Version 0.0 de type "*work in progress*"
Si vous avez trouvé un bug : denis.poinsot@univ-rennes1.fr

Je n'y connais rien mais je programme en R !

Remerciements :

Merci à Ross Ihaka et Robert Gentleman[1] ainsi qu'au *R Core Team* d'avoir mis au service de l'humanité le fabuleux outil gratuit et d'une puissance infinie qu'est R, et merci à Maxime Hervé pour son sens de la pédagogie lorsque ce fabuleux outil gratuit d'une puissance infinie se comporte d'une manière un peu trop gratuite et puissante pour moi.

[1]Ihaka R & R Gentleman, 1996. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics* **5**:299-314. L'article par lequel la révolution à commencé. Déjà cité près de *7000 fois*, ce qui est au même niveau que l'article de *Science* de 1983 dans lequel Françoise Barré-Sinoussi et l'équipe du Pr Luc Montagner décrivent la découverte du virus du SIDA...

Préambule

Ce document est tellement frais qu'il est en réalité en cours d'écriture et sera amélioré en fonction de vos questions et remarques.

Il résulte d'une bonne remarque (de plus) faite par Maxime Hervé, selon laquelle vous demander de programmer même une *petite partie* d'un script R serait trop dur si vous n'étiez guidés pas à pas.

Je connais très peu de choses en programmation donc ce document sera basique mais comme il s'agit de parler uniquement des choses simples, ça devrait suffire. De toute manière, n'oubliez jamais que si vous voulez en apprendre plus sur la fonction `machin()` de R il vous suffit d'écrire :

```
? machin()
```

Et l'aide s'ouvrira automatiquement.

Toutefois, je sais par dure expérience que l'aide de R n'est pas toujours très accueillante pour les débutants, et même pour les moins débutants, d'où l'intérêt probable des documents comme celui-ci.

Comme ce texte est centré sur les besoins spécifiques du défi de la *Penvins Cup*, je vais évidemment me limiter au strict nécessaire dans ce cadre. Le bon enseignant n'est il pas celui qui déploie des efforts surhumains pour cacher la complexité du monde à ses élèves tant qu'il n'est pas nécessaire qu'ils la connaissent ?

Si ça n'était pas le cas, on apprendrait le calcul matriciel en CP, vous imaginez le tableau ?

Et comme les demandes des débutants commencent en général par "*Pourquoi*..." ou "*Comment*...", c'est sous cette forme que les problèmes seront traités.

C'est parti.

Sommaire

1. Pourquoi apprendre à programmer en biologie ? 4
2. Pourquoi faire ses premiers pas en programmation *avec R* ? 5
3. Comment afficher quelque chose ? 8
   3.1 Comment afficher un message fixe ? 8
   3.2 Comment afficher un message utilisant des variables ? 9
   3.3 Comment afficher les chaînes de caractères *de votre tableau de données* ? 10
4. Comment faire *répéter* des instructions ? 11
5. Comment faire quelque chose *si* une condition est remplie ? 12
6. Comment tester et combiner les conditions en général ? 14
   6.1 Comment tester si une condition est remplie ? 14
   6.2 Comment tester si la condition A *n'est pas* remplie ? 16
   6.3 Comment tester si les conditions A *et* B sont remplies ? 16
   6.4 Comment tester si les conditions A *ou* B sont remplies ? 17
   6.5 Comment tester si *aucune* des deux conditions A et B n'est remplie ? 17
7. Comment manipuler les *colonnes* d'un tableau ? 17
   7.1 Comment *compter les colonnes* d'un tableau ? 18
   7.2 Comment *afficher le nom des colonnes* d'un tableau ? 18
   7.3 Comment tester *globalement* si les noms des colonnes sont identiques à une référence ? 18
   7.4 Comment afficher *uniquement* les noms de colonnes *incorrects* ? 19
   7.5 Comment tester le *type de données* stockées dans une colonne ? 19
8. Comment manipuler les *cases* d'un tableau ? 21
   8.1 Comment *accéder à la $i^{ème}$ ligne* de la colonne A ? 21
   8.2 Comment tester si une case contient une chose précise (chiffre, texte, NA) 22
   8.3 Comment tester si le contenu d'une case est bien situé entre deux valeurs ? 22
   8.4 Comment *remplacer automatiquement* une valeur aberrante par NA ? 22
   8.5 Comment calculer un *ratio entre deux colonnes numériques* ? 23
9. Qu'est-ce qu'une fonction et comment en créer une ? 24

1. Pourquoi apprendre à programmer en biologie ?

La programmation sert à une infinité de choses mais en particulier à : (i) automatiser les tâches répétitives, (ii) réaliser rapidement ce qui vous prendrait une éternité et (iii) prendre des décisions *pertinentes* automatiquement sans que vous ayez à intervenir.

Les exemples correspondants sont (i) l'automate qui bientôt demandera à notre place "Pourrais-je voir *l'autre* côté de votre carte, s'il vous plait ?" au 570 étudiants de première année ayant posé leur carte d'étudiant sur leur table en laissant visible la face orange sans intérêt qui dit simplement "université de Rennes 1", (ii) l'algorithme de tri qui scanne la gigantesque base de données de Google en une fraction de seconde selon les mots clés "Justin"+"Bieber" ou "R"+"help", et (iii) le processeur qui déclenche votre airbag sans solliciter votre autorisation par écrit en trois exemplaires si un choc violent est détecté.

Plus proche de vos préoccupations immédiates, supposons au hasard que vous disposiez d'une dizaine de fichiers .txt contenant des données que vous souhaitez analyser, et que vous ayez à force d'essais-(messages d')erreurs réussi à trouver les instructions de R qui vous permettent de charger un de ces fichiers dans R, d'afficher un nuage de points, d'y tracer une droite d'allométrie, de calculer la pente de la droite et de tester si elle est significativement différente de 1. Ne serait-il pas intéressant d'en faire un petit script qui ferait tout ça automatiquement sur les neuf autres fichiers d'un coup ? Ou même, sans en demander tant, qui automatiserait l'analyse *d'un* fichier, de manière à ce que vous ayez juste à saisir :

```
allo("Ta70ind.txt")
```

Pour que, une fraction de seconde plus tard, le nuage de points et la droite d'allométrie apparaissent, assortis d'un message de conclusion du type :

```
Ta70 : allométrie majorante significative (a=1.2, Z=2.7, n=928, P<0.001)
```

Ce rêve est beaucoup moins hors de portée que vous l'imaginez — en tout cas en utilisant R. Rien que pour la partie graphique, il vous faudrait peut être des mois d'apprentissage avant de savoir faire un truc pareil en Java ou en C++, sans parler de l'assembleur, voyez plus bas.

2. Pourquoi faire ses premiers pas en programmation *avec R* ?

Parce que c'est sacrément plus facile que dans d'autres langages qui font pourtant la même chose !

Le nombre de langages informatiques différents est absolument stupéfiant. On en dénombre plusieurs *milliers*, ce qui signifie qu'il y en a probablement davantage aujourd'hui que de langues humaines couramment parlées dans le monde ! Ce nombre indique évidemment que le langage idéal n'existe pas. Mon propos ici est de vous montrer que R est un très bon langage pour *s'initier aux rudiments* de la programmation[2]. Que les programmeurs en Java et C++ lèvent les yeux au ciel tant qu'ils veulent, je m'en contrefiche.

[2] Un autre excellent choix serait Python, mais comme vous êtes déjà tous utilisateurs de R...

Il y a en informatique un rituel sacré qui consiste à présenter un langage en commençant par le listing du programme qui permet d'afficher à l'écran la phrase "Hello world!".

Histoire de savourer le monde merveilleux dans lequel vous vivez maintenant, bande de petits veinards, voyez à quoi ressemblait ce programme élémentaire dans le seul langage informatique que pouvaient utiliser les tout premiers hackers[3], un langage nommé *assembleur.* L'exemple ci-dessous écrit "Hello World!" en utilisant l'assembleur X86 sous DOS. Régalez-vous :

```
cseg segment
assume cs:cseg, ds:cseg
org 100h
main proc
jmp debut
mess db 'Hello world!$'
debut:
mov dx, offset mess
mov ah, 9
int 21h
ret
main endp
cseg ends
end main
```

Intuitif, n'est ce pas ? Et puis tellement *concis* !

Aujourd'hui ça va nettement mieux grâce aux langages dits de "haut niveau", mais ça ne se lit toujours pas comme du Voltaire quand même. Voici le programme "Hello World!" dans deux des langages de programmation modernes les plus utilisés : Java et C++.

En Java, ça nous donne ceci :

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Vous n'auriez peut être pas deviné spontanément, hein ? Moi non plus, je ne parle pas un mot de Java.

En C++, ça ne vaut guère mieux (et je n'y toucherais pas même avec de longues pincettes) :

```
#include <iostream>

int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

Et maintenant (roulement de tambour), voici le programme "Hello world!" en R :

[3] Non, les hackers ne sont *pas* les pirates qui volent les codes de carte bleue et sont la plaie du web, bien au contraire. Lisez "*How to become a hacker*", par Eric S. Raymond *alias* esr, en ligne ici : http://www.catb.org/esr/faqs/hacker-howto.html

```
"Hello, world!"
```

Alors, quitte à vous initier aux bases élémentaires d'un langage de programmation, préférez-vous l'assembleur X86, le Java, le C++ ou bien R ?

NB. Pour les curieux qui veulent découvrir mille et mille manière d'afficher "Hello, World!" dans tous les langages informatiques de la terre, voyez ici : http://fr.wikipedia.org/wiki/Liste_de_programme_Hello_world

Dans le reste de ce document vous allez trouver sur fond `bleu` des tas de mini-scripts que vous pouvez recopier dans R et constater ce qu'ils font (vos erreurs de frappe vous apprendront d'ailleurs beaucoup sur l'importance des détails), mais aussi modifier pour constater les conséquences. C'est comme ça qu'on apprend.

Si vous voulez que le comportement de R soit rigoureusement le même que dans ce document, il vous faudra simplement écrire le code suivant en préambule : il crée un tableau `Kaamelot` de trois colonnes et quatre lignes et *l'attache* via la fonction `attach()` pour que les noms des colonnes soient ensuite directement accessibles sans devoir saisir à chaque fois `Kaamelot$nomColonne` :

Alors ouvrez R, allez dans **`fichier/nouveau script`** et écrivez-y les lignes suivantes (pas la peine d'écrire les commentaires qui sont en noir et précédés de **`#`**:

```
id=c("Arthur", "Guenièvre", "Perceval", "Bohort") #un "vecteur"
role=c("roi", "reine", "chevalier", "chevalier") #un autre vecteur
courage=c(500, NA, 100, 0) #oui, c'est bien un troisième vecteur
Kaamelot=data.frame(id, role, courage) #forme un tableau avec les vecteurs
rm(list=c("id", "role", "courage")) #supprime les vecteurs d'origine
attach(Kaamelot) #rend les noms des colonnes accessibles directement
```

Ensuite, deux options :

A la souris : édition/exécuter tout
Au clavier : Ctrl-A pour sélectionner tout, Ctrl-R pour exécuter dans R

La commande Ctrl-R va rapidement vous devenir familière car par défaut elle exécute *la ligne sur laquelle se trouve le curseur au moment ou vous la lancez*, ce qui est pratique pour traquer les bugs. Ca n'est pas ce qu'elle fait ici parce qu'on avait sélectionné auparavant un bloc de texte (ici, la totalité) donc elle exécute toutes les lignes du bloc sélectionnées.

Si ensuite vous demandez :

```
Kaamelot
```

vous devriez obtenir ceci :

```
         id      role courage
1    Arthur       roi     500
2 Guenièvre     reine      NA
3  Perceval chevalier     100
4    Bohort chevalier       0
```

Nous sommes prêts à commencer.

3. Comment afficher quelque chose ?

3.1 Comment afficher un message fixe ?

Eh bien... vous le savez déjà, non ?

```
"Hello, world!"
```

La réponse est alors :

```
[1] "Hello, world!"
```

Notre instruction `"hello world!"` est en réalité l'abréviation de `print("Hello world")`, qui peut aussi s'appliquer (sans guillemets) aux objets.

```
a="un objet quelconque"
print("a") #avec guillemets
[1] "a"

print(a) #sans guillemets
[1] "un objet quelconque"

a #la version abrégée de print(a)
[1] "un objet quelconque"
```

R fourmille d'abréviations de ce type, qui facilitent la vie *lorsqu'on les connait*, mais la compliquent au contraire quand on ne les connait pas et qu'on tombe dessus dans le script de quelqu'un d'autre, par exemple dans l'aide de R...

Notez que R donne en début de ligne le numéro de l'élément qui commence la ligne, ce qui est pratique pour se repérer le long de longues listes d'éléments qui couvrent plusieurs lignes, mais comme ici il y a un seul élément on obtient `[1]`. Vous observerez aussi que la réponse en question apparaît *entre guillemets* (on aimerait parfois s'en passer) et que R *va à la ligne* automatiquement après le message (c'est en général ce que vous voulez, mais pas toujours).

C'est pourquoi je vous montre tout de suite la fonction d'affichage qui permet d'obtenir de beaux messages, elle porte le nom d'un chat, c'est la fonction `cat()` pour *concatenate* (faire une chaîne) :

```
cat("Hello, world!")
Hello, world!>
```

Le `[1]` du début et les guillemets ont disparu (c'est donc plus agréable à lire), par contre il n'y a pas eu de retour à la ligne, le prompt `>` est juste derrière le texte. Comme en général ça n'est pas désirable, voici comment ajouter l'instruction "retour", par la combinaison "antislash n" : `\n`

```
cat("Hello, world! \n")
Hello, world!
>
```

Notez que le retour à la ligne `\n` s'utilise *à l'intérieur* des guillemets.

En parlant des guillemets, comment afficher des guillemets *dans* un message ?

Une première idée serait évidemment de... mettre des guillemets :

```
cat("je vais mettre "ceci" entre guillemets \n")
Erreur : symbole inattendu(e) dans "cat("je vais mettre "ceci"
>
```

Visiblement, ça ne lui a pas plu. Plus exactement, ça n'a pas plu au *parser* (prononcer "parseur", du verbe *to parse* : analyser, décomposer). Le *parser* est la partie de R qui analyse vos instructions symbole par symbole pour en tirer du sens. Il est extrêmement discipliné et absolument intraitable sur le respect de la syntaxe. Si vous pensez que je suis trop pointilleux sur la forme, essayez donc de négocier avec un *parser*.

Ici, le *parser* trouve des guillemets s'ouvrant juste avant `je`, il attend donc des guillemets fermants, et il les trouve juste avant `ceci`. Il note donc la chaîne `"je vais mettre"` et attend ensuite soit une *virgule*, qui annoncerait un autre élément à ajouter à la chaîne, soit la fin de la fonction `cat()` c'est à dire une *parenthèse fermante*. En revanche, la chose qu'il n'attend *pas* c'est de tomber directement sur un *caractère*, or il trouve le `c` de `ceci`, d'où le message d'erreur.

Pour contourner la difficulté, il y a deux méthodes.

La première est d'utiliser *deux* sortes de guillemets. Par exemple, des doubles guillemets (") pour l'extérieur et des simples (') pour l'intérieur (l'inverse marche aussi):

```
cat("je vais mettre 'ceci' entre guillemets \n")
je vais mettre 'ceci' entre guillemets
>
```

La seconde méthode est de signaler au *parser* que les guillemets internes doivent être interprétés de manière spéciale, c'est à dire ici en fait de manière quelconque (= comme n'importe quel autre caractère). Cette instruction se donne par la barre *antislash* `\` (Alt Gr+8), que nous avons déjà utilisée pour l'instruction `\n` qui signale au *parser* que ce `n` là est spécial : il demande un retour à la ligne.

```
cat("je vais mettre \"ceci\" entre guillemets \n")
je vais mettre "ceci" entre guillemets
>
```

3.2 Comment afficher un message utilisant des variables ?

Le véritable intérêt de la fonction `cat()` est de faire des chaînes pour inclure dans vos messages des données que R ira piocher dans vos données. Il faudra simplement séparer les composantes de la chaîne par des virgules. Votre texte sera entre guillemets, les données issues de votre fichier seront désignées par l'objet ou la fraction d'objet qui les contient.

Exemples :

```
a=5
cat("Il vous reste", a, "vies \n")
Il vous reste 5 vies
>
```

Le courage de Bohort étant stocké dans la ligne 4 de la colonne `courage` on peut écrire, puisque le tableau `Kaamelot` est attaché :

```
cat("Le courage de Bohort est :", courage[4], "\n")
Le courage de Bohort est : 0
>
```

Pour les amateurs de la série Kaamelot, il nous suffira d'évoquer l'épisode culte du *lapin adulte*.

Plus compliqué :

```
cat("Le courage moyen est :", mean(courage, na.rm=TRUE), "\n")
Le courage moyen est : 200
```

Il fallait ajouter l'option `na.rm=TRUE` (qui élimine les NA) à cause du courage de Guenièvre, qui est NA, sinon il se passerait ceci :

```
cat("Le courage moyen (zut, il y a un NA !) est :", mean(courage), "\n")
Le courage moyen (zut, il y a un NA !) est : NA
```

R ne peut en toute logique calculer une moyenne s'il manque une donnée.

3.3 Comment afficher les chaînes de caractères *de votre tableau de données* ?

Accrochez vous, ça va secouer un peu (si vous êtes fatigués, allez plutôt vous reposer, vous lirez ça demain).

Le *deuxième* personnage de la colonne `id` est `Guenièvre`. Comparez alors le résultat des instructions `print()` et `cat()` pour afficher le contenu de la case `id[2]` :

```
print(id[2])
[1] Guenièvre
Levels: Arthur Bohort Guenièvre Perceval
>
```

Le *post scriptum* (`levels` ?) est louche, mais au moins j'obtiens ce que j'attendais : `Guenièvre.`

```
cat(id[2], "\n")
3
>
```

Hein ? Comment ça `3` ?? Il n'y a du tout écrit `3` dans la case `id[2]`, il y a écrit `Guenièvre` !

Bon, là j'avoue qu'on entre un peu dans le dur. Lorsque vous chargez votre tableau, R considère par défaut que les colonnes contenant des chaines de caractères (et non des chiffres) sont des *facteurs*. En stat, un facteur sera par exemple le sexe, le mode ou l'espèce, et vous savez qu'en particulier l'ANOVA, l'ANCOVA et les glm() en font un usage intéressant. Or, pour des raisons qui m'échappent, la fonction cat() ne se comporte pas comme on pourrait s'y attendre quand elle est face à un facteur.

En fait, cat vous a donné le *niveau du facteur* qui se trouve dans la case `id[2]`. Il se trouve simplement qu'il y a quatre niveaux ici. Ils étaient affichés en *post-scriptum* par la fonction `print()`

mais vous pouvez y avoir accès directement en demandant "quels sont les *niveaux* du facteur `id`", ainsi :

```
> levels(id)
[1] "Arthur"    "Bohort"    "Guenièvre" "Perceval"
>
```

Le *troisième* niveau est bien `"Guenièvre"` (les niveaux d'un facteur sont tout simplement classés dans l'ordre alphabétique).

Que faire ?

Si vous voulez le *contenu* de la case d'une colonne de type `factor`, il vous faudra soit utiliser une instruction `print()`, soit indiquer à `cat()` que c'est la *chaine de caractère* qui vous intéresse. C'est le rôle de la fonction `as.character()`, qui va chercher la chaine de caractère correspondant au niveau du facteur de la case. Ainsi :

```
> cat(as.character(id[2]), "\n")
Guenièvre
>
```

Un peu compliqué hein ? Oui, je trouve aussi.

4. Comment faire répéter des instructions ?

Ceci est fondamental, car une grande partie de l'activité d'un programme est souvent de répéter n fois la même opération à toute vitesse. Cette répétition est obtenue en créant une *boucle*, qui va répéter l'opération tant qu'un *compteur* de boucles n'a pas atteint la valeur finale désirée. R possède des fonctionnalités puissantes qui permettent *d'éviter* les boucles dans bien des cas, mais comme l'objet est ici *d'apprendre* à faire une boucle, voyons la plus courante, la boucle `for`.

La syntaxe générale d'une boucle `for` est :

```
for ( <nom d'un compteur> in <valeur de départ> : <valeur de fin>) {
        <instruction 1>
        <instruction 2>
        <instruction 3>
        <etc.>
}
```

Le cas le plus élémentaire est celui dans lequel vous connaissez d'avance le nombre de boucles à effectuer, par exemple, si vous voulez *trois* répétitions, écrivez :

```
for(i in 1:3){
      cat("Et un, et deux, et trois zéro ! \n")
}
Et un, et deux, et trois zéro !
Et un, et deux, et trois zéro !
Et un, et deux, et trois zéro !
>
```

Il est cependant possible que vous ne sachiez pas à l'avance combien de boucles il va falloir effectuer. Pas de problème, ce nombre peut être ajusté aux besoins, à condition que l'information soit disponible dans les données à traiter.

Supposons une boucle qui va afficher toutes les valeurs d'une série de données. Votre programme ne peut deviner à l'avance combien de données il y a dans la série. Par contre, il peut déterminer cette longueur par la fonction `length()`, et vous savez qu'on peut désigner le $i^{ème}$ élément d'une série nommée machin grâce à la notation machin[i], donc :

```
for(i in 1:length(courage)){
      cat(courage[i], "\n")
}
500
NA
100
0
>
```

On reconnaît des courages d'Arthur, Guenièvre, Perceval et Bohort le lapinophobe.

Eh, mine de rien nous venons d'écrire un programme qui affiche le contenu d'une liste d'éléments en allant les chercher un par un !

Mais bien sûr, l'équivalent de cette fonction basique existe déjà, c'était juste un exemple ! Dans R, pour afficher le contenu d'un objet il suffit d'appeler son nom : `courage`, puisque je vous rappelle que c'est l'abréviation de l'instruction `print(courage)`

```
courage
[1] 500  NA 100   0
```

5. Comment faire quelque chose *si* une condition est remplie ?

Ceci se nomme un *test conditionnel*, et c'est la deuxième des choses les plus fondamentales en programmation, avec la capacité de répéter des instructions. On peut même dire sans exagérer qu'un programme n'est qu'une suite plus ou moins complexe de boucles et de tests. Si vous savez faire les deux, vous savez programmer. Les bons programmeurs sont ceux qui savent mieux que les autres comment organiser leurs boucles et leurs tests, et même comment les *éviter* lorsque c'est possible.

Voyons donc le test `if(){}else{}`, dont la syntaxe (et la mise en page habituelle) est :

```
if (<condition à vérifier>){
        <liste d'instructions à exécuter si la condition est vérifiée >
} else {
        <liste d'instructions à exécuter si condition n'est pas vérifiée>
        <notez que cette partie else {} est optionnelle>
   }
```

L'indentation du texte ci-dessus peut vous paraître hideuse et chaotique mais elle est au contraire stratégique pour éviter les erreurs. En effet, toute accolade ouverte doit être fermée, et il est extrêmement facile quand on emboite les tests et les boucles les uns dans les autres de ne plus savoir où on en est. Aussi, la logique est la suivante : *alignez votre accolade fermante sous le* ***nom***

de l'instruction qui l'a ouverte. C'est pourquoi l'accolade fermante du `if` est alignée sous le mot `if`, alors que l'accolade fermante du `else` est sous le mot `else`. C'est aussi pour permettre cette vérification visuelle qu'à chaque fois le texte est décalé vers la droite, ce qui fait apparaître des blocs homogènes.

Si vous voulez faire des scripts d'une certaine longueur sans souffrances inutiles (je n'ai pas dit sans souffrances, j'ai dit sans souffrances *inutiles*), croyez-en ma pourtant *minuscule* expérience de programmation : soignez la présentation.

En fait, si vous voulez que vos scripts dépassent une certaine taille, utilisez impérativement un « traitement de texte pour geek » (ou IDE pour *Integrated Development Environment*) comme le très simple Geany, qui gèrera pour vous beaucoup de ces problèmes d'alignement et de parenthèses mal fermées) :

http://www.geany.org/

En tout cas, si vous voulez éviter de vous arracher autant de cheveux que moi sur le test `if`, *notez bien* la partie en rouge : le mot `else` doit impérativement se trouver sur la même ligne que l'accolade `}` qui finit la partie `if`. Si vous placez le mot `else` sur la ligne d'après, vous obtiendrez un message d'erreur complètement abscons :

```
if(1<2){print("ben oui, normal")}
      else {print("plaît-il?")}
Erreur : 'else' inattendu(e) dans "else"
>
```

Eh ?

Voyons plutôt l'exemple d'un test conditionnel correct :

```
if(1>2){
      cat("Scoop ! Un est supérieur à deux ! \n")

} else {
      cat("Rien de nouveau sous le soleil, \n")
      cat("deux valent plus qu'un seul \n")
      cat("tous les amoureux vous le diront... \n")
   }
Rien de nouveau sous le soleil,
deux valent plus qu'un seul
tous les amoureux vous le diront...
>
```

Vous noterez que j'ai sauté une ligne entre le bloc d'instruction du `if` (ici réduit à une ligne) et le début du `else`. Cela est non seulement autorisé (R ignore les lignes vides) mais bienvenu pour aérer le code et le rendre plus lisible. N'hésitez jamais à sauter une ligne dans vos scripts entre deux unités logiques, de la même manière que vous le feriez dans un texte écrit entre deux idées.

C'est bien joli mais tester si 1>2 n'a vraiment aucun intérêt ! Certes, mais le principe peut être étendu à tout ce que vous voulez. Poursuivons pour l'instant avec un type d'action drastique à effectuer sous certaines conditions

Comment stopper le programme si quelque chose ne va pas ?

En utilisant l'instruction `stop()`. Cette instruction brutale est à utiliser dans les cas extrêmes (l'erreur constatée n'est pas réparable ou contournable par le programme lui même). Par défaut, la fonction stop écrit `"Erreur :"` et rien d'autre, mais elle suppose que vous allez écrire dans sa parenthèse un message qui explique pourquoi le programme a été stoppé, donc utilisez évidemment cette option et *soyez clair*, rien n'est plus exaspérant pour l'utilisateur qu'un message d'erreur incompréhensible.

Exemple : Voici comment stopper le programme immédiatement si l'objet `Kaamelot` n'a pas la structure d'un tableau (`data.frame`).

```
if(class(Kaamelot)!="data.frame"){ #NB: != signifie "différent de"
      stop("format incorrect : l'objet doit être de classe data.frame")
}
>
```

Ici, aucun message d'erreur puisque la structure de `mydata` est bien celle d'un tableau, comme vous pouvez le vérifier facilement avec la fonction `class()` qui indique le type d'un d'objet :

```
class(Kaamelot)
[1] "data.frame"
>
```

6. Comment tester et combiner les conditions en général ?

6.1 Comment tester si une condition est remplie ?

Pour tester dans R si quelque chose est vrai, il suffit d'écrire une affirmation évaluable de manière mathématique ou logique. Le *parser* va alors faire son travail et R répondra `TRUE` si elle est (mathématiquement ou logiquement) correcte, `FALSE` dans le cas contraire.

Voici les opérateurs à connaître :

opérateur	signification	opérateur	signification
`>`	supérieur à	`==`	égal à
`<`	inférieur à	`!=`	différent de
`>=`	supérieur ou égal à	`&`	et
`<=`	inférieur ou égal à	`|`	ou

Au cas où vous la chercheriez, la barre verticale `|` est obtenue par la combinaison Alt Gr+6.

Exemples faciles :

```
1>2
[1] FALSE
```

```
1<=1
[1] TRUE
```

Attention, vous allez souvent vous tromper au début dans la manipulation du *double* signe égal (`==`) car par réflexe vous écrirez un seul signe égal (`=`) pour tester les égalités (et c'est un réflexe qui a la vie dure, croyez moi).

Or, il faut comprendre que dans R, le signe simple égal `=` est équivalent à l'instruction "flèche gauche" `<-`. C'est donc un opérateur qui *affecte* une valeur à un objet qui se trouve à gauche du signe égal.

`a=5` signifie :

Si l'objet `a` n'existait pas, il est créé et contient maintenant uniquement : 5.
Si l'objet `a` existait, *j'écrase son contenu et le remplace sans remède par* : 5.

`a==b` signifie :

`a` et `b` sont ils égaux ? (mais selon la structure des objets, la réponse peut surprendre)

Donc :

```
1==2
[1] FALSE
```

Mais :

```
1=2
Erreur dans 1 = 2 : membre gauche de l'assignation (do_set) incorrect
```

Eh oui : le simple égal signifie « mettre `2` dans *l'objet* `1` », or 1 est un chiffre et non un objet.

En manipulant des objets :

```
a="le résultat de douze heures de travail épuisant dans R"
b="rien du tout"
a==b
[1] FALSE
```

```
a=b
```

Aucune réponse. Pourtant il s'est passé un truc que vous ne vouliez probablement pas :

```
a
[1] "rien du tout"
```

Oups ! Le produit de vos douze heures de travail a été écrasé pour toujours... (*non*, il n'y a pas de "undo" ni de "pomme-Z" dans R).

Et maintenant, plus compliqué. Si vous testez l'égalité `A==B` entre deux objets complexes comme des colonnes (c'est à dire des listes de chiffres ou de chaînes de caractères) ou des tableaux, R va répondre en comparant *chaque* entité de l'objet `A` avec *chaque* entité de l'objet `B` et répondre `TRUE` ou `FALSE` pour chaque comparaison effectuée et non pas globalement. Vous vous retrouverez donc avec une *liste* plus ou moins longue[4] de `TRUE` ou `FALSE`. Pour comparer globalement l'identité de

[4] si vos deux objets sont de longueur différente, il se passera une chose bizarre que j'expliquerai en parlant des ratios.

deux objets, il faudra utiliser la fonction `identical()`, qui renvoie `TRUE` si les objet sont *entièrement* identiques et `FALSE` s'ils ne le sont pas.

Soit la nouvelle liste ordonnée de personnages (mais nous gardons Arthur à sa place):

```
medley=c("Arthur", "Mulan", "Darwin", "Steve McQueen")
```

Comparaison détaillée :

```
id==medley
[1]  TRUE  FALSE FALSE FALSE #R a comparé les personnages un par un
>
```

Comparaison globale :

```
identical(id, medley)
[1] FALSE # R a comparé les deux objets globalement.
>
```

6.2 Comment tester si la condition A *n'est pas* remplie ?

Dans le cas ou votre condition A était du type "a>b ?", c'est évident : s'il n'est pas vrai que a>b alors c'est que a<=b (n'oubliez pas la possibilité d'une égalité), et c'est ce test que vous effectuerez.

Par contre, dans le cas où votre condition A était une égalité, il vous faut pour tester le contraire utiliser l'opérateur `!=` qui signifie "différent de" :

```
rien=0
beaucoup=999
rien!=beaucoup
[1] TRUE
>
```

6.3 Comment tester si les conditions A *et* B sont remplies ?

Il faut utiliser l'opérateur `&` ainsi :

<condition A> `&` <condition B>

R ne répondra `TRUE` que si les *deux* conditions sont remplies :

```
rien=0
peu=1
beaucoup=999
rien<peu & peu>beaucoup
[1] FALSE
>
```

Ici, seule la première condition `rien<peu` était remplie, c'est insuffisant.

Dans un test `if`, cela donnera :

```
if(rien<peu & peu<beaucoup){
    cat("peu vaut plus que rien mais moins que beaucoup \n")
```

```
}
peu vaut plus que rien mais moins que beaucoup
>
```

6.4 Comment tester si les conditions A *ou* B sont remplies ?

Il faut utiliser l'opérateur `|` ainsi :

<condition a> `|` <condition b>

R répondra `TRUE` si *au moins une* des deux conditions est remplie :

```
rien=0
peu=1
beaucoup=999
rien<peu | peu>beaucoup
[1] TRUE
>
```

Ici, seule la première condition `rien<peu` était remplie, mais ça suffit.

Dans un test `if`, on aura donc :

```
if(rien<peu | peu<beaucoup){
    cat("Pour l'instant je ne suis par encore certain \n")
    cat("de la position de peu par rapport à rien et beaucoup \n")
}
Pour l'instant je ne suis par encore certain
de la position de peu par rapport à rien et beaucoup
>
```

6.5 Comment tester si *aucune* des deux conditions A et B n'est remplie ?

Il faut tout simplement utiliser `&` pour relier les conditions *contraires* à ce que vous voulez.

Exemple :
condition A : `machin==100` (donc, condition contraire : `machin!=100`
condition B : `truc>0` (donc, condition contraire : `truc<=0`)

Pour vérifier qu'aucune des deux conditions A et B n'est remplie :

```
machin!=100 & truc <=0
```

7. Manipulation des colonnes d'un tableau

Si vous ne l'aviez pas encore fait, c'est *vraiment* le moment de créer le tableau `Kaamelot` comme indiqué au début de ce document. Rappel de son contenu :

```
Kaamelot
         id      role courage
1    Arthur       roi     500
2 Guenièvre     reine      NA
3  Perceval chevalier     100
4    Bohort chevalier       0
```

7.1 Comment *compter les colonnes* d'un tableau ?

Vous vous souvenez peut être de notre emploi de la fonction `length()` pour récupérer la *longueur* d'une colonne de tableau. Appliquée à un *tableau* (qui est en deux dimensions) cette même fonction vous donne le *nombre* de colonnes (*et non pas* le nombre de lignes) du tableau. Notre tableau Kaamelot contient *trois* colonnes de *quatre* lignes, ainsi :

```
length(Kaamelot)
[1] 3
>
```

Affiche bien le nombre de *colonnes* et non celui des lignes.

7.2 Comment *afficher le nom des colonnes* d'un tableau ?

Vous connaissez déjà la fonction `head()`, qui affiche les premières lignes d'un tableau (*dont* les noms des colonnes) ainsi que la fonction `summary()` qui vous donne des infos plus complètes sur le contenu du tableau *dont* le nom des colonnes. Cependant, si vous voulez que votre programme vérifie le nom des colonnes, il faut pouvoir obtenir ces noms de manière *isolée* et pas au milieu d'un fatras d'autres choses. C'est l'objet de la fonction `names()`. Dans le cas d'un tableau (objet de structure `data.frame`), elle renvoie le nom des colonnes qui constituent le tableau.

```
names(Kaamelot)
[1] "id"      "role"    "courage"
>
```

Pour connaître le nom de colonne se situant à un rang particulier dans la liste des noms, utilisez le principe de l'indice [i] avec i le numéro du rang qui vous intéresse, ainsi :

```
names(Kaamelot)[2]
[1] "role"
>
```

Ceci vous servira lorsque votre boucle parcourra la liste des noms de colonnes pour rechercher les noms incorrects.

7.3 Comment tester *globalement* si les noms des colonnes sont identiques à une référence ?

En utilisant une fonction déjà vue : `identical(`<objet 1>`,`<objet 2>`)`. Cette fonction répond `TRUE` si les deux objets sont totalement identiques et `FALSE` dans le cas contraire.

Exemple :

```
ref=c("Id", "Role", "Courage")
identical(names(Kaamelot), ref)
[1] FALSE
>
```

C'est à cause des majuscules (vous l'aviez tous, remarqué, bien sûr ?):

```
identical(names(Kaamelot), c("id", "role", "courage"))
[1] TRUE
>
```

Si vous voulez utiliser la réponse (`TRUE` ou `FALSE`) pour déclencher une action, il faudra utiliser la

fonction `identical()` à l'intérieur d'un test `if/else`, comme ceci :

```
if(identical(names(Kaamelot), ref)==TRUE) { #attention au double égal
      cat("OK, c'est pareil \n")
} else {
      cat("Ah ben non, c'est différent ! \n")
   }
Ah ben non, c'est différent !
>
```

Maintenant que vous comprenez le principe, on peut simplifier car dans un test `if`, lorsqu'on ne précise pas quelle réponse on attend, R comprend implicitement que la réponse attendue est `TRUE`. Aussi, il est équivalent d'écrire :

```
if(A==TRUE)
```

et

```
if(A)
```

En fait ça se comprend bien : c'est comme ça qu'un humain interprèterait « si A ».

Donc on peut réécrire le test plus simplement ainsi :

```
if(identical(names(Kaamelot), ref)) { #sous entendu : si c'est vrai
      cat("OK, c'est pareil \n")
} else {
      cat("Ah ben non, c'est différent ! \n")
   }
Ah ben non, c'est différent !
>
```

7.4 Comment afficher *uniquement* les noms de colonnes *incorrects* ?

Vu le principe de l'exercice (apprendre à faire des boucles et des tests conditionnels, même si R *pourrait* procéder autrement), la logique sera de créer une boucle qui lise les noms des colonnes de votre tableau de données Penvins `mydata` un par un, et d'intégrer à cette boucle un test conditionnel qui affichera le nom de la colonne (avec un message d'erreur) *uniquement* s'il ne correspond pas au nom qui se situe au même rang dans les noms de référence à respecter. Si si, vous *pouvez* le faire.

7.5 Comment tester le *type de données* stockées dans une colonne ?

Vous utiliserez pour cela la commande `class()`. S'agissant des colonnes d'un tableau (objet de structure `data.frame`), elle aura les réponses suivantes :

Contenu de la colonne	`class()` répondra :
des chiffres (et des NA eventuels)	"numeric"
des lettres (et des NA éventuels)	"factor"
des NA (uniquement!)	"logical"

Si on l'applique à `Kaamelot` :

```
class(courage)
[1] "numeric"
>

class(id)
[1] "factor"
>
```

Le fait que des chiffres donnent le type `numeric` n'a rien de surprenant. Par contre, le terme `factor` peut étonner pour des chaines de caractères, mais par défaut c'est comme ça que R considèrera les colonnes de `data.frame` contenant des caractères.

C'est un réglage par défaut désirable parce que 99 fois sur 100, lorsque vous chargez un tableau dans R c'est pour faire des stats, et les différents niveaux de vos facteurs seront toujours désignés avec des lettres (sinon, danger ! ils seraient considérés comme des variables numériques, et dans une ANOVA ça ne fait pas du tout la même chose !).

Si vous créez une simple liste de mots, voilà ce qui se passe :

```
mots=c("caverne", "chevaliers", "lapin", "carnage", "grenade")
class(mots)
[1] "character"
>
```

Donc, le type (chaînes de) caractères existe bien dans R et se nomme `character`.

Mais si vous créez un tableau avec cette liste (ici, un tableau à une seule colonne) et demandez le type de la colonne `mots` :

```
> tableau=data.frame(mots)
class(tableau)
[1] "data.frame"
>


class(tableau$mots) # notation tableau$ car tableau n'est pas attaché
[1] "factor"
>
```

Voilà la colonne `mots` de l'objet `tableau` bel et bien devenue un facteur.

Vous découvrirez qu'on peut transformer d'autre types les uns dans les autres (dans certaines limites) si on en a besoin, mais vous n'en avez pas besoin ici donc je zappe ce sujet.

Quant au fait que les colonnes ne contenant que des `NA` soient classées (faute de mieux) dans la classe `logical` qui normalement est faite pour gérer les colonnes de `TRUE` et `FALSE`, cela vous sera surtout utile ici pour... détecter les colonnes qui ne contiennent que des `NA`.

8. Manipulation des lignes d'un tableau

8.1 Comment *accéder à la i^{ème} ligne* de la colonne A ?

De la même manière que vous accèderiez au i^{ème} élément de n'importe quel objet : en utilisant la notation objet[i] — où i est un nombre entier — qui renvoie le i^{ème} élément de l'objet s'il existe.

```
stuff=c("European swallow", "African swallow", "coconut")
stuff[3]
[1] "coconut"
```

Attention, si vous indiquez un rang trop grand, R ne vous donnera pas de message indiquant que l'objet n'est pas aussi long que ça, il répondra simplement : **NA**

```
stuff[958756]
[1] NA
```

Et là où ça se complique hélas (un peu), c'est que appliquée à un tableau, cette notation objet[i] renvoie non pas des lignes mais des *colonnes* (car un tableau est un objet en poupées russes dont les éléments de premier niveau sont des colonnes. Ainsi :

```
> Kaamelot[3]
  courage
1     500
2      NA
3     100
4       0
>
```

La colonne **courage** est bien la troisième colonne du tableau.

Si vous vouliez seulement la première ligne de cette troisième colonne, il fallait le préciser ainsi :

```
Kaamelot[1, 3]
[1] 500
>
```

Vous aurez noté que c'est le numéro de *ligne* qu'on indique en premier.

Il y a d'autres possibilités pour faire la même chose :

```
Kaamelot$courage[1]
[1] 500
>
```

Et le plus pratique encore, comme on l'a déjà vu, puisque votre fichier a été attaché par la fonction **attach(Kaamelot)**, on peut invoquer directement le nom des colonnes :

```
courage[1]
[1] 500
>
```

Les opérateurs objet[i] ont bien d'autres subtilités dont nous nous passerons ici (avec un tableau, les

curieux pourront explorer en particulier le comportement du double crochet : tableau[[i]] et sa combinaison avec un autre simple crochet : tableau[[i]][j]).

8.2 Comment tester si une ligne particulière contient une chose précise (chiffre, texte, NA)

Ce n'est rien d'autre qu'une expression logique utilisant la notation nécessaire pour désigner la case que vous voulez examiner.

```
role[3]=="chevalier" #Perceval
[1] TRUE
>
```

Pour les NA, on dispose de la fonction `is.na()`, qui renvoie `TRUE` si c'est un NA et `FALSE` sinon.

```
is.na(courage[2]) #le courage de Guenièvre, qui est effectivement NA
[1] TRUE
```

Attention au piège des guillemets :

```
is.na("NA")
[1] FALSE
```

Comment ça "faux ?". C'est parce que `NA` est un mot de type "logical", qui s'écrit sans guillemets.

```
is.na(NA)
[1] TRUE
```

Dans le cadre de cet exercice, pour examiner une par une les cases d'une colonne, vous créerez bien sûr une boucle for, combinée à la notation objet[i]. Dans cette boucle, vous pourrez alors inclure un test conditionnel if/else affichant un message d'erreur pertinent si le contenu de la case est anormal.

8.3 Comment tester si le contenu d'une case est bien situé entre deux valeurs min et max ?

Pour lire le contenu d'une case, voir section 8.1, et pour tester si un chiffre est entre deux valeurs, il suffit de tester simultanément deux conditions : qu'il soit inférieur à min et supérieur à max (donc, avec l'opérateur `&`) voir section 6.2.

8.4 Comment *remplacer automatiquement* une valeur aberrante par NA ?

L'opérateur d'affectation `=` (équivalent à la flèche gauche `<-` mais que je lui préfère par pure paresse parce qu'il représente une touche et non deux) vous permet de remplacer n'importe quelle case de votre tableau par ce que vous voulez, y compris un NA.

Rien ne vous empêche donc de déclencher cette opération sur la base d'un test conditionnel qui aura déterminé que la valeur contenue dans la case était matériellement impossible. Dans ce cas, il serait cependant bon de prévenir l'utilisateur par un message indiquant l'endroit où ce remplacement a été effectué et quelle était la valeur aberrante.

Soyez également conscients que ce remplacement affecte uniquement le tableau qui est temporairement en mémoire dans R, *pas* votre fichier de données originel. Il faudra donc retourner au cahier de manip pour tenter de déceler une simple faute de frappe, et corriger votre fichier de données le cas échéant.

8.5 Comment calculer un *ratio entre deux colonnes numériques* ?

Si j'allais au bout de la logique de l'exercice, je vous demanderais d'utiliser une boucle `for` calculant ce ratio pour chaque ligne et plaçant le résultat ligne par ligne dans un nouvel objet. Ce serait quand même pousser un peu loin, car R est conçu pour manipuler les données *en bloc* justement pour éviter ce genre de boucles, et je pense que vous avez mangé suffisamment de boucles comme ça à ce stade.

Ce ratio sera donc tout simplement obtenu en demandant à R de diviser une colonne par l'autre. R va alors diviser *chaque valeur* de la colonne 1 par la valeur *de même rang* dans la colonne 2.

Pour que cette opération fasse ce que vous voulez ici, il faut évidemment que les objets à diviser l'un par l'autre *aient la même longueur*, (ce qui est garanti dans un tableau). Comme notre tableau `Kaamelot` ne contient qu'une colonne numérique, qu'à cela ne tienne, créons-en une autre :

```
Kaamelot[4]=c(5,1,1,1)
```

Le tableau devient :

```
Kaamelot
         id      role courage V4
1    Arthur       roi     500  5
2 Guenièvre     reine      NA  1
3  Perceval chevalier     100  1
4    Bohort chevalier       0  1
```

Cette colonne 4 a été nommée par défaut `V4` pour "variable 4". Changeons ce nom cryptique :

```
names(Kaamelot)[4]="diviseur"
Kaamelot
         id      role courage diviseur
1    Arthur       roi     500        5
2 Guenièvre     reine      NA        1
3  Perceval chevalier     100        1
4    Bohort chevalier       0        1
>
```

Pour que ce nom de nouvelle colonne soit accessible directement, je détache l'ancienne version de Kaamelot et j'attache la nouvelle :

```
detach(Kaamelot)
attach(Kaamelot)
```

Il ne reste plus qu'à obtenir le ratio

```
courage/diviseur
[1] 100  NA 100   0
>
```

Que se passe t-il si les deux objets ne sont pas de même longueur ?

```
a=c(1, 2, 3)
b=c(4, 5)
a/b
```

```
[1] 0.25 0.40 0.75
Message d'avis :
In a/b :la taille d'un objet plus long n'est pas multiple de la taille d'un
objet plus court
>
```

Les deux premiers chiffres obtenus correspondent bien à 1/4 et 2/5 mais que s'est il passé ensuite ? D'où sort ce `0.75` alors qu'il n'y avait que deux éléments diviseurs dans `b` ?

Il s'est passé que, ayant épuisé les deux diviseurs disponibles pour les deux premières valeurs de `a`, R a commencé à "recycler" l'objet `b`, en recommençant au début. Or le premier diviseur de `b` est 4. Il obtient évidemment 3/4 = `0.75` . Mais comme ça n'est *probablement* pas ce que vous vouliez faire, le message *d'avis* (et non *d'erreur*) vous prévient que vos deux objets n'avaient pas la même longueur.

Mais pourquoi avoir permis de réaliser cette opération au lieu de tout bloquer avec un message d'*erreur* (et non un simple *avis*) ? Parce que dans certains cas c'est très pratique : supposons que vous ayez une colonne `bigdata` contenant 100 000 valeurs que vous voulez diviser *alternativement* par 2 et 10. Au lieu de créer une énorme colonne `denominator` alternant les 2 et les 10 puis faire le ratio, il vous suffit de demander (je décompose l'opération pour plus de clarté):

```
denominator =c(2, 10)
newbigdata=bigdata/denominator
```

Le dénominateur sera recyclé autant de fois que nécessaire, pour traiter les 100 000 données, en alternant les divisions par 2 et par 10, et le résultat sera stocké dans `newbigdata`.

9. Qu'est-ce qu'une fonction et comment en créer une ?

NB. vous pourriez réaliser entièrement le script *CheckPenvins* sans créer une seule fonction, mais cette notion devrait faciliter grandement l'aspect collaboratif du travail en évitant que les variables créées par les uns écrasent celles des autres...

D'abord, une définition.

> *Une fonction est une suite d'instructions que l'on déclenche en appelant son nom.*

Pour *créer* une fonction, il faut choisir son *nom* et utiliser la syntaxe suivante :

```
<nom>=function(<arguments éventuels>){suite d'instructions}
```

Pour *utiliser* une fonction, il faut utiliser cette syntaxe là :

```
<nom de la fonction>(<arguments éventuels>)
```

Voici un exemple de mini-fonction d'une ligne, utilisable "vide" (sans arguments) :

```
salut=function(){cat("Veuillez agréer, Madame, Monsieur, etc. \n")}
```

Les seules choses indispensables ici sont en rouge :
1) la fonction a un nom (salut)
2) elle est déclarée comme une fonction (function)

3) il y a des parenthèses (vides ici car cette fonction n'utilise pas d'arguments)
4) il y a des accolades, contenant ici une seule instruction (en fait, elle n'est même pas indispensable, mais vous ne voudrez probablement pas créer de fonction qui ne fait absolument rien).

Ensuite, eh bien c'est comme d'habitude :

```
salut() #ne pas oublier les parenthèses, même si elles sont vides.
Veuillez agréer, Madame, Monsieur, etc.
```

Si nous voulons raffiner la fonction pour qu'elle utilise une information pertinente fournie par l'utilisateur (ou le programme, qui ira la chercher dans les données), il faut lui faire utiliser des *arguments*. Ici, il y en a un qui paraît évident, et qui permet de réécrire la fonction ainsi :

```
salut=function(x){cat("Veuillez agréer,", x, ", l'expression de etc. \n")}
```

Ma fonction attend maintenant un argument, et ne manquera pas de me le rappeler si je l'oublie :

```
salut() #pas d'argument
Erreur dans cat("Veuillez agréer,", x, ", l'expression de etc. \n") :
  l'argument "x" est manquant, avec aucune valeur par défaut
>
```

Par contre, si je joue le jeu, ça marche :

```
salut("espèce d'abruti") #ne pas oublier les guillemets, c'est du texte
Veuillez agréer, espèce d'abruti, l'expression de etc.
>
```

Comble du raffinement, je peux indiquer une *valeur par défaut* pour tout argument de la fonction pour lequel ça m'intéresse d'en donner un. Ma fonction devient par exemple :

```
salut=function(x="Mesdames, Messieur et autres entités, "){
      cat("Veuillez agréer,", x, "l'expression de etc. \n")
}
```

Ainsi, *si je n'indique rien* pour x, c'est sa valeur par défaut qui sera utilisée :

```
salut()#pas d'argument, donc on tape dans l'argument par défaut
Veuillez agréer, Mesdames, Messieurs et autres entités, l'expression de etc.
>
```

Vous avez déjà utilisé des tas de fonctions dans R, car pratiquement toutes les commandes que vous saisissez sont des fonctions (vous les reconnaissez à leurs parenthèses).

Si jamais vous oubliez les parenthèses d'une fonction, R vous répond habituellement en vous envoyant le *code brut* de la fonction. Exemple avec la fonction `sd()` qui calcule l'écart type :

```
sd #oups, pas de parenthèses
function (x, na.rm = FALSE)
{
    if (is.matrix(x)) {
        msg <- "sd(<matrix>) is deprecated.\n Use apply(*, 2, sd) instead."
        warning(paste(msg, collapse = ""), call. = FALSE, domain = NA)
```

```
        apply(x, 2, sd, na.rm = na.rm)
    }
    else if (is.vector(x))
        sqrt(var(x, na.rm = na.rm))
    else if (is.data.frame(x)) {
        msg <- "sd(<data.frame>) is deprecated.\n Use sapply(*, sd) instead."
        warning(paste(msg, collapse = ""), call. = FALSE, domain = NA)
        sapply(x, sd, na.rm = na.rm)
    }
    else sqrt(var(as.vector(x), na.rm = na.rm))
}
```

Ca fout les jetons hein ?[5]

Une fonction est donc une suite d'instructions empaquetée dans une boite dans laquelle on peut faire rentrer plein de choses et qui peut calculer/afficher tout ce que vous voudrez et qui, lorsqu'elle se ferme, *renvoie* soit une valeur, soit une chaîne de caractère, soit un objet aussi complexe que vous voulez, qui pourra ensuite être utilisé par le reste du programme. Une fonction peut aussi se contenter *d'afficher* quelque chose (par exemple un graphe).

Par contre, les fonctions sont en particulier incapables de *modifier* les objets existants (mais elles peuvent en faire des copies et modifier ces copies), et ne peuvent pas *imposer* de nouveaux objets au reste du programme (c'est le reste du programme qui décide quoi faire avec ce que lui renvoie la fonction lorsqu'elle se ferme).

Il peut sembler complètement idiot de créer de telles usines à gaz si une seule chose peut en sortir et qu'elles n'ont prise sur rien, mais en réalité c'est une excellente idée :

1) une fonction ne peut en elle même rien imposer au reste du programme (changer des valeurs de variables, supprimer des objets, obliger à réaliser telle ou telle action, à quelques exceptions près...)
2) une fonction est beaucoup plus facilement compatible avec d'autres fonctions, que des bouts de programmes non empaquetés dans des fonctions, puisqu'elle n'a *pas accès* à leur contenu et *vice versa*.
3) malgré ces limitations apparentes, une fonction peut parfaitement *afficher* de nombreuses données, *écrire* de nombreuses données dans un fichier, bref il y a quand même moyen de faire des tas de choses (y compris des dégâts) avec une fonction si on sait comment s'y prendre, ce qui explique que les programmes bien écrits ne sont guères que de multiples appels à des fonctions qui se répondent les unes aux autres en utilisant par exemple ce que retourne une fonction comme argument pour la fonction suivante.

Cette incapacité des fonctions à affecter les autres objets du programme autrement que par ce qu'elles renvoient (et qui sera utilisé *ou pas*) est un don du ciel en particulier pour une chose : plusieurs programmeurs peuvent bosser *en parallèle* sur un projet, en choisissant des noms d'objets, de compteurs i, j, k, etc. *librement* à l'intérieur de leurs fonctions, sans devoir se préoccuper du fait que untel aurait pu aussi choisir le mot "cible" ou donner à i la valeur 257 dans sa propre fonction ou le reste du programme.

Bref, utiliser des fonctions pour vos micro-projets ne serait pas une mauvaise idée.

[5] et vous n'avez encore rien vu : essayez donc `read.table` !