Tests et exceptions en R

Sophie Baillargeon, Université Laval

2020-03-16

Table des matières

Ces notes présentent comment tester et déboguer nos fonctions en R. Nous allons également y apprendre à contrôler les messages d'erreur et d'avertissement produits par nos fonctions. Cependant, avant d'entrer dans le vif du sujet, revenons sur les étapes conseillées de développement de fonctions.

Bonnes pratiques dans le développement de fonctions

Rappelons-nous que l'objectif numéro 1 des bonnes pratiques de programmation en R est de développer du code qui produit les résultats escomptés. Voici quelques conseils pour atteindre cet objectif.

Démarche de travail recommandée

Les étapes de développement de fonction conseillées dans les notes sur les fonctions R peuvent être complétées comme suit :

1. **Planifier** le travail (pas de programmation encore) :
   - définir clairement la tâche à accomplir par la fonction et la sortie qu'elle doit produire,

- prévoir les étapes à suivre afin d'effectuer cette tâche,
- identifier les arguments devant être fournis en entrée à la fonction.

2. **Développer le corps de la fonction**
   - 2.1. Écrire le programme par étapes, d'abord sans former la fonction, en commentant bien le code et en travaillant sur des mini-données test.
   - 2.2 Pour chaque petite étape ou sous-tâche, tester interactivement si le programme produit le résultat escompté (tester souvent en cours de travail, ainsi il y a moins de débogage à faire).
3. **Créer la fonction** à partir du programme développé.
4. **Documenter** la fonction.
5. **Tester** la fonction : sauvegarder nos tests et bien les structurer, car ils serviront souvent.
6. Si nous rencontrons des comportements indésirables lors des tests, **déboguer** la fonction :
   - cerner le ou les problèmes,
   - apporter les correctifs nécessaires à la fonction (que ce soit dans son corps ou dans liste de ses arguments),
   - adapter la documentation et les tests au besoin,
   - rouler de nouveau les tests,
   - répéter ces sous-étapes jusqu'à ce que les tests ne révèlent plus aucun problème à régler ou aucune amélioration à apporter.

Les étapes 5 et 6 ont été ajoutées à celles déjà présentées. L'étape 5 sert à s'assurer de rencontrer l'objectif de produire les résultats escomptés. L'étape 6, celle du débogage, est nécessaire lorsque quelque chose cloche dans le comportement de la fonction. Les sous-étapes du débogage nous mène, espérons le, à régler les anomalies.

La démarche de travail recommandée ici a aussi pour but d'aider à atteindre le deuxième objectif des bonnes pratiques : développer du code facile à maintenir. Au fil du temps, il n'est pas rare d'avoir besoin de modifier une fonction que nous avons créée. La modification peut avoir pour but de :

- ajouter une fonctionnalité,
- corriger un bogue découvert par un utilisateur,
- rendre la fonction plus facile d'utilisation,
- rendre la fonction plus rapide,
- etc.

Quand vient le temps de modifier une fonction, notre travail est facilité si celle-ci a été bien documentée et testée. Avant d'apporter des changements au code, il est encore recommandé de bien planifier le travail (donc de réfléchir avant de programmer). Après avoir modifié la fonction, il faut mettre à jour la documentation et les tests au besoin. Exécuter ces tests de nouveau nous permet de nous assurer que les modifications apportées n'ont pas altéré des comportements de la fonction qui ne doivent pas changer.

Exemple de développement de fonction

Créons ensemble une fonction qui calcule la distance de Manhattan entre deux points.

Planification (étape 1) :

- implanter la formule (simple, une seule étape) : http://fr.wikipedia.org/wiki/Distance_de_Manhattan
- sortie = la distance (une seule valeur)
- entrée = les coordonnées de deux points (2 arguments)

Développement du corps de la fonction (étape 2) :

```
# mini-données test
pt1 <- c(0,0)
pt2 <- c(1,1)

# Code le plus simple qui me vient en tête
abs(pt1[1] - pt2[1]) + abs(pt1[2] - pt2[2])
```

```
## [1] 2
```

```r
# Il faudrait que ça fonctionne peu importe la dimension de l'espace dans lequel mes
# points sont représentés (donc peu importe la longueur des vecteurs pt1 et pt2)
sum(abs(pt1 - pt2))
```

```
## [1] 2
```

Création de la fonction à partir du programme développé (étape 3) :

```r
dist_manhattan <- function(point1, point2) {
  sum(abs(point1 - point2))
}
```

Documentation de la fonction (étape 4) :

Les informations minimales à fournir sont :

- ce que la fonction fait,
- quels arguments la fonction accepte en entrée,
- ce que la fonction retourne en sortie.

Le plus simple est de fournir des informations en commentaires avant la définition de la fonction ou au début du corps de celle-ci.

```r
# Calcule la distance de Manhattan entre deux points
# Arguments :
# - point1 : Un vecteur numerique des coordonnees du premier point.
# - point2 : Un vecteur numerique des coordonnees du deuxieme point.
# Sortie : La distance de Manhattan entre pt1 et pt2 (une seule valeur).
dist_manhattan <- function(point1, point2) {
  sum(abs(point1 - point2))
}
```

Nous allons revenir sur ce point dans le cours sur la création de packages, car toute fonction d'un package doit avoir une fiche d'aide. Nous verrons donc une façon plus formelle de documenter des fonctions.

Test et débogage (étapes 5 et 6) :

Nous allons faire les tests et le débogage après avoir vu la théorie à ce sujet dans les sections suivantes.

Organisation du code

Une bonne pratique dans l'organisation de notre code est de placer la définition de nos fonctions dans un fichier distinct. Ainsi, les instructions contenant des appels à nos fonctions ne sont pas dans le même programme R que les définitions des fonctions.

Afin de pouvoir utiliser nos fonctions, elles doivent être présentes dans un des environnements du chemin de recherche de R. Nous pourrions les mettre dans un package que nous créons et charger ce package. Plus simplement, nous pourrions soumettre le code définissant les fonctions dans la console R afin de créer les fonctions dans notre environnement de travail. C'est la façon de faire utilisée dans le cours jusqu'à maintenant.

Si nous avons placé les définitions de fonctions dans un fichier à part, il est facile de soumettre d'un coup tout le code contenu dans le fichier en une seule commande : un appel à la fonction `source`. Par exemple, si nos fonctions sont définies dans le fichier `mes_fonctions.R` du répertoire `C:\coursR`, la commande suivante :

```r
source("C:/coursR/mes_fonctions.R")
```

évalue toutes les instructions contenues dans `mes_fonctions.R`. Les objets créés par ces instructions sont stockés par défaut dans l'environnement de travail.

Appeler la fonction `source` est donc similaire à sélectionner tout le code contenu dans un fichier et le soumettre dans la console. Cependant, les deux façons de faire ne sont pas identiques. Avec `source`, dès que le code dans le fichier comporte au moins une erreur de syntaxe, aucune ligne de code du fichier n'est soumise. Aussi, seuls les appels spécifiques à la fonction `print` provoquent des impressions, alors qu'une instruction contenant seulement le nom d'un objet ne génère aucune impression. Mais la plus grande différence entre les deux approches est que soumettre une commande `source` est plus efficace en terme de temps de travail que de sélectionner des lignes de code dans un script R, puis de soumettre toutes ces lignes. Avec `source`, le script R contenant les définitions des fonctions n'a même pas besoin d'être ouvert.

Ainsi, pour compléter la bonne pratique de placer la définition de nos fonctions dans un fichier distinct, il est recommandé d'inclure en entête de tout programme R utilisant les fonctions définies un certain fichier un appel à la fonction `source` pour soumettre les instructions contenues dans le fichier en question.

Tests

Tester ses fonctions consiste à appeler les fonctions en donnant en entrée des valeurs d'arguments pour lesquelles nous savons quel résultat devrait être obtenu.

- Nous pouvons faire ça avec des mini-exemples pour lesquels nous pouvons faire les calculs à la main pour trouver le résultat escompté. Il est bien que ces cas soient représentatifs (en plus simple) de diverses situations qui peuvent être rencontrées en pratique.
- Si des fonctions qui font le même calcul existent déjà, il est bon de comparer les résultats de nos fonctions aux résultats de ces fonctions.
- Nous pouvons aussi comparer les résultats de nos fonctions à des résultats publiés dans des articles scientifiques ou des résultats théoriques. Si nos fonctions proposent de nouvelles méthodes de calculs, nous ne nous attendons pas nécessairement à reproduire de façon exacte les résultats publiés, mais nos résultats devraient être cohérents avec ceux publiés.

Les objectifs sont d'obtenir les résultats escomptés, mais aussi de générer des erreurs et avertissement en temps opportun.

Objectif 1 : obtenir le résultat escompté

Afin de vérifier si une fonction retourne le résultat escompté, il faut la tester dans toutes sortes de situations.

Exemple :

Testons la fonction `dist_manhattan` avec d'autres points que ceux utilisés pour développer la fonction.

- Points avec coordonnées négatives :

```
dist_manhattan(point1 = c(0, -5), point2 = c(0, -15))
```

```
## [1] 10
```

Résultat attendu selon un calcul à la main : 10 = résultat obtenu.

- Points de dimension supérieure à 2 :

```
dist_manhattan(point1 = c(0,0,0,0,0), point2 = c(1,1,1,1,1))
```

```
## [1] 5
```

Résultat attendu selon un calcul à la main : 5 = résultat obtenu.

- Comparaisons avec la fonction `dist` qui implémente le même calcul que notre fonction `dist_manhattan` :

```r
dist_manhattan(point1 = c(0,0), point2 = c(1,1))
```

```
## [1] 2
```

```r
dist(rbind(c(0,0), c(1,1)), method = "manhattan")
```

```
##   1
## 2 2
```

```r
dist_manhattan(c(0,0), c(1,1)) == dist(rbind(c(0,0), c(1,1)), method = "manhattan")[1]
```

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

```r
dist_manhattan(point1 = c(0,-5), point2 = c(0,-15))
```

```
## [1] 10
```

```r
dist(rbind(c(0,-5), c(0,-15)), method = "manhattan")
```

```
##    1
## 2 10
```

```r
dist_manhattan(c(0,-5), c(0,-15)) == dist(rbind(c(0,-5), c(0,-15)), method = "manhattan")[1]
```

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

```r
dist_manhattan(point1 = c(0,0,0,0,0), point2 = c(1,1,1,1,1))
```

```
## [1] 5
```

```r
dist(rbind(c(0,0,0,0,0), c(1,1,1,1,1)), method = "manhattan")
```

```
##   1
## 2 5
```

```r
dist_manhattan(c(0,0,0,0,0), c(1,1,1,1,1)) ==
  dist(rbind(c(0,0,0,0,0), c(1,1,1,1,1)), method = "manhattan")[1]
```

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

Nous obtenons les mêmes distances.

Si nous n'avions pas obtenu les résultats escomptés, il aurait fallu apporter des correctifs à notre fonction.

Objectif 2 : correctement gérer les exceptions

Les tests visent aussi à vérifier si une fonction réagit correctement aux exceptions. Qu'est-ce qu'une exception ?

> Une **exception** est une situation anormale ou exceptionnelle qui requiert un traitement spécial (souvent l'arrêt de la fonction).

Des exemples d'exceptions sont :

- des arguments incorrects fournis en entrée,
- des résultats de calcul inattendus.

Lors de la rencontre d'exceptions, les fonctions R réagissent en générant des erreurs ou des avertissements.

> Les erreurs et avertissements sont appelés **conditions** en R.

En plus des erreurs et avertissements, R comporte un troisième type de condition : les messages.

Les différents types de conditions sont définis ainsi :

- **Erreur** : L'exécution de la fonction est interrompue et un message d'erreur est affiché.

- **Avertissement** : Un message d'avertissement est affiché pour signifier un problème potentiel. L'exécution de la fonction n'est pas interrompue.
- **Message** : Un message est affiché pour apporter une information supplémentaire (par exemple la valeur par défaut utilisée pour un argument important non fourni en entrée). L'exécution de la fonction n'est pas interrompue. Ce type de condition est moins utilisé que les deux autres.

Notons que les messages associés à une condition R, peu importe son type, sont parfois traduits de façon automatique en fonction de la langue de notre système d'exploitation.

Exemple :

Testons si notre fonction `dist_manhattan` gère correctement quelques exceptions.

- Si nous donnons en entrée à notre fonction `dist_manhattan` des points de dimensions différentes, que se passe-t-il ?

```
dist_manhattan(point1 = c(-1,0), point2 = c(1,2,3))
```

```
## Warning in point1 - point2: longer object length is not a multiple of shorter
## object length
```

```
## [1] 8
```

Nous pourrions préférer que la fonction retourne une erreur plutôt qu'un avertissement. Nous y reviendrons plus loin.

- Si nous donnons en entrée à `dist_manhattan` des arguments non numériques, que se passe-t-il ?

```
dist_manhattan(point1 = c("a", "b"), point2 = c("c","d"))
```

```
## Error in point1 - point2: non-numeric argument to binary operator
```

L'exécution de la fonction s'arrête et le message d'erreur affiché est informatif. Un **message informatif** aide l'utilisateur à comprendre ce qu'il a fait incorrectement. Un message non informatif ne guide pas suffisamment l'utilisateur dans la modification de son appel de la fonction afin de ne plus avoir d'erreur.

- Si nous donnons en entrée des matrices, que se passe-t-il ?

```
dist_manhattan(point1 = rbind(c(0,0), c(1,0)), point2 = rbind(c(3,2), c(2,3)))
```

```
## [1] 9
```

Ce résultat peut être surprenant pour quelqu'un qui pensait obtenir plus d'une distance, par exemple une entre la ligne i de `point1` et la ligne i de `point2` pour tout i = 1, . . . , `nrow(point1)`. Nous pourrions envisager de produire un message d'avertissement (nous verrons comment faire plus loin).

Formaliser des tests avec le package testthat

Le package `testthat` offre des fonctions facilitant l'écriture, l'organisation et l'exécution automatique de tests unitaires en R. Voici quelques fonctions du package :

- fonction d'écriture (un appel à une de ces fonctions = un test unitaire) :
  - `expect_equal`,
  - `expect_error`,
  - `expect_warning`,
  - `expect_true`,
  - etc.
- fonction d'organisation : `context`, `test_that`
- fonction d'exécution : `test_file`, `test_dir`, etc.

L'utilisation de ce package n'est pas décrite en détail ici, mais un court exemple est présenté pour illustrer son utilisation. Écrire ses tests avec `testthat` demande un certain investissement en temps, mais une fois cette étape terminée, il est facile de lancer ses tests à plusieurs reprises en cours de travail.

Exemple :

Voici quelques exemples qui reprennent des tests effectués précédemment sur la fonction `dist_manhattan`.

```
library(testthat)
```

- Cas simple pour lequel nous pouvons calculer à la main le résultat escompté :

```
test_that(
  "nous reproduisons un calcul à la main",
  expect_equal(dist_manhattan(point1 = c(0,-5), point2 = c(0,-15)), 10)
)
```

- Comparaison avec le résultat d'une autre fonction :

```
test_that(
  "nous obtenons le même résultat que la fonction dist",
  expect_equal(
    dist_manhattan(point1 = c(0,0), point2 = c(1,1)),
    as.vector(dist(rbind(c(0,0), c(1,1)), method = "manhattan"))
  )
)
```

- Vérification de la gestion d'exceptions

```
test_that(
  "des vecteurs de dimensions différentes génèrent une erreur",
  expect_error(dist_manhattan(point1 = c(-1,0), point2 = c(1,2,3)))
)
```

```
## Error: Test failed: 'des vecteurs de dimensions différentes génèrent une erreur'
## * `dist_manhattan(point1 = c(-1, 0), point2 = c(1, 2, 3))` did not throw an error.
```

```
test_that(
  "des matrices génèrent un avertissement",
  {
    mat1 <- rbind(c(0,0), c(1,0))
    mat2 <- rbind(c(3,2), c(2,3))
    expect_warning(dist_manhattan(point1 = mat1, point2 = mat2))
  }
)
```

```
## Error: Test failed: 'des matrices génèrent un avertissement'
## * <text>:6: `dist_manhattan(point1 = mat1, point2 = mat2)` did not produce any warnings.
```

Nous allons apporter plus loin des changements à `dist_manhattan` afin de passer avec succès tous ces tests. Pour l'instant, nous passons avec succès les 2 premiers, mais échouons les deux derniers.

Débogage

> Le **débogage** est un processus méthodique pour trouver et régler les bogues dans un programme informatique, soit les anomalies de fonctionnement du programme.

Si nos tests ont révélé des résultats inattendus ou des exceptions mal gérées, un débogage est de mise. La première étape du débogage est de repérer le bout de code responsable du bogue et de comprendre pourquoi le bogue est rencontré. Ensuite, il faut modifier le code pour corriger le problème.

Les outils présentés pour accomplir la première étape du débogage peuvent aussi servir à comprendre une condition obtenue lors de l'utilisation d'une fonction programmée par quelqu'un d'autre.

Lorsque nous obtenons une erreur en appelant une fonction, le message d'erreur explique parfois suffisamment clairement pourquoi la fonction ne peut pas retourner de résultats.

Exemple : argument fourni dans un mauvais format

```
aggregate(x = iris$Sepal.Length, by = iris$Species, FUN = min)
```

```
## Error in aggregate.data.frame(as.data.frame(x), ...): 'by' must be a list
```

Dans cet exemple, le message d'erreur nous aide à comprendre que nous avons mal utilisé la fonction et nous met sur une piste pour modifier notre appel à la fonction.

Dans d'autres cas, les messages d'erreur ou d'avertissement ne sont pas très informatifs. Dans un tel cas, la documentation de la fonction peut parfois nous aider. Une autre option est d'utiliser des outils de débogage pour comprendre la nature de l'exception rencontrée et comment utiliser correctement la fonction.

Rapporter un bogue

Lorsque nous croyons avoir découvert un bogue dans du code que nous n'avons pas développé nous même, il est bien de contacter le mainteneur du code pour lui en faire part. Il faut par contre d'abord s'assurer d'utiliser la dernière version du code. Le mainteneur des fonctions de base de R est le R core team. La page web https://www.r-project.org/bugs.html explique comment faire part de bogues potentiels à cette équipe. Pour signaler un bogue dans un package R, il suffit de contacter son mainteneur par courriel. Toute documentation de package contient l'adresse courriel de son mainteneur. Si le package est développé en utilisant un service web public d'hébergement et de gestion de versions, tel que GitHub, la meilleure façon de rapporter un bogue est de créer un nouvel *issue*.

Faire part d'un bogue potentiel à un mainteneur présente des avantages pour tous. L'utilisateur arrive souvent ainsi à régler le problème qu'il rencontre et le mainteneur a l'opportunité d'améliorer son code en corrigeant des bogues ou en identifiant les aspects moins compris de son code ou de sa documentation.

Exemple de bogue dans notre propre code :

Supposons que nous développons une fonction qui calcule des moyennes. Si l'argument donné en entrée est un vecteur, la fonction doit calculer une seule moyenne, celle des observations dans le vecteur. Si l'argument donné en entrée a plus d'une dimension, la fonction `colMeans` doit être appelée. Pour une matrice en entrée, nous obtiendrions donc la moyenne des observations dans chaque colonne.

```
mean_2 <- function(x) {
  if (is.null(dim(x))) {
    colMeans(x)
  } else {
    mean(x)
  }
}

mean_2(matrix(1:4, 2, 2))
```

```
## [1] 2.5
```

```
mean_2(1:4)
```

```
## Error in colMeans(x): 'x' must be an array of at least two dimensions
```

La fonction ne fait pas ce que nous voulions. Déboguons-la.

Outil 1 : la fonction `traceback`

La première chose à faire en cas d'erreur rencontrée est de tenter de comprendre le message d'erreur affiché. La fonction `traceback` peut apporter plus d'informations concernant la provenance de l'erreur.

```
traceback()
```

```
## 3: stop("'x' must be an array of at least two dimensions")
## 2: colMeans(x) at #3
## 1: mean_2(1:4)
```

Cette fonction retourne la séquence des appels de fonctions qui a mené à l'erreur. Nous apprenons ici que l'erreur a été générée par la fonction `stop`, dans un appel à la fonction `colMeans`, à l'intérieur de l'appel à `mean_2`. En fait, ici, le message d'erreur nous avait déjà informés que l'erreur provenait d'un appel à `colMeans`. Pourquoi la fonction `colMeans` est-elle appelée alors que la valeur de `x` fournie en entrée est un vecteur ?

Outil 2 : la fonction `browser` (seule, avec `trace` ou avec `debug`)

La fonction `browser` permet d'interrompre l'exécution d'une fonction, de donner accès à l'environnement d'exécution de la fonction et d'exécuter le corps de la fonction une instruction à la fois. Pour ce faire, il suffit d'insérer l'instruction

```
browser()
```

dans le corps de la fonction, à l'endroit où nous souhaitons interrompre l'exécution. Ensuite, il faut soumettre de nouveau la définition de la fonction. Le prochain appel à cette fonction sera interrompu lorsque l'instruction `browser()` sera rencontré.

Lorsque l'outil d'inspection de code ouvert par la fonction `browser` est actif, le symbole d'invite de commandes (*prompt*) dans la console devient `> Browse[d]` au lieu de `>`. Ici, `d` représente la profondeur de la séquence d'appels de fonctions. Les **mots-clés** suivants sont alors compris (voir `help(browser)` pour la liste complète des mots-clés) :

- `n` : pour exécuter la prochaine commande,
- `c` : pour exécuter jusqu'au prochain point d'arrêt (ex. une autre commande `browser()`),
- `Q` : pour sortir de l'outil d'inspection de code et retourner au mode R interactif usuel ;

Il est aussi possible de soumettre n'importe quelle commande R dans l'outil d'inspection de code. Par contre, si un objet porte le nom d'un des mots-clés, nous ne pouvons plus taper directement son nom dans la console pour l'afficher. Il faut passer par une commande telle que `print(n)`.

> La commande `browser()` peut être appelée un point d'arrêt (*breakpoint*), pour réutiliser un terme usuel en débogage informatique.

RStudio :

L'environnement intégré de développement RStudio offre des fonctionnalités facilitant grandement l'utilisation de la fonction `browser`. Lorsque la fonction `browser` est appelée, RStudio :

- ouvre une fenêtre contenant le corps de la fonction dans laquelle `browser` a été appelé et souligne en jaune le prochain bout de code à être soumis dans l'exécution pas à pas,
- permet de visualiser le contenu de l'environnement d'exécution de la fonction dans laquelle `browser` a été appelé à partir de la sous-fenêtre *Environment*,
- ajoute dans l'entête de la console une barre de boutons pouvant remplacer l'utilisation des mots-clés `n`, `c` et `Q`.

Différentes façons d'insérer un appel à la fonction `browser` dans le corps d'une fonction

- À la main, en éditant le code source.

Il ne faut pas oublier d'aller retirer la commande et de soumettre de nouveau le code source de la fonction lorsque le débogage est terminé. Ainsi, l'outil d'inspection de code ne sera plus ouvert à chaque fois que la fonction est appelée.

- En utilisant la fonction `trace`, comme suit :

```
trace(mean_2, browser)
```

L'argument `at` permet de spécifier à quel endroit dans le code la commande `browser()` doit être insérée. Par défaut elle est mise dans la première ligne. La commande `browser()` est ensuite retirée avec la fonction `untrace`, comme suit :

```
untrace(mean_2)
```

- En utilisant la fonction `debug`, comme suit :

```
debug(mean_2)
```

La commande `browser()` est alors insérée dans la première ligne du corps de la fonction `mean_2`. La commande `browser()` est ensuite retirée avec la fonction `undebug`, comme suit :

```
undebug(mean_2)
```

- En utilisant une fonctionnalité de RStudio : insérer un point d'arrêt dans un code source, soit en cliquant dans la marge de droite dans l'éditeur de script R de RStudio, ou par le menu *Debug -> Toggle Breakpoint*.

Outil 3 : l'option `error`

Il est possible du faire du **débogage post mortem** en R. Ce type de débogage consiste à tenter de trouver la cause d'une erreur après que l'exécution de la fonction ait été interrompue. La fonction `traceback` est donc en fait un outil de débogage post mortem, mais pas très puissant.

Si nous donnons comme valeur à l'option globale nommée `error` la fonction `recover` comme suit

```
options(error = recover)
```

R donne accès à l'environnement d'exécution de toute fonction dans laquelle une erreur est générée. Par exemple, essayons de soumettre la commande

```
mean_2(1:4)  # fonction non soumise ici, à essayer dans une session R
```

R nous demande alors d'identifier l'environnement que nous souhaitons inspecter : celui de l'exécution de `mean_2` ou celui de l'exécution de `colMeans` (car l'erreur a été rencontrée dans un appel à `colMeans`, qui a eu lieu dans un appel à `mean_2`). Après avoir fait notre choix, nous pouvons visualiser les objets dans l'environnement d'exécution choisi.

Pour remettre l'option `error` à sa valeur par défaut, il faut soumettre le code suivant :

```
options(error = NULL)
```

Dans l'exemple de la fonction `mean_2`, vous l'avez déjà trouvé, l'erreur est simplement que ce n'est pas la bonne branche du `if` qui est sélectionné selon la nature de `x`.

Correction :

```
mean_2 <- function(x) {
  if (!is.null(dim(x))) { # ajout d'une négation ici
    colMeans(x)
```

```
  } else {
    mean(x)
  }
}

mean_2(matrix(1:4, 2, 2))
```

```
## [1] 1.5 3.5
```

```
mean_2(1:4)
```

```
## [1] 2.5
```

Outil 4 : `print` et `cat`

Les fonctions `print` et `cat` s'avère aussi être des outils de débogage très simples en R. Ces fonctions permettent d'imprimer une trace des calculs effectués dans la fonction.

Exemple :

Intéressons-nous au cas particulier d'une boucle qui est arrêtée à cause d'une erreur. Il est alors informatif de savoir quelle itération de la boucle est problématique.

Voici un exemple de fonction qui sert à inverser une série de matrices fournies en entrée.

```
inverses <- function(...) {
  matrices <- list(...)
  inverses <- vector(mode = "list", length = length(matrices))
  for (i in 1:length(matrices)) {
    inverses[[i]] <- solve(matrices[[i]])
  }
  inverses
}

inverses(
  a = matrix(1:4, 2, 2),
  b = matrix(c(1, 0, -2, 0, 1, 2, -1, -2, -2), 3, 3),
  c = matrix(c(1, 3, 2, 6, 4, 2, 3, 5, 6) , 3, 3)
)
```

```
## Error in solve.default(matrices[[i]]) :
##  Lapack routine dgesv: system is exactly singular: U[3,3] = 0
```

Il est possible d'obtenir de l'information concernant l'itération problématique avec du débogage post mortem utilisant l'option `error`. Une autre possibilité serait de faire imprimer une trace temporaire des calculs à chaque itération.

```
inverses <- function(...) {
  matrices <- list(...)
  inverses <- vector(mode = "list", length = length(matrices))
  for (i in 1:length(matrices)) {
    cat("itération", i, "\n")
    # ou
    # print(i)
    inverses[[i]] <- solve(matrices[[i]])
  }
  inverses
```

```
}

inverses(
  a = matrix(1:4, 2, 2),
  b = matrix(c(1, 0, -2, 0, 1, 2, -1, -2, -2), 3, 3),
  c = matrix(c(1, 3, 2, 6, 4, 2, 3, 5, 6) , 3, 3)
)
```

```
## itération 1
## itération 2

## Error in solve.default(matrices[[i]]) :
##  Lapack routine dgesv: system is exactly singular: U[3,3] = 0
```

Nous savons maintenant que l'erreur est causée par la deuxième matrice fournie en entrée, soit la matrice `b`.

Une fois le problème compris et réglé (ce qui sera fait plus loin pour cet exemple), nous souhaitons la plupart du temps retirer les appels à la fonction `print` ou `cat` de la fonction.

Gestion d'exceptions

Produire des erreurs et des avertissements

Nous avons parfois besoin que nos fonctions génèrent des erreurs et des avertissements, notamment :

- pour communiquer avec l'utilisateur dans le cas de résultats de calcul inattendus,
- pour la validation des arguments fournis en entrée.

Il vaut mieux arrêter l'exécution de la fonction si les arguments fournis en entrée sont incorrects et que le comportement de la fonction n'est pas approprié (mauvais calcul ou message d'erreur non informatif).

Fonctions utiles :

- Pour générer une erreur : `stop`, `stopifnot`, `match.arg` (vue dans les notes sur la création de fonctions en R),
- Pour générer un avertissement : `warning`.

Remarque : Pour la tâche spécifique de valider les valeurs fournies en argument, le package `checkmate` propose plusieurs fonctions rendant la tâche plus facile au développeur, par exemple les fonctions `checkCount`, `checkScalar`, `checkIntegerish`, etc. Nous ne verrons cependant pas ce package ici.

Exemple :

Faisons générer une erreur à notre fonction `dist_manhattan` lorsqu'elle reçoit en entrée deux vecteurs qui ne sont pas de mêmes longueurs.

```
dist_manhattan <- function(point1, point2) {
  if (length(point1) != length(point2)) {
    stop("'point1' and 'point2' must have the same length")
  }
  sum(abs(point1 - point2))
}

dist_manhattan(c(-1,0), c(1,2,3))
```

```
## Error in dist_manhattan(c(-1, 0), c(1, 2, 3)) :
##   'point1' and 'point2' must have the same length
```

ou encore

```
dist_manhattan <- function(point1, point2) {
  stopifnot(length(point1) == length(point2))
  sum(abs(point1 - point2))
}

dist_manhattan(c(-1,0), c(1,2,3))
```

```
## Error in dist_manhattan(c(-1, 0), c(1, 2, 3)): length(point1) == length(point2) is not TRUE
```

Faisons générer un avertissement à notre fonction `dist_manhattan` si les arguments `point1` et `point2` sont de dimension supérieure à 1.

```
dist_manhattan <- function(point1, point2) {
  if (length(point1) != length(point2)) {
    stop("'point1' and 'point2' must have the same length")
  }
  if (!is.null(dim(point1)) || !is.null(dim(point2))) {
    warning("'point1' and 'point2' are treated as dimension 1 vectors")
  }
  sum(abs(point1 - point2))
}

dist_manhattan(rbind(c(0,0), c(1,0)), rbind(c(3,2), c(2,3)))
```

```
## Warning in dist_manhattan(rbind(c(0, 0), c(1, 0)), rbind(c(3, 2), c(2, 3))):
## 'point1' and 'point2' are treated as dimension 1 vectors
```

```
## [1] 9
```

Nos tests ne devraient maintenant plus échouer.

```
test_that(
  "nous reproduisons un calcul à la main",
  expect_equal(dist_manhattan(point1 = c(0,-5), point2 = c(0,-15)), 10)
)

test_that(
  "nous obtenons le même résultat que la fonction dist",
  expect_equal(
    dist_manhattan(point1 = c(0,0), point2 = c(1,1)),
    as.vector(dist(rbind(c(0,0), c(1,1)), method = "manhattan"))
  )
)

test_that(
  "des vecteurs de dimensions différentes génèrent une erreur",
  expect_error(dist_manhattan(point1 = c(-1,0), point2 = c(1,2,3)))
)

test_that(
  "des matrices génèrent un avertissement",
  {
    mat1 <- rbind(c(0,0), c(1,0))
```

```
    mat2 <- rbind(c(3,2), c(2,3))
    expect_warning(dist_manhattan(point1 = mat1, point2 = mat2))
  }
)
```

C'est bien le cas, car ce code de tests roule sans erreur.

Manipuler des erreurs et des avertissements

Il est possible d'attraper des erreurs et de les manipuler avec la fonction `try`. Cette fonction permet entre autres d'éviter l'arrêt d'une boucle lorsqu'une erreur est rencontrée pour une certaine itération.

Exemple :

Rappelons que l'exécution de notre fonction `inverses` est arrêtée dès qu'elle rencontre une matrice non inversible :

```
inverses(
  a = matrix(1:4, 2, 2),
  b = matrix(c(1, 0, -2, 0, 1, 2, -1, -2, -2), 3, 3),
  c = matrix(c(1, 3, 2, 6, 4, 2, 3, 5, 6) , 3, 3)
)
```

```
## Error in solve.default(matrices[[i]]) :
##  Lapack routine dgesv: system is exactly singular: U[3,3] = 0
```

Il serait plutôt souhaitable que le calcul soit fait pour toutes les matrices, en sautant celles non inversibles.

```
inverses <- function(...) {
  matrices <- list(...)
  inverses <- vector(mode = "list", length = length(matrices))
  for (i in 1:length(matrices)) {
    tentative <- try(solve(matrices[[i]]), silent = TRUE)
    if (inherits(tentative, "try-error")) {
      # Si la commande a généré une erreur, retourner une matrice de NA
      inverses[[i]] <- matrix(NA, nrow = nrow(matrices[[i]]), ncol = ncol(matrices[[i]]))
    } else {
      # Sinon, retourner la matrice inversée
      inverses[[i]] <- tentative
    }
  }
  inverses
}

inverses(
  a = matrix(1:4, 2, 2),
  b = matrix(c(1, 0, -2, 0, 1, 2, -1, -2, -2), 3, 3),
  c = matrix(c(1, 3, 2, 6, 4, 2, 3, 5, 6) , 3, 3)
)
```

```
## [[1]]
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
##
## [[2]]
```

```
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
## [3,]   NA   NA   NA
##
## [[3]]
##       [,1]  [,2]  [,3]
## [1,] -0.35  0.75 -0.45
## [2,]  0.20  0.00 -0.10
## [3,]  0.05 -0.25  0.35
```

Il faut donner comme premier argument à la fonction `try` une expression. Dans l'exemple précédent, il s'agissait d'une seule instruction. Il aurait aussi pu s'agir d'une série d'instructions, entre accolades. L'argument `silent = TRUE` a signifié à `try` de ne pas afficher de messages.

L'objet retourné par `try` est l'objet retourné par l'expression fournie en premier argument si aucune erreur n'est rencontrée. Sinon, il s'agit d'un objet de classe `"try-error"` contenant le message d'erreur. L'instruction `inherits(tentative, "try-error")` retourne `TRUE` si l'objet `tentative` possède la classe `"try-error"`, `FALSE` sinon. L'utilisation de la fonction `inherits` est l'outil recommandé pour tester l'appartenance d'un objet à un classe.

Autres outils pour manipuler des erreurs et des avertissements en R :

- l'option globale `warn`
  - si `warn` prend une valeur négative : tous les avertissements sont ignorés,
  - si `warn` prend la valeur 0 (option par défaut) : les avertissements sont affichés à la fin de l'exécution de la fonction,
  - si `warn` prend la valeur 1 : les avertissements sont affichés au fur et à mesure qu'ils surviennent,
  - si `warn` prend la valeur 2 : tous les avertissements sont transformés en erreurs ;
- la fonction `suppressWarnings` : permet d'ignorer les avertissements générés par des instructions R spécifiques.

```
# Exemple d'utilisation de la fonction suppressWarnings
suppressWarnings(dist_manhattan(rbind(c(0,0), c(1,0)), rbind(c(3,2), c(2,3))))
```

```
## [1] 9
```

Résumé

Bonnes pratiques dans le développement de fonctions

1. **Planifier** le travail (pas de programmation encore) :
   - définir clairement la tâche à accomplir par la fonction et la sortie qu'elle doit produire,
   - prévoir les étapes à suivre afin d'effectuer cette tâche,
   - identifier les arguments devant être fournis en entrée à la fonction.
2. **Développer le corps de la fonction**
   - 2.1. Écrire le programme par étapes, d'abord sans former la fonction, en commentant bien le code et en travaillant sur des mini-données test.
   - 2.2 Pour chaque petite étape ou sous-tâche, tester interactivement si le programme produit le résultat escompté (tester souvent en cours de travail, ainsi il y a moins de débogage à faire).
3. **Créer la fonction** à partir du programme développé.
4. **Documenter** la fonction.
5. **Tester** la fonction : sauvegarder nos tests et bien les structurer, car ils serviront souvent.
6. Si nous rencontrons des comportements indésirables lors des tests, **déboguer** la fonction :
   - cerner le ou les problèmes,

- apporter les correctifs nécessaires à la fonction (que ce soit dans son corps ou dans liste de ses arguments),
- adapter la documentation et les tests au besoin,
- rouler de nouveau les tests,
- répéter ces sous-étapes jusqu'à ce que les tests ne révèlent plus aucun problème à régler ou aucune amélioration à apporter.

Organisation du code

- placer la définition de nos fonctions dans un fichier distinct, disons `mes_fonctions.R`;
- inclure en entête de tout programme R utilisant ces fonctions la commande `source("chemin/mes_fonctions.R")`.

Tests

Appeler les fonctions en donnant en entrée des valeurs d'arguments pour lesquelles nous savons quel résultat nous devrions obtenir.

- Mini-exemples pour lesquels nous pouvons faire les calculs à la main pour trouver le résultat escompté.
- Comparer les résultats de nos fonctions aux résultats de fonctions effectuant le même calcul, s'il en existe.
- Comparer les résultats de nos fonctions à des résultats théoriques ou des résultats publiés dans des articles scientifiques, si applicable.

Objectifs :

1. obtenir le résultat escompté
2. correctement gérer les exceptions

Exception = situation anormale ou particulière qui requiert un traitement spécial

Des exemples d'exceptions sont :

- des arguments incorrects fournis en entrée,
- des résultats de calcul inattendus.

Réaction à des exceptions : erreurs ou avertissements générés.

Différents types de **conditions en R** :

- **Erreur** : exécution interrompue, message d'erreur affiché.
- **Avertissement** : exécution non interrompue, message d'avertissement affiché pour signaler un problème potentiel.
- **Message** (moins fréquent) : exécution non interrompue, message affiché pour apporter une information supplémentaire.

Formaliser des tests avec le package testthat

Fonctions pour l'écriture, l'organisation et l'exécution automatique de tests unitaires en R :

- fonction d'écriture (un appel à une de ces fonctions = un test unitaire) :
  - `expect_equal`,
  - `expect_error`,
  - `expect_warning`,
  - `expect_true`,
  - etc.
- fonction d'organisation : `context`, `test_that`
- fonction d'exécution : `test_file`, `test_dir`, etc.

Débogage

débogage = processus méthodique pour trouver et régler des bogues informatiques

bogues = anomalies de fonctionnement d'un programme

Outils de débogage en R :

1. `traceback()` : retourne la séquence des appels de fonctions provoquant une erreur
2. fonction `browser` (seule, avec `trace` ou avec `debug`) : permet
   - d'interrompre l'exécution d'une fonction,
   - de donner accès à l'environnement d'exécution de la fonction et
   - d'exécuter le corps de la fonction une commande à la fois ;
3. option `error` : débogage post mortem
   donne accès à l'environnement d'exécution de la fonction au moment de la génération d'une erreur.
4. `print` et `cat` : imprime une trace des calculs

Gestion d'exceptions

Produire des erreurs et des avertissements

- générer une erreur : `stop`, `stopifnot`, `match.arg` ;
- générer un avertissement : `warning`.

Manipuler des erreurs et des avertissements

- attraper des erreurs et les manipuler : `try` ;
- modifier la gestion des avertissements dans la session R : option globale `warn` ;
- ignorer des avertissements : `suppressWarnings`.

Références

Tests :

- Wickham, H., RStudio et R Core team (2020). testthat : Unit Testing for R. R package version 2.3.2. https://CRAN.R-project.org/package=testthat
  - documentation du package : fiches d'aide du package, http://testthat.r-lib.org/
  - Wickham, H. (2015). *R packages*. O'Reilly. Chapitre 7. http://r-pkgs.had.co.nz/tests.html

Débogage :

- Matloff, N. (2011). *The Art of R Programming : A Tour of Statistical Software Design*. No Starch Press. Chapitre 13.
- Peng, R. D. (2019). *R Programming for Data Science*. Chapitre 18. https://bookdown.org/rdpeng/rprogdatascience/debugging.html
- R Core Team (2020). *Writing R Extensions*. R Foundation for Statistical Computing. Chapitre 4. http://cran.r-project.org/doc/manuals/r-release/R-exts.pdf
- Wickham, H. (2019). Advanced R, Second Edition. Chapman and Hall/CRC. Chapitre 8. https://adv-r.hadley.nz/debugging.html).