Lesson notes | Coding basics (Translation 2, Joel from Fiverr)

Created by the GRAPH Courses team

January 2023

This document serves as an accompaniment for a lesson found on https://thegraphcourses.org.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, a non-profit headquartered at the University of Geneva Global Health Institute, and supported by the World Health Organization (WHO) and other partners

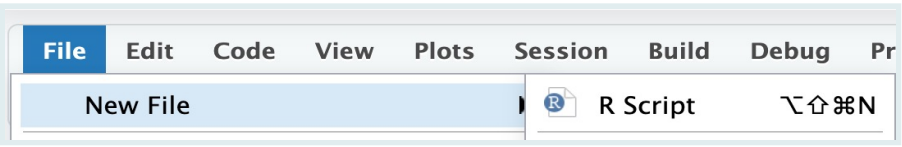
Objectifs d'apprentissage

1. Vous pouvez écrire des commentaires dans R.
2. Vous pouvez créer des en-têtes de section dans RStudio.
3. Vous savez utiliser R comme une calculatrice.
4. Vous savez créer, écraser et manipuler des objets R.
5. Vous comprenez les règles de base pour nommer les objets R.
6. Vous comprenez la syntaxe d'appel des fonctions R.
7. Vous savez comment imbriquer plusieurs fonctions.
8. Vous pouvez utiliser l'installation et le chargement de paquets R complémentaires et appeler les fonctions de ces paquets.

Introduction

Dans la dernière leçon, vous avez appris à utiliser RStudio, le merveilleux environnement de développement intégré (IDE) qui facilite grandement le travail avec R. Dans cette leçon, vous apprendrez les bases de l'utilisation de R lui-même. Dans cette leçon, vous allez apprendre les bases de l'utilisation de R lui-même.

Pour commencer, ouvrez RStudio, et ouvrez un nouveau script avec `File > New File > R Script` dans le menu RStudio.



Ensuite, **enregistrez le script** avec `File > Save` dans le menu RStudio ou en utilisant le raccourci `Command`/`Control` + `S` . Cela devrait faire apparaître la boîte de dialogue Enregistrer le fichier. Enregistrez le fichier avec un nom comme "coding_basics".

Vous devez maintenant taper tout le code de cette leçon dans ce script.

Commentaires

Il existe deux principaux types de texte dans un script R : les commandes et les commentaires. Une commande est une ou plusieurs lignes de code R qui demande à R de faire quelque chose (par exemple `2 + 2`).

Un commentaire est un texte qui est ignoré par l'ordinateur.

Tout ce qui suit le symbole `#` (prononcé "dièse" ou "livre") sur une ligne donnée est un commentaire. Essayez de taper et d'exécuter le code ci-dessous pour vous en rendre compte :

```
# Un commentaire
2 + 2 # Un autre commentaire
# 2 + 2
```

Puisqu'ils sont ignorés par l'ordinateur, les commentaires sont destinés aux *humains*. Ils vous aident, ainsi que les autres, à garder une trace de ce que fait votre code. Utilisez-les souvent ! Comme votre mère le dit toujours, "trop de tout est mauvais, sauf les commentaires R".

Question 1

Vrai ou Faux : les deux morceaux de code ci-dessous sont des façons valides de commenter du code: ?

```
# additionner deux nombres
2 + 2
```

```
2 + 2 # additionner deux nombres
```

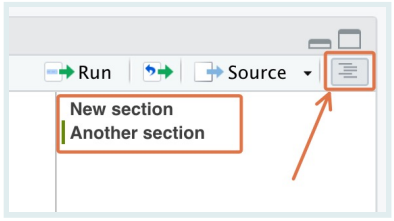
Note : Toutes les réponses aux questions se trouvent à la fin de la leçon.

Une utilisation fantastique des commentaires est de séparer vos scripts en sections. Si vous mettez quatre tirets après un commentaire, RStudio créera une nouvelle section dans votre code :

```
# Nouvelle section ----
```

Cela présente deux avantages intéressants. Tout d'abord, vous pouvez cliquer sur la petite flèche à côté de l'en-tête de la section pour replier, ou réduire, cette section de code :

Ensuite, vous pouvez cliquer sur l'icône "Outline" en haut à droite de l'éditeur pour afficher et parcourir tout le contenu de votre script :



R est une calculatrice

R fonctionne comme une calculatrice, et obéit à l'ordre correct des opérations. Tapez et exécutez les expressions suivantes et observez leur résultat :

```
2 + 2
```

```
## [1] 4
```

```
2 - 2
```

```
## [1] 0
```

```
2 * 2 # deux fois deux
```

```
## [1] 4
```

```
2 / 2 # deux divisé par deux
```

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

```
2 ^ 2 # deux élevé à la puissance deux
```

```
## [1] 4
```

```
2 + 2 * 2 # ceci est évalué en suivant l'ordre des opérations
```

```
## [1] 6
```

```r
sqrt(100) # racine carrée
```

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

La commande de racine carrée présentée sur la dernière ligne est un bon exemple de *fonction* R, où `100` est l'*argument* de la fonction. Vous verrez bientôt d'autres fonctions.

Nous espérons que vous vous souvenez du raccourci pour exécuter du code !

Pour **exécuter une seule ligne de code**, placez votre curseur n'importe où sur cette ligne, puis tapez `Command` + `Enter` sur macOS, ou `Control` + `Enter` sur Windows.

Pour **exécuter plusieurs lignes**, faites glisser votre curseur pour mettre en surbrillance les lignes concernées, puis appuyez à nouveau sur `Command`/`Control` + `Enter`.

Question 2

Dans l'expression suivante, quel signe est évalué en premier par R, le moins ou la division ?

```r
2 - 2 / 2
```

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

Formatage du code

R ne se soucie pas de la façon dont vous choisissez d'espacer votre code.

Pour les opérations mathématiques que nous avons effectuées ci-dessus, tout ce qui suit serait du code valide :

```r
2+2
```

```
## [1] 4
```

```
## [1] 4
```

```
2 + 2
```

```
## [1] 4
```

De même, pour la fonction `sqrt()` utilisée ci-dessus, n'importe laquelle de ces propositions serait valide :

```
sqrt(100)
```

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

```
sqrt( 100 )
```

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

```
# Vous pouvez même espacer la commande sur plusieurs lignes
sqrt(
  100
    )
```

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

Mais bien sûr, vous devriez essayer d'espacer votre code de manière raisonnable. Qu'est-ce qui est exactement "raisonnable" ? Eh bien, il peut être difficile pour vous de le savoir pour le moment. Avec le temps, en lisant le code d'autres personnes, vous apprendrez qu'il existe certaines *conventions* R pour l'espacement et le formatage du code.

En attendant, vous pouvez demander à RStudio de vous aider à formater votre code. Pour ce faire, mettez en surbrillance une section de code que vous voulez reformater, et, dans le menu RStudio, allez à `Code > Reformater le code`, ou utilisez le raccourci `Shift` + `Command/Control` + `A`.

Attachez le signe +

Si vous exécutez une ligne de code incomplète, R imprimera un signe + pour indiquer qu'il attend que vous terminiez le code.

Par exemple, si vous exécutez le code suivant :

```
sqrt(100
```

vous n'obtiendrez pas le résultat attendu (10). La console affichera plutôt `sqrt(` et un signe `+` :

```
> sqrt(100
+
```

R vous attend pour compléter la parenthèse fermante. Vous pouvez compléter le code et vous débarrasser du `+` en saisissant simplement la parenthèse manquante :

```
)
```

```
> sqrt(100
+ )
[1] 10
```

Vous pouvez également appuyer sur la touche d'échappement `ESC` lorsque votre curseur se trouve dans la console pour recommencer.

Les objets dans R

Créer un objet

Lorsque vous exécutez du code comme nous l'avons fait ci-dessus, le résultat de la commande (ou sa *valeur*) est simplement affiché dans la console - il n'est stocké nulle part.

```
2 + 2 # R imprime ce résultat, 4, mais ne le stocke pas.
```

```
## [1] 4
```

Pour stocker une valeur en vue d'une utilisation future, affectez-la à un *objet* avec l'opérateur d'affectation,* `<-` :

```
mon_obj <- 2 + 2 # assigne le résultat de `2 + 2` à l'objet appelé `mon_obj`.
mon_obj # imprimer mon_obj
```

```
## [1] 4
```

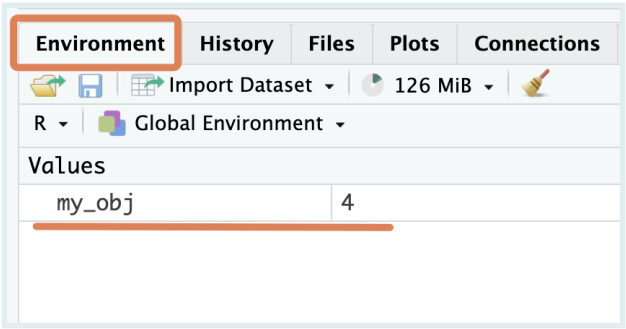
L'opérateur d'affectation, `<-` , est composé du signe "moins que", `<` , et d'un moins, `-` . Vous l'utiliserez des milliers de fois au cours de votre vie R, alors ne le tapez pas manuellement ! Utilisez plutôt le raccourci de RStudio, **`alt` + `-`** (**alt** ET **minus**) sous Windows ou **`option` + `-`** (**option** ET **minus**) sous macOS.

::: note complémentaire Notez également que vous pouvez utiliser le signe *équivalents*, `=`, pour l'affectation.

```
mon_objet = 2 + 2
```

Mais cette méthode n'est pas couramment utilisée par la communauté R (principalement pour des raisons historiques), c'est pourquoi nous la déconseillons également. Suivez la convention et utilisez `<-`. :::

Maintenant que vous avez créé l'objet `my_obj`, R sait tout à son sujet et en gardera la trace pendant cette session R. Vous pouvez voir tous les objets créés dans le menu déroulant. Vous pouvez voir tous les objets créés dans l'onglet *Environnement* de RStudio.

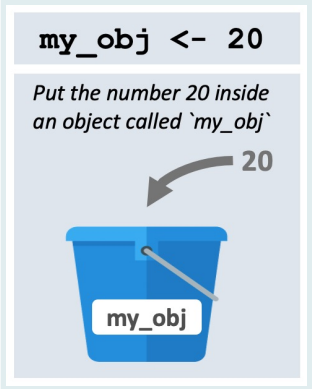


Qu'est-ce qu'un objet ?

Qu'est-ce qu'un objet exactement ? Pensez-y comme un seau nommé qui peut contenir n'importe quoi. Lorsque vous exécutez le code ci-dessous :

```
mon_obj <- 20
```

vous dites à R, "mettez le nombre 20 dans un seau nommé 'mon_obj'".

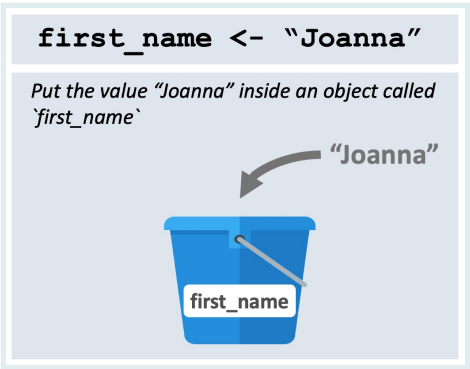


Une fois le code exécuté, nous dirons, en termes R, que "la valeur de l'objet appelé `mon_obj` est 20".

Et si vous exécutez ce code :

```
prénom <- "Joanna"
```

vous donnez l'ordre à R de "mettre la valeur 'Joanna' dans le panier appelé 'first_name'".



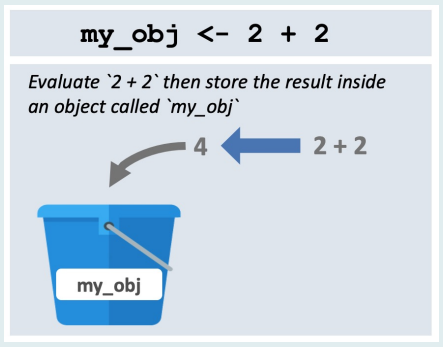
Une fois le code exécuté, nous dirons, en termes R, que "la valeur de l'objet `first_name` est Joanna".

Notez que R évalue le code *avant* de le placer dans le seau.

Donc, avant, quand nous avons exécuté ce code,

```
mon_obj <- 2 + 2
```

R fait d'abord le calcul de `2 + 2`, puis stocke le résultat, 4, à l'intérieur de l'objet.



Question 3

Considérez le morceau de code ci-dessous :

```
résultat <- 2 + 2 + 2
```

Quelle est la valeur de l'objet `result` créé ?

A. `2 + 2 + 2`

B. numérique

C. 6

Les ensembles de données sont aussi des objets

Jusqu'à présent, vous avez travaillé avec des objets très simples. Vous vous dites peut-être : "Où sont les feuilles de calcul et les ensembles de données ? Pourquoi écrit-on `mon_obj <- 2 + 2` ? Est-ce que c'est un cours de maths de l'école primaire ?".

Soyez patient.

Nous souhaitons que vous vous familiarisiez avec le concept d'objet R, car lorsque vous commencerez à traiter des ensembles de données réels, ceux-ci seront également stockés sous forme d'objets R.

Voyons maintenant un aperçu de ceci. Tapez le code ci-dessous pour télécharger un ensemble de données sur les cas d'Ebola que nous avons stocké sur Google Drive et le mettre dans l'objet `ebola_sierra_leone_data`.

```
ebola_sierra_leone_data <- read.csv("https://tinyurl.com/ebola-data-sample")
ebola_sierra_leone_data # print ebola_data
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
```

```
## 4 187  NA   F confirmed    2014-06-19     2014-06-24 Kailahun
## 5  85  20   M confirmed    2014-06-08     2014-06-24 Kailahun
```

Ces données contiennent un échantillon d'informations sur les patients de l'épidémie d'Ebola de 2014-2016 en Sierra Leone.

Comme vous pouvez stocker les ensembles de données sous forme d'objets, il est très facile de travailler avec plusieurs ensembles de données en même temps.

Ci-dessous, nous importons et visualisons un autre jeu de données depuis le Web :

```
diabète_chine <- read.csv("https://tinyurl.com/diabetes-china")
```

Comme l'ensemble de données ci-dessus est assez volumineux, il peut être utile de l'examiner dans le visualiseur de données :

```
Vue(diabète_chine)
```

Remarquez que les deux ensembles de données apparaissent maintenant dans votre onglet *Environnement*.

Plutôt que de lire des données à partir d'un disque Internet comme nous l'avons fait ci-dessus, il est plus probable que vous ayez les données sur votre ordinateur et que vous souhaitiez les lire dans R à partir de celui-ci. Nous aborderons ce sujet dans une prochaine leçon.

Plus tard dans le cours, nous vous montrerons également comment stocker et lire des données à partir d'un service Web comme Google Drive, ce qui est pratique pour une portabilité facile.

Renommer un objet

Vous souhaitez parfois renommer un objet. Il n'est pas possible de le faire directement.

Pour renommer un objet, on fait une copie de l'objet avec un nouveau nom, et on supprime l'original.

Par exemple, on peut décider que le nom de l'objet `ebola_sierra_leone_data` est trop long. Pour le remplacer par le plus court "ebola_data", exécutez :

```
ebola_data <- ebola_sierra_leone_data
```

Ceci a copié le contenu du *bucket* "ebola_sierra_leone_data" vers un nouveau *bucket* "ebola_data".

Vous pouvez maintenant vous débarrasser de l'ancien seau `ebola_sierra_leone_data` avec la fonction `rm()`, qui signifie "remove" :

```
rm(ebola_sierra_leone_data)
```

Écraser un objet

Ecraser un objet revient à modifier le *contenu* d'un *bucket*.

Par exemple, nous avons précédemment exécuté ce code pour stocker la valeur "Joanna" dans l'objet `first_name` :

```
premier_nom <- "Joanna"
```

Pour remplacer cette valeur par une autre, il suffit de ré-exécuter la ligne avec une autre valeur :

```
prénom <- "Luigi"
```

Vous pouvez jeter un coup d'oeil à l'onglet Environnement pour observer le changement.

Travailler avec des objets

La plupart de votre temps dans R sera consacré à la manipulation d'objets R. Voyons quelques exemples rapides.

Vous pouvez exécuter des commandes simples sur les objets. Par exemple, ci-dessous, nous stockons la valeur `100` dans un objet, puis nous prenons la racine carrée de cet objet :

```
mon_nombre <- 100
sqrt(mon_nombre)
```

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

R "voit" `my_number` comme le nombre 100, et est donc capable d'évaluer sa racine carrée.

Vous pouvez également combiner des objets existants pour créer de nouveaux objets. Par exemple, tapez le code ci-dessous pour ajouter `mon_nombre` à lui-même, et stocker le résultat dans un nouvel objet appelé `mon_somme` :

```
ma_somme <- mon_nombre + mon_nombre
```

Quelle devrait être la valeur de "ma_somme" ? Faites d'abord une supposition, puis vérifiez-la.

Pour vérifier la valeur d'un objet, tel que `mon_somme`, vous pouvez taper et exécuter juste le code `mon_somme` dans la Console ou l'Editeur. Vous pouvez aussi simplement mettre en surbrillance la valeur `my_sum` dans le code existant et appuyer sur `Command/Control` + `Enter`.

Mais bien sûr, la plupart de vos analyses impliqueront de travailler avec des objets *data*, comme l'objet `ebola_data` que nous avons créé précédemment.

Voyons un exemple très simple de la façon d'interagir avec un objet de données ; nous l'aborderons correctement dans la prochaine leçon.

Pour obtenir un tableau de la répartition des patients par sexe dans l'objet `ebola_data`, nous pouvons exécuter ce qui suit :

```
table(ebola_data$sex)
```

```
##
##   F   M
## 124  76
```

Le symbole du dollar, `$`, ci-dessus nous a permis de faire un sous-ensemble d'une colonne spécifique.

Question 4

a. Considérez le code ci-dessous. Quelle est la valeur de l'objet `answer` ?

```
huit <- 9
réponse <- huit - 8
```

b. Utilisez `table()` pour créer un tableau avec la répartition des patients entre les districts dans l'objet `ebola_data`.

Quelques erreurs avec les objets

```
first_name <- "Luigi"
nom_de_famille <- "Fenway"
```

```
nom_complet <- nom_prénom + nom_famille
```

```
Erreur dans first_name + last_name : argument non numérique à l'opérateur
binaire
```

Le message d'erreur vous indique que ces objets ne sont pas des nombres et ne peuvent donc pas être additionnés avec +. Il s'agit d'un type d'erreur assez courant, causé par la tentative de faire des choses inappropriées à vos objets. Faites attention à cela.

Dans ce cas particulier, nous pouvons utiliser la fonction `paste()` pour mettre ces deux objets ensemble :

```
nom_complet <- paste(first_name, last_name)
nom_complet
```

```
## [1] "Luigi Fenway"
```

Une autre erreur que vous rencontrerez souvent est "Erreur : objet 'XXX' non trouvé". Par exemple :

```
mon_nombre <- 48 # Définir `mon_objet`.
mon_nombre + 2 # tentative d'ajout de 2 à `mon_objet'.
```

```
Erreur : l'objet 'Mon_nombre' n'a pas été trouvé.
```

Ici, R renvoie un message d'erreur car nous n'avons pas encore créé (ou *défini*) l'objet `Mon_obj`. (Rappelez-vous que R est sensible à la casse).

Lorsque vous commencez à apprendre R, traiter les erreurs peut être frustrant. Elles sont souvent difficiles à comprendre (par exemple, que signifie exactement "*Argument non numérique à l'opérateur binaire*").

Essayez de rechercher sur Google les messages d'erreur que vous obtenez et parcourez les premiers résultats. Cela vous mènera à des forums (par exemple, stackoverflow.com) où d'autres apprenants de R se sont plaints de la même erreur. Vous y trouverez peut-être des explications et des solutions à vos problèmes.

Question 5

a. Le code ci-dessous renvoie une erreur. Pourquoi ?

```
mon_prénom <- "Kene"
mon_nom_de_noms <- "Nwosu"
mon_prénom + mon_nom_de_famille
```

b. Le code ci-dessous renvoie une erreur. Pourquoi ? (Regardez attentivement)

```
mon_1er_nom <- "Kene"
mon_dernier_nom <- "Nwosu"

paste(mon_nom_inst, mon_nom_dernier)
```

Nommer des objets

> Il n'y a que deux choses difficiles en informatique : l'invalidation du cache et le nommage des objets.
>
> – Phil Karlton.

Comme une grande partie de votre travail dans R implique d'interagir avec des objets que vous avez créés, il est important de choisir des noms intelligents pour ces objets.

Il est difficile de nommer les objets car les noms doivent être à la fois **courts** (pour que vous puissiez les taper rapidement) et **informatifs** (pour que vous puissiez facilement vous souvenir de ce que contient l'objet), et ces deux objectifs sont souvent en conflit.

Ainsi, les noms trop longs, comme celui ci-dessous, sont mauvais car ils prennent une éternité à taper.

```
échantillon_de_l'ensemble_de_données_de_l'éruption_de_bola_de_sierra_leone_en_20
```

Un nom comme " data " est mauvais car il n'est pas informatif ; le nom ne donne pas une bonne idée de ce qu'est l'objet.

Au fur et à mesure que vous écrirez du code R, vous apprendrez à écrire des noms courts et informatifs.

Pour les noms comportant plusieurs mots, il existe quelques conventions sur la façon de séparer les mots :

```
snake_case <- "Snake case utilise des underscores".
period.case <- "Period case utilise des points"
camelCase <- "Camel case met les nouveaux mots en majuscules (mais pas le
        premier mot)"
```

Nous recommandons snake_case, qui utilise tous les mots en minuscules, et sépare les mots avec _.

Notez également qu'il existe certaines limitations sur les noms d'objets : - Les noms doivent commencer par une lettre. Ainsi, `2014_data` n'est pas un nom valide (car il commence par un chiffre).

- Les noms ne peuvent contenir que des lettres, des chiffres, des points (.) et des traits de soulignement (_). Donc `ebola-data` ou `ebola~data` ou `ebola data` avec un espace ne sont pas des noms valides.

Si vous voulez vraiment utiliser ces caractères dans vos noms d'objets, vous pouvez entourer les noms de points de suspension :

```
`ebola-data`
`ebola~data`
`ebola data`
```

Tous les caractères ci-dessus sont des noms d'objets R valides. Par exemple, tapez et exécutez le code suivant :

```
`ebola~data` <- ebola_data
`ebola~data`
```

Mais en général, vous devriez éviter d'utiliser des backticks pour sauver les mauvais noms d'objets. Ecrivez simplement des noms corrects.

**Question 6

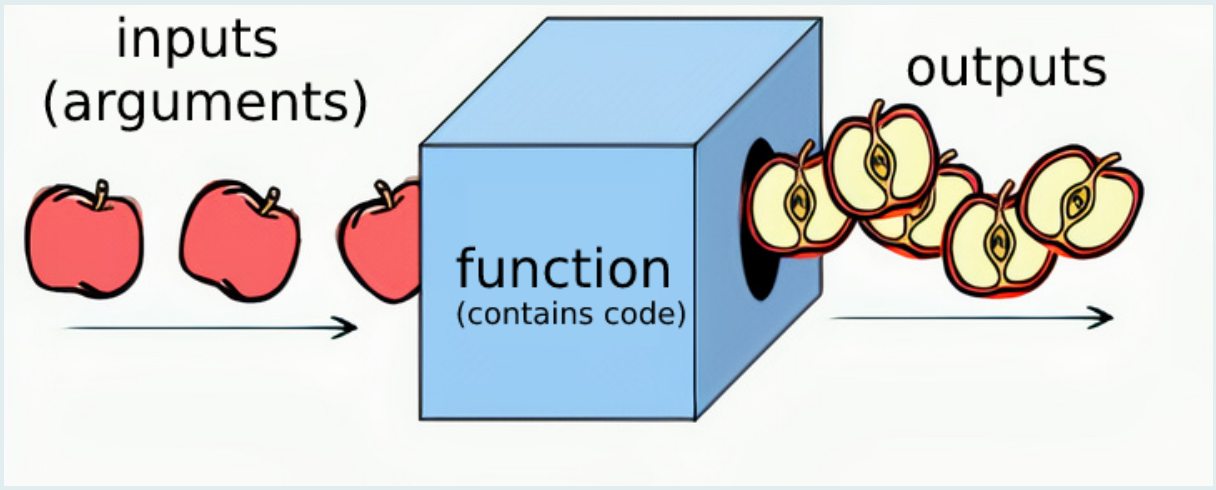
Dans le morceau de code ci-dessous, nous essayons de prendre les 20 premières lignes de la table `ebola_data`. Toutes ces lignes, sauf une, contiennent une erreur. Quelle ligne s'exécutera correctement ?

```
20_top_rows <- head(ebola_data, 20)
twenty-top-rows <- head(ebola_data, 20)
top_20_rows <- head(ebola_data, 20)
```

Fonctions

Une grande partie de votre travail dans R consistera à appeler des *fonctions*.

Vous pouvez considérer chaque fonction comme une machine qui prend une entrée (ou *arguments*) et renvoie une sortie.



Jusqu'à présent, vous avez déjà vu de nombreuses fonctions, dont `sqrt()`, `paste()` et `plot()`. Exécutez les lignes ci-dessous pour vous rafraîchir la mémoire :

```
sqrt(100)
paste("Je suis un nombre", 2 + 2)
plot(femmes)
```

Syntaxe de base des fonctions

La façon standard d'appeler une fonction est de fournir une *valeur* pour chaque *argument* :

```
nom_fonction(argument1 = "valeur", argument2 = "valeur")
```

Faisons une démonstration avec la fonction `head()`, qui renvoie les premiers éléments d'un objet.

Pour renvoyer les trois premières lignes de l'ensemble de données Ebola, vous exécutez :

```
head(x = ebola_data, n = 3)
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
```

Dans le code ci-dessus, `head()` prend deux arguments :

- `x`, l'objet d'intérêt, et
- `n`, le nombre d'éléments à retourner.

Nous pouvons également intervertir l'ordre des arguments :

```
head(n = 3, x = ebola_data)
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
```

Si vous mettez les valeurs des arguments dans le bon ordre, vous pouvez éviter de taper leurs noms. Ainsi, les deux lignes de code suivantes sont équivalentes et s'exécutent toutes les deux :

```
head(x = ebola_data, n = 3)
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
```

```
head(ebola_data, 3)
```

```
##    id age sex    status date_of_onset date_of_sample district
## 1 167  55   M confirmed    2014-06-15     2014-06-21   Kenema
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
```

Mais si les valeurs des arguments sont dans le mauvais ordre, vous obtiendrez une erreur si vous ne tapez pas les noms des arguments. Ci-dessous, la première ligne s'exécute mais la seconde ne s'exécute pas :

```
head(n = 3, x = ebola_data)
head(3, ebola_data)
```

(Pour connaître le "bon ordre" des arguments, consultez le fichier d'aide de la fonction `head()`)

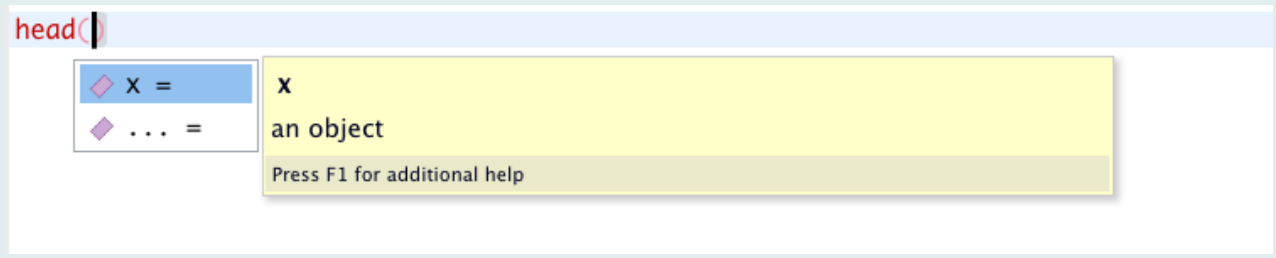
Certains arguments de fonction peuvent être ignorés, car ils ont des valeurs *par défaut*.

Par exemple, avec `head()`, la valeur par défaut de `n` est 6, donc l'exécution de `head(ebola_data)` retournera les 6 premières lignes.

```
head(ebola_data)
```

```
## 2 129  41   M confirmed    2014-06-13     2014-06-18 Kailahun
## 3 270  12   F confirmed    2014-06-28     2014-07-03 Kailahun
## 4 187  NA   F confirmed    2014-06-19     2014-06-24 Kailahun
## 5  85  20   M confirmed    2014-06-08     2014-06-24 Kailahun
## 6 277  30   F confirmed    2014-06-29     2014-07-01   Kenema
```

Pour afficher les arguments d'une fonction, appuyez sur la touche **Tab** lorsque votre curseur se trouve à l'intérieur des parenthèses de la fonction :



Question 7

Dans les lignes de code ci-dessous, nous essayons de prendre les 6 premières lignes de l'ensemble de données `women` (qui est intégré dans R). Quelle ligne est invalide ?

```
head(femmes)
head(femmes, 6)
head(x = femmes, 6)
head(x = femmes, n = 6)
head(6, femmes)
```

(Si vous n'êtes pas sûr, essayez simplement de taper et d'exécuter chaque ligne. Rappelez-vous que le but ici est que vous gagniez un peu de pratique).

Passons un peu de temps à jouer avec une autre fonction, la fonction `paste()`, que nous avons déjà vue plus haut. Cette fonction est un peu spéciale car elle peut prendre n'importe quel nombre d'arguments en entrée.

Vous pouvez donc avoir deux arguments :

```
paste("Luigi", "Fenway")
```

```
## [1] "Luigi Fenway"
```

Ou quatre arguments :

```
paste("Luigi", "Fenway", "Luigi", "Fenway")
```

```
## [1] "Luigi Fenway Luigi Fenway"
```

Et ainsi de suite jusqu'à l'infini.

Et comme vous vous en souvenez peut-être, on peut aussi `paste()` des objets nommés :

```
prénom <- "Luigi"
paste("Mon nom est", first_name, "et mon nom de famille est", last_name)
```

```
## [1] "Mon nom est Luigi et mon nom de famille est Fenway"
```

Les fonctions comme `paste()` peuvent prendre de nombreuses valeurs car elles ont un argument spécial, une ellipse : â€¦ Si vous consultez le fichier d'aide de la fonction paste, vous verrez ceci :

Arguments

...	one or more R objects, to be converted to character vectors.

Un autre argument utile pour `paste()` est appelé `sep`. Il indique à R quel caractère utiliser pour séparer les termes :

```
paste("Luigi", "Fenway", sep = "-")
```

```
## [1] "Luigi-Fenway"
```

Fonctions imbriquées

La sortie d'une fonction peut être immédiatement reprise par une autre fonction. C'est ce qu'on appelle l'imbrication de fonctions.

Par exemple, la fonction `tolower()` convertit une chaîne de caractères en minuscules.

```
tolower("LUIGI")
```

```
## [1] "luigi"
```

Vous pouvez prendre la sortie de cette fonction et la passer directement dans une autre fonction :

```
paste(tolower("LUIGI"), "is my name")
```

```
## [1] "luigi is my name"
```

Sans cette option d'imbrication, vous devriez assigner un objet intermédiaire :

```
mon_nom_de_minceur <- tolower("LUIGI")
paste(my_lowercase_name, "is my name")
```

```
## [1] "luigi is my name"
```

L'imbrication des fonctions sera bientôt très utile.

Question 8

Les morceaux de code ci-dessous sont tous des exemples d'imbrication de fonctions. Une des lignes contient une erreur. De quelle ligne s'agit-il, et quelle est l'erreur ?

```
sqrt(head(women))
```

```
paste(sqrt(9), "plus 1 est", sqrt(16))
```

```
sqrt(tolower("LUIGI"))
```

Paquets

Comme nous l'avons mentionné précédemment, R est merveilleux parce qu'il est extensible par l'utilisateur : n'importe qui peut créer un *package* logiciel qui ajoute de nouvelles fonctionnalités. La plupart de la puissance de R provient de ces paquets.

Dans la leçon précédente, vous avez installé et chargé le paquet {highcharter} en utilisant les fonctions `install.packages()` et `library()`. Nous allons maintenant en apprendre un peu plus sur les paquets.

Un premier exemple : le paquet {tableone}.

Installons et utilisons maintenant un autre paquetage R, appelé `tableone` :

```
install.packages("tableone")
```

```
library(tableone)
```

Notez que vous n'avez besoin d'installer un paquet qu'une seule fois, mais que vous devez le charger avec `library()` chaque fois que vous voulez l'utiliser. Cela signifie que vous

devriez généralement exécuter la ligne `install.packages()` directement depuis la console, plutôt que de la taper dans votre script.

Le paquetage facilite la construction du "Tableau 1", c'est-à-dire un tableau avec les caractéristiques de l'échantillon de l'étude que l'on trouve couramment dans les articles de recherche biomédicale.

Le cas d'utilisation le plus simple est le résumé de l'ensemble des données. Vous pouvez simplement introduire le cadre de données dans l'argument `data` de la fonction principale `CreateTableOne()`.

```
CreateTableOne(données = ebola_data)
```

```
## Error in CreateTableOne(données = ebola_data): unused argument (données =
ebola_data)
```

Vous pouvez voir qu'il y a 200 patients dans cet ensemble de données, que l'âge moyen est de 33 ans et que 38% de l'échantillon est masculin, entre autres détails.

Très cool ! (Un problème est que le package suppose que les variables de date sont catégoriques ; à cause de cela, le tableau de sortie est beaucoup trop long).

Le but de cette démonstration de {tableone} est de vous montrer que les packages R externes sont très puissants. C'est l'une des grandes forces du travail avec R, un langage open-source avec un écosystème dynamique de contributeurs. Des milliers de personnes travaillent en ce moment même sur des paquets qui pourraient vous être utiles un jour.

Vous pouvez chercher sur Google "Cool R packages" et parcourir les réponses si vous êtes désireux d'en savoir plus sur d'autres paquets R.

Vous avez peut-être remarqué que nous englobons les noms de paquets entre accolades, par exemple {tableone}. Il s'agit simplement d'une convention de style entre les utilisateurs/enseignants de R. Les accolades ne signifient rien.

Significatifs complets

Le *signifiant complet* d'une fonction comprend à la fois le nom du paquet et le nom de la fonction : `package::function()`.

Ainsi, par exemple, au lieu d'écrire :

```
CreateTableOne(data = ebola_data)
```

on pourrait écrire cette fonction avec son signifiant complet, `package::function()` :

```
tableone::CreateTableOne(data = ebola_data)
```

En général, vous n'avez pas besoin d'utiliser ces signifiants complets dans vos scripts. Mais il existe des situations où cela peut être utile :

La raison la plus courante est que vous voulez indiquer très clairement de quel paquetage provient une fonction.

Deuxièmement, vous voulez parfois éviter d'avoir à exécuter `library(package)` avant d'accéder aux fonctions d'un package. C'est-à-dire que vous voulez utiliser une fonction d'un paquetage sans avoir à charger ce paquetage depuis la bibliothèque. Dans ce cas, vous pouvez utiliser la syntaxe complète du signifiant.

Voici donc ce qui suit :

```
tableone::CreateTableOne(data = ebola_data)
```

est équivalent à :

```
library(tableone)
CreateTableOne(données = ebola_data)
```

Question 9

Considérez le code ci-dessous :

```
tableone::CreateTableOne(data = ebola_data)
```

Laquelle des propositions suivantes est une interprétation correcte de la signification de ce code ?

A. Le code applique la fonction `CreateTableOne` du package {tableone} sur l'objet `ebola_data`.

B. Le code applique l'argument `CreateTableOne` de la fonction {tableone} sur le package `ebola_data`.

C. Le code applique la fonction `CreateTableOne` du package {tableone} sur le package `ebola_data`.

pacman::p_load()

Plutôt que d'utiliser deux fonctions distinctes, `install.packages()` puis `library()`, pour installer puis charger des paquets, vous pouvez utiliser une seule fonction, `p_load()`, du paquet {pacman} pour installer automatiquement un paquet s'il n'est pas encore installé, *et* charger le paquet. Nous encourageons cette approche dans la suite de ce cours.

Installez {pacman} maintenant en exécutant ceci dans votre console :

```
install.packages("pacman")
```

À partir de maintenant, quand un nouveau paquet vous est présenté, vous pouvez simplement utiliser `pacman::p_load(nom_du_paquet)` pour installer et charger le paquet :

Essayez ceci maintenant pour le paquet `outbreaks`, que nous utiliserons bientôt :

```
pacman::p_load(outbreaks)
```

Maintenant, nous avons un petit problème. La merveilleuse fonction `pacman::p_load()` installe et charge automatiquement les paquets.

Mais il serait bien d'avoir un code qui installe automatiquement le paquet {pacman} lui-même, s'il est manquant sur l'ordinateur de l'utilisateur.

Mais si vous mettez la ligne `install.packages()` dans un script, comme ceci :

```
install.packages("pacman")
pacman::p_load(ici, rmarkdown)
```

vous allez perdre beaucoup de temps. En effet, chaque fois qu'un utilisateur ouvre et exécute un script, il *réinstalle* {pacman}, ce qui peut prendre un certain temps. A la place, nous avons besoin d'un code qui commence par *vérifier si pacman n'est pas encore installé* et l'installe si ce n'est pas le cas.

Nous pouvons faire cela avec le code suivant :

```
if(!require(pacman)) install.packages("pacman")
```

Vous n'avez pas besoin de le comprendre pour le moment, car il utilise une syntaxe que vous n'avez pas encore apprise. Notez simplement que dans les prochains chapitres, nous commencerons souvent un script avec un code comme celui-ci :

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(here, rmarkdown)
```

La première ligne va installer {pacman} s'il n'est pas encore installé. La deuxième ligne utilisera la fonction `p_load()` de {pacman} pour charger les paquets restants (et `pacman::p_load()` installe tous les paquets qui ne sont pas encore installés).

Ouf ! J'espère que votre tête est toujours intacte.

Question 10

Au début d'un script R, nous aimerions installer et charger le paquet appelé {janitor}. Lequel des morceaux de code suivants est-il recommandé d'inclure dans votre script ?

A.

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(janitor)
```

B.

```
install.packages("janitor")
library(janitor)
```

C.

```
install.packages("janitor")
pacman::p_load(gardien)
```

Conclusion

Avec votre nouvelle connaissance des objets R, des fonctions R et des paquets d'où proviennent les fonctions, vous êtes prêt, croyez-le ou non, à faire de l'analyse de données de base dans R. Nous allons nous y plonger dès la prochaine leçon. À bientôt !

Réponses {.non listé .non numéroté}

1. Vrai.
2. Le signe de division est évalué en premier.
3. La réponse est C. Le code "2 + 2 + 2" est évalué avant d'être stocké dans l'objet. 4.
4. a. La valeur est 1. Le code est évalué à `9-8`.

   b. table(ebola_data$district)
5. a. Vous ne pouvez pas ajouter deux chaînes de caractères. L'addition ne fonctionne que pour les nombres.

   b. Mon_1er_nom`est tapé avec le chiffre 1 initialement, mais dans la commande`paste()`, il est tapé avec la lettre "l".
6. La troisième ligne est la seule ligne avec un nom d'objet valide : `top_20_rows`.
7. La dernière ligne, `head(6, women)`, est invalide parce que les arguments sont dans le mauvais ordre et qu'ils ne sont pas nommés.
8. Le troisième morceau de code a un problème. Il tente de trouver la racine carrée d'un caractère, ce qui est impossible.

9. La première ligne, A, est l'interprétation correcte.

10. Le premier morceau de code est la manière recommandée d'installer et de charger le paquet {janitor}.

Contributeurs

Les membres suivants de l'équipe ont contribué à cette leçon :

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

LAMECK AGASA

Statistician/Data Scientist

OLIVIA KEISER

Head of division of Infectious Diseases and Mathematical Modelling, University of Geneva

Références

Certains éléments de cette leçon ont été adaptés à partir des sources suivantes :

- "File:Apple fonction de découpage.png." *Wikimedia Commons, the free media repository*. 1 Oct 2021, 04:26 UTC. 20 Mar 2022, 17:27 <https://commons.wikimedia.org/w/index.php?title=File:Apple_slicing_function.png&oldid=594767630>.

- "PsyteachR | Data Skills for Reproducible Research." 2021. Github.io. 2021. https://psyteachr.github.io/reprores-v2/index.html.

- Douglas, Alex, Deon Roos, Francesca Mancini, Ana Couto et David Lusseau. 2022. "Une introduction à R." Intro2r.com. 27 janvier 2022. https://intro2r.com/.