nom_fonction_generique.nom_classe` n'existe pas

Qu'arrive-t-il si la fonction générique `nom_fonction_generique` reçoit comme premier argument en entrée un objet de classe `nom_classe`, mais que la méthode `nom_fonction_generique.nom_classe` n'existe pas ?

Dans ce cas, R utilise la méthode par défaut, si elle existe, sinon il retourne une erreur. La méthode par défaut d'une fonction générique se nomme toujours `nom_fonction_generique.default`, où `nom_fonction_generique` est le nom de la fonction générique.

Par exemple, reprenons l'objet `women_2` de classe implicite `"list"`. Comment agit la fonction générique `print` avec cet objet ?

```r
print(women_2)
```

```
## $height
##  [1] 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
##
## $weight
##  [1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 164
##
## attr(,"row.names")
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Lors de l'évaluation de la commande `print(women_2)`, R cherche d'abord à appeler la méthode `print.list`, mais celle-ci n'existe pas.

```r
any(methods(print) == "print.list")
```

```
## [1] FALSE
```

Il se rabat donc sur la méthode `print` par défaut et évalue l'appel de fonction suivant :

```r
print.default(women_2)
```

```
## $height
##  [1] 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
##
## $weight
##  [1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 164
##
## attr(,"row.names")
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Cependant, certaines fonctions génériques ne possèdent pas de méthode par défaut. Par exemple, c'est le cas de la fonction générique `anova`.

```r
methods(anova)
```

```
## [1] anova.glm*     anova.glmlist* anova.lm*      anova.lmlist*  anova.loess*
## [6] anova.mlm*     anova.nls*
## see '?methods' for accessing help and source code
```

Si nous donnons en entrée à la fonction `anova` un objet qui n'est pas de classe `glm`, `glmlist`, `lm`, `lmlist`, `loess`, `mlm` ou `nls`, une erreur est retournée.

```r
anova(women_2)
```

```
## Error in UseMethod("anova") :
##   no applicable method for 'anova' applied to an object of class "list"
```

Remarque

Notons que la fonction `methods` permet aussi d'énumérer les fonctions génériques possédant une méthode associée à une classe en particulier. Par exemple, les méthodes associées à la classe `lm` présentes dans le chemin de recherche de ma session R sont les suivantes :

```r
methods(class = "lm")
```

```
##  [1] add1           alias          anova          case.names     coerce
##  [6] confint        cooks.distance deviance       dfbeta         dfbetas
## [11] drop1          dummy.coef     effects        extractAIC     family
## [16] formula        hatvalues      influence      initialize     kappa
```

```
## [21] labels         logLik         model.frame    model.matrix   nobs
## [26] plot           predict        print          proj           qr
## [31] residuals      rstandard      rstudent       show           simulate
## [36] slotsFromS3    summary        variable.names vcov
## see '?methods' for accessing help and source code
```

Situation particulière 3 : `class` retourne plus d'une classe

Les objets R peuvent posséder plus d'une classe. Par exemple, les objets R retournés par la fonction `aov` ont deux classes : `aov` et `lm`.

```
model <- aov(Sepal.Length ~ Species, data = iris)
class(model)
```

```
## [1] "aov" "lm"
```

Qu'arrive-t-il lorsque `class` retourne plus d'une classe ?

Si un objet possédant plus d'une classe est fourni en entrée à une fonction générique, R cherche d'abord à utiliser la méthode associée à la première classe de la liste. Si celle-ci n'existe pas, R utilise la méthode associée à le seconde classe. Si celle-ci n'existe pas, R continue à parcourir le vecteur des noms de classe jusqu'à ce qu'il trouve une méthode pour une classe. S'il n'existe de méthode pour aucune classe de la liste, c'est la méthode par défaut qui est employée, si elle existe, sinon une erreur est générée, comme nous venons de le voir.

Situation particulière 4 : la fonction générique possède plus d'un argument

Certaines fonctions génériques, telle que la fonction `plot`, possèdent plus d'un argument.

```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x000000000f9fe6f0>
## <environment: namespace:graphics>
```

Quel impact ont ces autres arguments sur le choix de la méthode à utiliser ?

Ils n'ont aucun impact sur le choix de la méthode. Cette méthode peut cependant utiliser ces autres arguments pour déterminer ce qu'elle va faire. Par exemple, le corps de la méthode `plot.factor` contient une série de `if ... else` qui servent à sélectionner la fonction graphique à appeler en fonction des caractéristiques de l'objet assigné au deuxième argument, soit `y`.

```
graphics:::plot.factor
```

```
## function (x, y, legend.text = NULL, ...)
## {
##     if (missing(y) || is.factor(y)) {
##         dargs <- list(...)
##         axisnames <- if (!is.null(dargs$axes))
##             dargs$axes
##         else if (!is.null(dargs$xaxt))
##             dargs$xaxt != "n"
##         else TRUE
##     }
##     if (missing(y)) {
##         barplot(table(x), axisnames = axisnames, ...)
##     }
##     else if (is.factor(y)) {
```

```
##         if (is.null(legend.text))
##             spineplot(x, y, ...)
##         else {
##             args <- c(list(x = x, y = y), list(...))
##             args$yaxlabels <- legend.text
##             do.call("spineplot", args)
##         }
##     }
##     else if (is.numeric(y))
##         boxplot(y ~ x, ...)
##     else NextMethod("plot")
## }
## <bytecode: 0x0000000011abc810>
## <environment: namespace:graphics>
```

Création de classes et de méthodes de type S3

Il est facile de créer de nouvelles méthodes pour des fonctions génériques existantes, telles que `print`, `plot` et `summary`. Par exemple, supposons que nous voulons modifier l'impression de l'objet suivant.

```
valeurs_aleatoires <- rnorm(7)
print(valeurs_aleatoires)
```

```
## [1]  0.4319268  0.4780331 -0.6331304 -0.0511430 -0.1312676 -1.1149903  0.1742424
```

Cet objet a la classe implicite suivante.

```
class(valeurs_aleatoires)
```

```
## [1] "numeric"
```

Est-ce qu'une méthode de la fonction générique `print` existe pour cette classe ?

```
methods(class = "numeric")
```

```
## [1] all.equal     as.data.frame as.Date       as.POSIXct    as.POSIXlt
## [6] as.raster     coerce        Ops
## see '?methods' for accessing help and source code
```

```
any(methods(class = "numeric") == "print")
```

```
## [1] FALSE
```

Non. Ainsi, l'impression produite par la commande `print(valeurs_aleatoires)` provient d'un appel à la méthode `print.default`.

Pour modifier l'impression de l'objet `valeurs_aleatoires`, nous pourrions créer une méthode `print` pour la classe `"numeric"`. Cette façon de faire fonctionnerait, mais est-ce une bonne idée ?

Non. Procéder de cette façon aurait un désavantage majeur : l'impression de tous les objets de classe `"numeric"` serait modifiée. Nous ne souhaitons pas changer le comportement général de R avec les objets de classe `"numeric"`. Nous souhaitons changer l'impression d'un seul objet : `valeurs_aleatoires`.

Il est donc préférable d'assigner une toute nouvelle classe à l'objet `valeurs_aleatoires`, puis d'écrire une méthode `print` pour cette classe.

Assignation d'une classe à un objet

Pour assigner une classe à un objet R, il suffit de l'encadrer d'un appel à la fonction `class`, suivi d'un opérateur d'assignation et du nom de la classe sous forme de chaîne de caractère. Un nom de classe ne doit pas comporter d'espaces. Voici un exemple avec l'objet `valeurs_aleatoires`.

```r
class(valeurs_aleatoires) <- "mon_vecteur"
print(valeurs_aleatoires)
```

```
## [1]  0.4319268  0.4780331 -0.6331304 -0.0511430 -0.1312676 -1.1149903  0.1742424
## attr(,"class")
## [1] "mon_vecteur"
```

Conception d'une méthode pour une classe

Les méthodes sont des fonctions. Alors créer une méthode signifie simplement créer une fonction. C'est le nom d'une fonction qui détermine si elle est une simple fonction ou une méthode associée à une fonction générique.

Une fonction dont le nom est de la forme `nom_fonction_generique.nom_classe` est une méthode associée à la fonction générique nommée `nom_fonction_generique`, pour la classe nommée `nom_classe`. Il faut évidemment que `nom_fonction_generique` soit le nom d'une fonction générique. La fonction `data.frame` n'est pas une méthode associée à la fonction générique `data` pour la classe `frame`. Alors la simple présence d'un point dans le nom d'une fonction n'implique pas nécessairement qu'il s'agisse d'une méthode de type S3.

Une méthode comporte typiquement les arguments suivants :

- même premier argument que la fonction générique (portant idéalement le même nom)
- autres arguments de la fonction générique, au besoin ;
- arguments supplémentaires, au besoin ;
- l'argument `...` (même s'il n'est pas utilisé).

Il est important que le premier argument de la méthode concorde avec le premier argument de la fonction générique, car lorsque la fonction générique appellera la méthode, elle assignera comme valeur à son premier argument ce qu'elle a reçu comme valeur pour son propre premier argument.

L'ordre des autres arguments n'est cependant pas tellement important. Par exemple, la fonction générique `aggregate` possède 2 arguments : `x` et `...` :

```r
aggregate
```

```
## function (x, ...)
## UseMethod("aggregate")
## <bytecode: 0x000000000ef73358>
## <environment: namespace:stats>
```

Il existe 4 méthodes pour cette fonction générique dans l'installation de base de R :

```r
methods(aggregate)
```

```
## [1] aggregate.data.frame aggregate.default*   aggregate.formula*   aggregate.ts
## see '?methods' for accessing help and source code
```

Ces méthodes possèdent les arguments suivants :

```r
args(aggregate.data.frame)
```

```
## function (x, by, FUN, ..., simplify = TRUE, drop = TRUE)
## NULL
```

```r
args(getS3method("aggregate", "default"))
```

```
## function (x, ...)
## NULL
```

```r
args(getS3method("aggregate", "formula"))
```

```
## function (formula, data, FUN, ..., subset, na.action = na.omit)
```

```
## NULL
```

```
args(aggregate.ts)
```

```
## function (x, nfrequency = 1, FUN = sum, ndeltat = 1, ts.eps = getOption("ts.eps"),
##     ...)
## NULL
```

L'argument `...` est toujours présent, mais il n'est pas toujours placé en dernier. La méthode `aggregate.default` ne possède pas d'arguments supplémentaires, contrairement aux autres méthodes. De plus, une de ces méthodes (`aggregate.formula`) ne possède même pas d'argument nommé `x`.

Exemple avec la méthode générique `print`

Créons une méthode `print` pour la classe `mon_vecteur`.

```
print.mon_vecteur <- function(x, ..., intro = TRUE) {
  if (intro) {
    cat("Voici le vecteur :\n")
  }
  print(unclass(x), ...)
  invisible(x)
}
```

Voyons l'impression produit par notre nouvelle méthode.

```
print(valeurs_aleatoires)
```

```
## Voici le vecteur :
## [1]  0.4319268  0.4780331 -0.6331304 -0.0511430 -0.1312676 -1.1149903  0.1742424
```

```
print(valeurs_aleatoires, intro = FALSE)
```

```
## [1]  0.4319268  0.4780331 -0.6331304 -0.0511430 -0.1312676 -1.1149903  0.1742424
```

Utilité de l'appel à la fonction `unclass`

Vous remarquerez que le corps de la méthode `print.mon_vecteur` comporte un appel à la fonction `unclass` dans l'instruction `print(unclass(x), ...)`. L'appel à `unclass` est important. Sans lui, une boucle sans fin serait exécutée lors de l'appel de la fonction `print` en lui donnant un objet de classe `mon_vecteur` en entrée : la fonction générique `print` appellerait la méthode `print.mon_vecteur`, qui appellerait la fonction générique `print`, qui appellerait la méthode `print.mon_vecteur` et ainsi de suite.

L'appel à la fonction `unclass` dans `print(unclass(x), ...)` permet de fournir en entrée à `print` l'objet `x` sans sont attribut `"class"`. Ainsi, `print` appellera la méthode correspondant à la classe implicite de `x`.

Impact du positionnement de l'argument `...`

Dans cet exemple, nous avons placé l'argument `...` avant les arguments supplémentaires dans la déclaration de la méthode. Dans le corps de la fonction, nous avons passé l'argument `...` à un appel à la fonction `print`. En conséquence, si nous appelons `print.mon_vecteur` (via `print` ou non) en passant une deuxième valeur sans l'assigner à un nom d'argument, cette valeur sera automatiquement assignée au deuxième argument de la méthode `print` appelée et non à l'argument `intro` de la méthode `print.mon_vecteur`.

Voici un exemple.

```
print(valeurs_aleatoires, FALSE)
```

```
## Voici le vecteur :
## [1]  0  0 -1 -0 -0 -1  0
```

Ici, `print` appelle sa méthode `print.mon_vecteur`, qui appelle de nouveau la méthode `print`, qui choisit d'appeler sa méthode `print.default` (car `class(unclass(valeurs_aleatoires)) == "numeric"` et `print.numeric` n'existe pas). Le deuxième argument de `print.default` est `digits`. Il reçoit la valeur `FALSE`, qui est alors transformée en la valeur numérique 0 et les nombres affichés sont arrondis à l'unité près.

Utilité de l'appel à la fonction `invisible`

Il est courant pour une méthode `print` de terminer par la commande `invisible(x)`. La fonction `invisible` provoque le retour d'une valeur par la fonction, tout comme la fonction `return`. Cependant, contrairement à `return`, `invisible` ne provoque pas une impression lorsque l'appel de la fonction n'est pas assigné à un nom, évitant ainsi une impression double suite à une commande comme la suivante :

```
print(valeurs_aleatoires)
```

```
## Voici le vecteur :
## [1]  0.4319268  0.4780331 -0.6331304 -0.0511430 -0.1312676 -1.1149903  0.1742424
```

Si `invisible` était remplacé par `return` dans le corps de la méthode `print.mon_vecteur` comme suit

```
print.mon_vecteur <- function(x, ..., intro = TRUE) {
  if (intro) {
    cat("Voici le vecteur :\n")
  }
  print(unclass(x), ...)
  return(x)
}
```

nous obtiendrions plutôt :

```
print(valeurs_aleatoires)
```

```
## Voici le vecteur :
## [1]  0.4319268  0.4780331 -0.6331304 -0.0511430 -0.1312676 -1.1149903  0.1742424
```

```
## Voici le vecteur :
## [1]  0.4319268  0.4780331 -0.6331304 -0.0511430 -0.1312676 -1.1149903  0.1742424
```

Une autre option serait de ne rien faire retourner à la méthode, mais le R code team juge qu'il est plus pratique qu'une méthode `print` retourne silencieusement une copie de l'objet qu'elle imprime.

Exemple avec la méthode générique `plot`

Tentons maintenant de créer une méthode `plot` pour un objet de la classe `mon_vecteur`. La fonction générique `plot` possède deux arguments, mais notre méthode reprendra uniquement son premier argument, auquel un argument supplémentaire sera ajouté. Elle générera un histogramme.

```
plot.mon_vecteur <- function(x, main = "Distribution de x", ...) {
  par.default <- par(mar = c(5, 5, 4, 1) + 0.1)
  on.exit(par(par.default), add = TRUE)
  out <- hist(x = x, main = main, ylab = "fréquence")
  invisible(out)
}
```

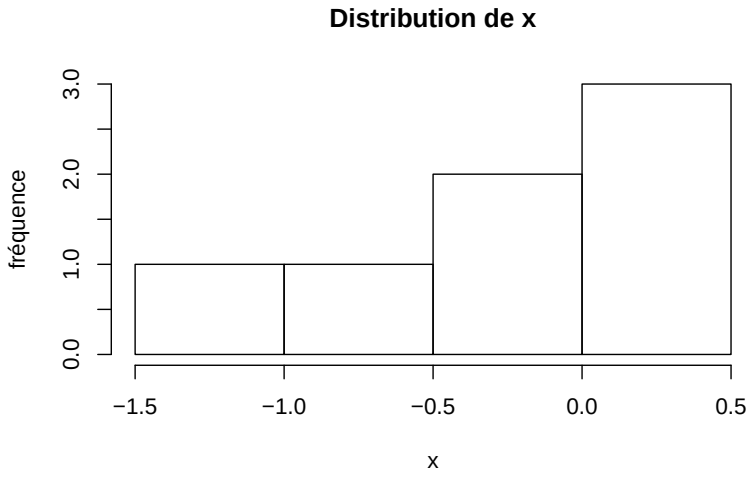
Nous avons maintenant défini deux méthodes pour la classe `mon_vecteur`.

```
methods(class = "mon_vecteur")
```

```
## [1] plot  print
## see '?methods' for accessing help and source code
```

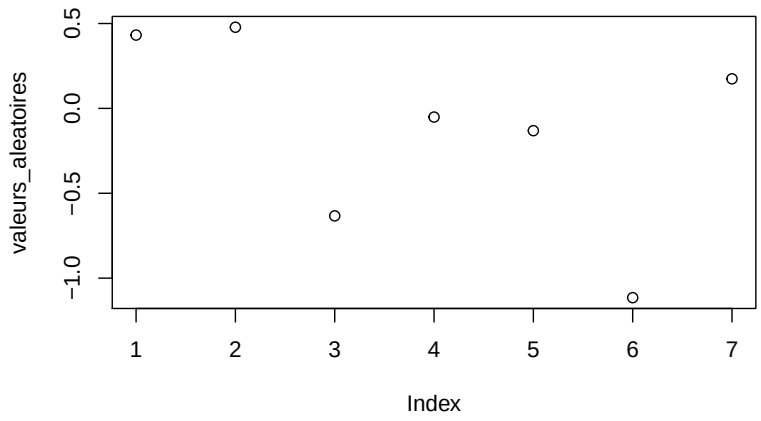
La méthode pour la fonction générique `plot` produit le résultat suivant.

```
plot(valeurs_aleatoires)
```



Si `valeurs_aleatoires` n'était pas de classe `mon_vecteur`, nous obtiendrions plutôt ce graphique.

```
valeurs_aleatoires <- unclass(valeurs_aleatoires)
plot(valeurs_aleatoires)
```



Cet exemple illustre le fait qu'une méthode ne doit pas obligatoirement reprendre tous les arguments de la fonction générique associée.

Utilité de l'appel à la fonction `on.exit`

La fonction `on.exit` permet de s'assurer qu'une instruction est évaluée avant de terminer l'exécution d'un appel à une fonction, et ce, même si une erreur provoque l'arrêt de cette exécution avant la fin du corps de la fonction. Dans l'exemple de la méthode `plot.mon_vecteur`, le paramètre graphique `mar` (largeur des marges internes) a été modifié par un appel à la fonction `par`. Nous savons qu'une bonne pratique est de remettre les paramètres graphiques modifiés avec `par` à leurs valeurs par défaut à la fin d'un code de production d'un graphique. Lorsque ce code est dans le corps d'une fonction, il est plus prudent de programmer la remise aux valeurs par défaut par un appel à la fonction `on.exit` placé juste après l'appel à la fonction `par`.

De plus, spécifier l'argument `add = TRUE` dans l'appel à la fonction `on.exit` est recommandé afin de permettre le cumul des instructions de fin d'exécution. Notons qu'avec un seul appel à `on.exit` dans le corps d'une fonction, c'est argument n'a cependant aucun impact.

Méthode `print` pour formater l'impression des sorties de nos fonctions

Il est souvent utile de créer des méthodes `print` pour formater l'impression des sorties de nos fonctions. Pour ce faire, il suffit de compléter les deux étapes suivantes.

1. D'abord, dans le corps de la fonction, il **faut attribuer une nouvelle classe**, avec la fonction `class`, à l'objet retourné en sortie.

Il est pratique courante en R d'utiliser comme nom de la classe de l'objet retourné en sortie d'une fonction le nom de la fonction. C'est ce que fait par exemple la fonction `lm`. Ce n'est cependant pas obligatoire de reprendre le nom de la fonction. Le nom de la classe peut être quelconque, pourvu qu'il ne soit pas déjà utilisé par d'autres fonctions dans un package chargé afin d'éviter de modifier l'impression d'autres objets.

2. Il faut ensuite créer une fonction nommée : `print.nom_classe`.

Pour illustrer ces étapes, créons une méthode `print` pour un objet retourné par la fonction `stats_desc` créée dans les notes de cours sur les fonctions. Tout d'abord, attribuons une nouvelle classe à la sortie de `stats_desc`.

```
stats_desc <- function(x, format_sortie = c("vecteur", "matrice", "liste"), ...) {
  # Calcul
  if (is.numeric(x)) {
    stats <- c(min = min(x, ...), moy = mean(x, ...), max = max(x, ...))
  } else if (is.character(x) || is.factor(x)) {
    stats <- table(x)
  } else {
    stats <- NA
  }
  # Production de la sortie
  format_sortie <- match.arg(format_sortie)
  if (format_sortie == "matrice") {
    stats <- as.matrix(stats)
    colnames(stats) <- if (is.character(x) || is.factor(x)) "frequence" else "stat"
  } else if (format_sortie == "liste") {
    stats <- as.list(stats)
  }
  out <- list(stats = stats)
  class(out) <- "stats_desc"
  out
}
```

Dans cet exemple, en plus de l'instruction `class(out) <- "stats_desc"` ajoutée pour attribuer une classe à la sortie de la fonction, l'objet retourné en sortie a été formaté en liste contenant tout ce qu'il y a à retourner (ici un seul objet). Il n'est pas obligatoire qu'une sortie de fonction qui possède une classe soit une liste, mais, encore là, c'est une pratique très courante.

Notons que les instructions

```
  out <- list(stats = stats)
  class(out) <- "stats_desc"
  out
```

auraient pu être remplacées par

```
  structure(list(stats = stats),  class = "stats_desc")
```

Maintenant, écrivons le code de notre nouvelle méthode `print`, pour un objet de classe `stats_desc`.

```
print.stats_desc <- function(x, ...) {
  cat("Statistiques descriptives :\n")
  cat("**************************\n")
  print(x$stats, ...)
  cat("**************************\n")
  invisible(x)
}
```

Si la méthode `print.stats_desc` est appelée par `print`, c'est que l'objet donné comme premier argument à `print` est assurément de classe `stats_desc`. Ainsi, dans le corps de la méthode `print.stats_desc`, nous pouvons exploiter les caractéristiques des objets de cette classe. Nous savons qu'un objet de cette classe est une liste comportant un élément nommé `stats`. L'instruction `print(x$stats, ...)` exploite cette caractéristique.

Mais pourquoi cette instruction ne comporte-t-elle pas d'appel à la fonction `unclass` comme dans le corps de la méthode `print.mon_vecteur` ? Cette instruction ne provoquera-t-elle pas un appel en boucle à la méthode `print.stats_desc` ?

Non, parce que dans le corps de la méthode `print.stats_desc` l'objet `x` possède la classe `print.stats_desc`, mais pas le sous-objet `stats` dans `x`.

Pour une fonction qui retourne une très longue liste, attribuer une classe à sa sortie et écrire une méthode `print` pour cette classe permet d'éviter l'impression dans la console de la liste entière retournée en sortie.

Voici un exemple d'impression produit par la méthode `print.stats_desc`.

```
stats_desc(x = iris$Species, format_sortie = "matrice")
```

```
## Statistiques descriptives :
## **************************
##            frequence
## setosa            50
## versicolor        50
## virginica         50
## **************************
```

Système S4

Manipulation d'objets dont la classe est de type S4

Même si nous n'illustrons pas ici comment créer des classes de type S4, il est bon de savoir comment utiliser ce type de classes qui est assez courant, particulièrement dans les packages distribués sur Bioconductor. Ces classes sont utilisables en R grâce au package `methods`, inclus dans l'installation de base.

Pour illustrer les classes de type S4, installons le package `sp`, qui exploite ce type de classe.

```
install.packages("sp")
```

Voici un exemple d'utilisation d'une fonction de ce package, tiré d'une fiche d'aide du package.

```
library(sp)
x = c(1, 2, 3, 4, 5)
y = c(3, 2, 5, 1, 4)
S <- SpatialPoints(cbind(x, y))
S
```

```
## SpatialPoints:
##      x y
## [1,] 1 3
## [2,] 2 2
## [3,] 3 5
## [4,] 4 1
## [5,] 5 4
## Coordinate Reference System (CRS) arguments: NA
```

```
str(S)
```

```
## Formal class 'SpatialPoints' [package "sp"] with 3 slots
##   ..@ coords     : num [1:5, 1:2] 1 2 3 4 5 3 2 5 1 4
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "x" "y"
##   ..@ bbox       : num [1:2, 1:2] 1 1 5 5
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:2] "x" "y"
##   .. .. ..$ : chr [1:2] "min" "max"
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr NA
```

L'objet retourné par la fonction `SpatialPoints` n'est pas une liste. C'est un objet appartenant à une classe de type S4, définie dans le package `sp`.

```
isS4(S)
```

```
## [1] TRUE
```

```
class(S)
```

```
## [1] "SpatialPoints"
## attr(,"package")
## [1] "sp"
```

Pour atteindre les éléments dans l'objet, il est possible d'utiliser une méthode conçue à cet effet. Par exemple, la fiche d'aide ouverte par la commande `help("SpatialPoints-class")` nous informe qu'une méthode `coordinates` est définie pour les objets de la classe `"SpatialPoints"`. Nous pouvons aussi énumérer toutes les fonctions génériques possédant une méthode définie pour une certaine classe de type S4 avec la fonction `methods` comme suit.

```
methods(class = "SpatialPoints")
```

```
##  [1] $             $<-           [             [[            [[<-          [<-
##  [7] addAttrToGeom as.data.frame bbox          coerce        coordinates   coordinates<-
## [13] coordnames    coordnames<-  dimensions    fullgrid      geometry      geometry<-
## [19] gridded       gridded<-     is.projected  length        merge         over
## [25] plot          points        polygons      print         proj4string   proj4string<-
## [31] rbind         row.names     row.names<-   show          spChFIDs<-    split
## [37] sppanel       spsample      spTransform   summary
## see '?methods' for accessing help and source code
```

En fait, des méthodes de types S3 et S4 peuvent être définies pour des objets dont la classe est de type S4. La fonction `methods` retourne les méthodes des deux types. Pour se limiter à un seul type, il faut utiliser les fonctions `.S3methods` et `.S4methods`.

```
.S3methods(class = "SpatialPoints")
```

```
## [1] as.data.frame length        points        print         rbind         row.names
## [7] row.names<-   split
## see '?methods' for accessing help and source code
```

```
.S4methods(class = "SpatialPoints")
```

```
##  [1] $             $<-           [             [[            [[<-          [<-
##  [7] addAttrToGeom bbox          coerce        coordinates   coordinates<- coordnames
## [13] coordnames<-  dimensions    fullgrid      geometry      geometry<-    gridded
## [19] gridded<-     is.projected  merge         over          plot          polygons
## [25] proj4string   proj4string<- show          spChFIDs<-    sppanel       spsample
## [31] spTransform   summary
## see '?methods' for accessing help and source code
```

Une fonction générique dans le système S4 n'a pas la même allure que dans le système S3.

```
coordinates
```

```
## standardGeneric for "coordinates" defined from package "sp"
##
## function (obj, ...)
## standardGeneric("coordinates")
## <bytecode: 0x0000000015870c40>
## <environment: 0x0000000015842dd8>
## Methods may be defined for arguments: obj
## Use  showMethods("coordinates")  for currently available ones.
```

La méthode `coordinates` pour un objet de classe `"SpatialPoints"` extrait l'élément de l'objet `S` nommé `coords`.

```
coordinates(S)
```

```
##      x y
## [1,] 1 3
## [2,] 2 2
## [3,] 3 5
## [4,] 4 1
## [5,] 5 4
```

Nous pouvons accéder à la définition de cette méthode grâce à la fonction `getMethod` comme suit.

```
getMethod(coordinates, signature = "SpatialPoints")
```

```
## Method Definition:
##
## function (obj, ...)
## {
##     .local <- function (obj)
##     obj@coords
##     .local(obj, ...)
## }
## <bytecode: 0x000000001587f728>
## <environment: namespace:sp>
##
## Signatures:
##         obj
## target  "SpatialPoints"
## defined "SpatialPoints"
```

Cette définition n'est pas aussi simple que celle d'une méthode de type S3. Je ne vais pas l'approfondir ici.

Pour extraire des éléments d'un objet dont la classe est de type S4, il est aussi possible d'utiliser l'opérateur `@` (et non `$` puisqu'il ne s'agit pas d'une liste).

```
S@coords
```

```
##      x y
## [1,] 1 3
## [2,] 2 2
## [3,] 3 5
## [4,] 4 1
## [5,] 5 4
```

ou encore la fonction `slot`.

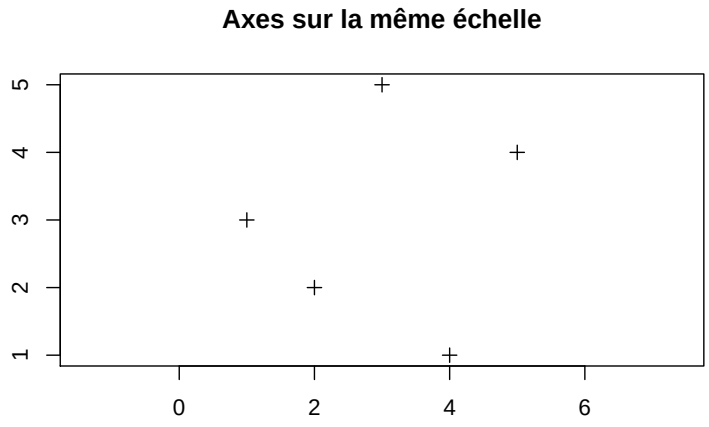
```
slot(S, "coords")
```

```
##      x y
## [1,] 1 3
## [2,] 2 2
## [3,] 3 5
## [4,] 4 1
## [5,] 5 4
```

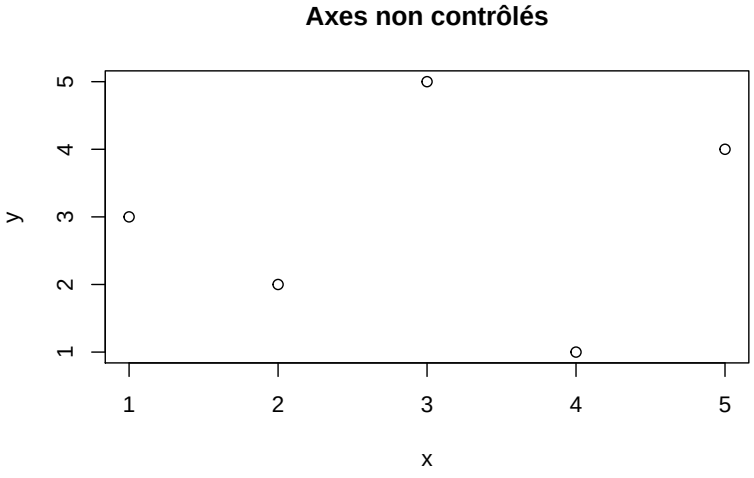
Ainsi, utiliser des classes de type S4 est simple. Il suffit de d'abord bien identifier que nous manipulons un objet dont la classe est de type S4. La mention `Formal class` dans la sortie de `str` nous l'indique. La fonction `isS4` peut aussi nous le confirmer. Ensuite, nous pouvons manipuler l'objet avec les fonctions génériques possédant des méthodes pour sa classe et nous pouvons extraire des éléments de l'objet avec l'opérateur `@` ou la fonction `slot`.

Notons qu'un des intérêts du package `sp` est la production facilitée de graphiques représentant des données spatiales, par exemple des coordonnées géographiques.

```
plot(S, main = "Axes sur la même échelle", axes=TRUE)
```




```
plot(x, y, main = "Axes non contrôlés")
```



Résumé

Systèmes de programmation orientée objet en R :

• **S3** : le plus ancien, le moins formel, mais encore le plus utilisé.

• **S4** : semblable au S3, mais plus formel ; la norme sur Bioconductor

Les systèmes S3 et S4 se basent tous deux sur :

- des classes attribuées aux objets,
- des fonctions génériques (ex. `print`, `plot`, `summary`, etc.),
- des méthodes = différentes versions d'une fonction générique, associées à des classes spécifiques.
  - la fonction `methods` retourne les noms des :
    - méthodes existantes pour une fonction générique donnée ou
    - fonctions génériques possédant une méthode pour une classe donnée.

• **Autres** : **RC** (Reference Classes ou R5), **R6**, etc.
→ pas encore beaucoup utilisé, pas couvert dans le cours.

Fonctionnement du système S3

- fonction générique de type S3 : appelle la fonction `UseMethod`
- méthode de type S3 = fonction dont le nom a la forme `nom_fonction_generique.nom_classe`

C'est la valeur retournée par la fonction `class` sur l'objet fourni comme premier argument en entrée à une fonction générique qui détermine quelle méthode est utilisée.

- `class` retourne l'attribut `class` d'un objet, s'il existe (*il peut être de longueur supérieure à 1*).
- Sinon `class` retourne la classe implicite de l'objet (ex. `"list"`, `"matrix"`, `"numeric"` `"character"`).
- La méthode utilisée est celle associée à la première classe, sinon la deuxième, etc.
- Si aucune méthode n'existe pour les classes de l'objet :
  - la méthode par défaut est utilisée, si elle existe ;
  - sinon une erreur est retournée.

Création de classes et de méthodes de type S3

Création de classe de type S3 = ajout d'un attribut `class` à un objet

```r
class(objet) <- "nom_classe"
```

Pratique courante = attribuer une classe à l'objet retourné par une de nos fonctions (permet notamment de contrôler l'impression de la sortie grâce à une méthode `print`).

Création de méthode de type S3 = création de fonctions avec bon nom

```r
nom_fonction_generique.nom_classe <- function(x, ...) {
  # corps de la méthode
}
```

où `nom_classe` peut être le nom d'une classe existante ou celui d'une nouvelle classe que nous avons créée.

Une méthode comporte typiquement les arguments suivants :

- même premier argument que la fonction générique (portant idéalement le même nom)
- autres arguments de la fonction générique, au besoin ;
- arguments supplémentaires, au besoin ;
- l'argument `...` (même s'il n'est pas utilisé).

Fonctions utiles :

- `unclass` : pour retirer l'argument `"class"` d'un objet ;
- `invisible` : remplacement de `return` pour faire retourner un objet à une fonction sans provoquer l'impression de l'objet lorsque la fonction est appelée sans assignation (typiquement utilisé dans les méthodes pour les fonctions génériques `print` et `plot`) ;
- `on.exit` : pour s'assurer qu'une instruction est évaluée avant de terminer l'exécution d'un appel à une fonction (p. ex. utile pour remettre à leurs valeurs par défaut des paramètres graphiques).

Manipulation d'objets dont la classe est de type S4

- Caractéristiques des **objets possédant une classe de type S4** :
  - possèdent un attribut `class` ;
  - `isS4(objet)` retourne `TRUE` ;
  - la sortie de `str(objet)` contient « `Formal class` » ;
  - contiennent des éléments (*slots*) qui peuvent être extraits :
    - avec des méthodes conçues à cet effet,
    - avec l'opérateur `@`,
    - avec la fonction `slot`.
- Caractéristiques des **fonctions génériques de type S4** :
  - appellent la fonction `standardGeneric`
- Caractéristiques des **méthodes de type S4** :
  - code source affiché avec `getMethod(nom_fonction_generique, signature = "nom_classe")`
  - l'impression code source contient « `Method Definition` ».

Références

- Matloff, N. (2011). *The Art of R Programming : A Tour of Statistical Software Design*. No Starch Press. Chapitre 9.
- Wickham, H. (2019). *Advanced R, Second Edition*. Chapman and Hall/CRC. Chapitres 12 à 16. URL https://adv-r.hadley.nz/oo.html