Chaînes de caractères

Algo & Prog avec R

A. Malapert, B. Martin, M. Pelleau, et J.-P. Roy

22 octobre 2020

Université Côte d'Azur, CNRS, I3S, France
firstname.lastname@univ-cotedazur.fr

Les chaînes de caractères

Un texte est une suite finie de caractères.
Lettres majuscules ou minuscules, chiffres, ponctuations, symboles mathématiques, etc. En programmation, un texte est une chaîne de caractères (string).

Chaîne de caractères délimitée par des apostrophes

```
> texte <- '1984 : Orwell !'
> texte
[1] "1984 : Orwell !"
```

Attention aux types des données
Ne confondez pas 1984 qui est un entier et '1984' qui est une chaîne.

```
> typeof(1984)
[1] "double"
> typeof('1984')
[1] "character"
```

```
> as.integer('1984')
[1] 1984
> as.character(1984)
[1] "1984"
```

Apostrophes et guillemets

Chaîne de caractères délimitée par des guillemets

```
> texte <- "1984 : Orwell !"
```

Comment inclure une apostrophe dans une chaîne ?

```
> 'Mais j'ai froid'
Erreur : unexpected symbol in "'Mais j'ai"
```

Délimiter avec des guillemets.

```
> "Mais j'ai froid"
[1] "Mais j'ai froid"
```

Protéger avec un backslash (\).

```
> 'Mais j\'ai froid'
[1] "Mais j'ai froid"
```

Une chaîne est une suite de caractères

Combien y a t-il de caractères dans une chaîne ?

```
> texte <- '1984 : Orwell !'
> nchar(texte)
[1] 15
```

Comment accéder au k-ème caractère ?

Pas très pratique, on extrait une sous-chaîne de taille 1 !

```
> substr(texte,1,1)
[1] "1"
> substr(texte,4,4)
[1] "4"
> substr(texte,nchar(texte),nchar(texte))
[1] "! "
> substr(texte,nchar(texte)+1,nchar(texte)+1)
[1] ""
```

En fait, R n'est pas conçu pour ce genre de manipulation.

Création d'une chaîne

La fonction paste combine de multiples chaînes en une seule.

```
  > paste("Sa", "lut")
[1] "Sa lut"
> paste("Sa", "lut",sep="") # ou paste0("Sa", "lut")
[1] "Salut"
```

En fait, la fonction paste fonctionne aussi sur les vecteurs/listes comme presque toutes les fonctions en R.

```
> mots <- c("pomme", "banane", "raisin")
> paste(mots, mots)
[1] "pomme pomme" "banane banane" "raisin raisin"
> paste(mots, mots, collapse="|")
[1] "pomme pomme|banane banane|raisin raisin"
> paste(mots, mots, sep="-",collapse="|")
[1] "pomme-pomme|banane-banane|raisin-raisin"
```

La fonction sprintf formatte une chaîne à la C.

```
> sprintf("%s%s", "Sa", "lut")
[1] "Salut"
```

Extraction d'une tranche de sous-chaîne

R propose une fonction (`substr` ou `substring`) pour extraire une tranche (slice) d'une chaîne repérée par ses positions extrêmes :

```
> texte <- "salut tout le monde"
> substring(texte,first=12)
[1] "le monde"
> substring(texte,first=1,last=11)
[1] "salut tout "
```

Puis-je modifier le k-ème caractère d'une chaîne ?

NON : une chaîne est un objet non mutable.

Mais, on peut quand même . . .

On peut modifier la valeur de la variable avec une affectation, mais cela reste compliquée.

```
> substr(texte,1,5) <- 'Hello'
> texte
[1] "Hello tout le monde"
```

Éclatement d'une chaîne

La fonction `strsplit` sépare une chaîne de caractères en une liste de sous-chaînes en utilisant un séparateur.

```
> word <- "apple|banana|grape"
> v <- strsplit(word,split="|",fixed=TRUE)
> v
[[1]]
[1] "apple" "banana" "grape"
> v[[1]][1]
[1] "apple"
```

- Le paramètre `fixed` indique que le séparateur doit être interprété comme une chaîne et non pas comme une expression régulière.
- Le résultat est renvoyé sous la forme d'une liste, une structure de données essentielle que nous découvrirons plus tard.

Expressions régulières

Expression régulière
une expression régulière (regular expression ou regexp) est un motif qui décrit un ensemble de chaînes de caractères.

- Une regexp est donc une "instruction" donnée à une fonction définissant la recherche (match) et les remplacements (replace) dans une chaîne de caractères.
- R propose une famille de fonctions similaires aux commandes shell `grep` pour la recherche et `sed` pour le remplacement.

```
> l <- c("apple","banana","grape","10","green.pepper")
> grep("a",l)
[1] 1 2 3
> sub("a","$",l)
[1] "$pple" "b$nana" "gr$pe" "10" "green.pepper"
> gsub("a","$",l)
[1] "$pple" "b$n$n$" "gr$pe" "10" "green.pepper"
```

Beaucoup de primitives sur les chaînes . . .

Un langage de programmation vaut aussi par l'étendue de ses fonctionnalités. Avant de programmer, cherchez dans la doc si la fonction convoitée n'est pas une primitive !

. . . mais certaines ont une drôle de tête !
Quasiment toutes les fonctions sur les chaînes prennent en argument et renvoient des VECTEURS ou LISTES. Pour l'instant, nous garderons une idée naïve de ce qu'est un vecteur ou une liste, une séquence d'éléments.

Autres méthodes sur des chaînes

```
> l <- c('', 'pommes', '12', 'CERISES')
> grepl("^[[:digit:]]+$", l) # que des chiffres ?
[1] FALSE FALSE TRUE FALSE
> grepl("^[[:lower:]]+$", l) # que des minuscules ?
[1] FALSE TRUE FALSE FALSE
> grepl("^[[:alpha:]]+$", l) # que des lettres?
[1] FALSE TRUE FALSE TRUE
> grepl("^[[:alnum:]]+$", l) # que des chiffres ou des
    lettres?
[1] FALSE TRUE TRUE TRUE
> tolower(l) # mettre en minuscules
[1] "" "pommes" "12" "cerises"
> toupper(l) # mettre en majuscules
[1] "" "POMMES" "12" "CERISES"
> chartr("oei", "OEI", l) #remplacement (c.f. tr en shell)
[1] "" "pOmmEs" "12" "CERISES"
```

Le code ASCII

Les caractères américains (dits internationaux) sont numérotés de 0 à 127, c'est le code ASCII (American Standard Code for Information Interchange). Par exemple le code ASCII de 'A' est égal à 65 mais on n'a pas besoin de le mémoriser :

```
> utf8ToInt('A')
[1] 65
> utf8ToInt('a')
[1] 97
```

```
> utf8ToInt('?')
[1] 63
> utf8ToInt('!')
[1] 33
```

```
> intToUtf8(65)
[1] "A"
> intToUtf8(63)
[1] "?"
```

Les caractères dont le code ASCII tombe entre 0 et 31, ainsi que le caractère numéro 127 ne sont pas affichables . Ce sont les caractères de contrôle nommés symboliquement NUL , LF , BEL , CR , ESC , DEL ...

```
> intToUtf8(27) # ESC
[1] "\033"
```

```
> intToUtf8(10) # LF
[1] "\n"
```

Table ASCII

```
> Ascii()
 [1]  32:    33:!   34:"   35:#   36:$   37:%   38:&   39:’   40:(   41:)
[11]  42:*   43:+   44:,   45:-   46:.   47:/   48:0   49:1   50:2   51:3
[21]  52:4   53:5   54:6   55:7   56:8   57:9   58::   59:;   60:<   61:=
[31]  62:>   63:?   64:@   65:A   66:B   67:C   68:D   69:E   70:F   71:G
[41]  72:H   73:I   74:J   75:K   76:L   77:M   78:N   79:O   80:P   81:Q
[51]  82:R   83:S   84:T   85:U   86:V   87:W   88:X   89:Y   90:Z   91:[
[61]  92:\\  93:]   94:^   95:_   96:‘   97:a   98:b   99:c  100:d  101:e
[71] 102:f  103:g  104:h  105:i  106:j  107:k  108:l  109:m  110:n  111:o
[81] 112:p  113:q  114:r  115:s  116:t  117:u  118:v  119:w  120:x  121:y
[91] 122:z  123:{  124:|  125:}  126:~
```

À faire en TP

```r
Ascii <- function() {
  # ...
}
```

Le code ASCII étendu

Le code du caractère accentué 'é' est 233, il tombe entre 128 et 255. Il s'agit du code ASCII étendu, qui contient des caractères propres à chaque pays. Nous utilisons le codage Iso-Latin-1 , qui couvre les langues de l'Europe de l'Ouest, avec le é, à, ñ, ö, etc. Les russes utilisent KOI8-R, les chinois sont bien ennuyés, etc.

Pourquoi ce nombre magique 255 ?
Parce que l'unité d'information sur un ordinateur est l'octet : un bloc de 8 bits (un bit = 0 ou 1). Sur les 8 bits, un bit était utilisé pour la vérification de la bonne transmission (le bit de parité), d'où un codage sur 7 bits, c'est l'ASCII. Avec la fiabilité des transmissions, le 8ème bit fut finalement utilisé pour l'ASCII étendu ...

1	1	1	0	1	0	0	1	é

Le code UNICODE

Et les chinois furent contents ! En effet, le consortium Unicode (env 1990, initié par Xerox et Apple) proposait d'abandonner la limitation des 255 caractères en traitant d'un seul coup toutes les langues du monde (plus de 65000 caractères !) ...

- Unicode donne à tout caractère de n'importe quel système d'écriture un nom et un identifiant numérique, et ce de manière unifiée, quelle que soit la plate-forme informatique ou le logiciel.
- Le jeu de caractères est la liste des caractères, leur nom et leur index, le point de code.
- Unicode sépare la définition du jeu de caractères de celle du codage.

UTF-8

- UTF-8 est un format de transformation, il définit un codage pour tout caractère Unicode.
- UTF-8 est compatible avec la norme ASCII.

Usage du code UNICODE

Les caractères Unicode sont numérotés en hexadécimal (base 16). Par exemple, le symbole euro a pour numéro '20AC' en hexa :

```
> '\u20ac' # symbole euro
> '\u6211\u662F\u6cd5\u56fd\u4eba' # je suis français
> for(i in 1633:1642) print(intToUtf8(i)) # ?
```

Conversion en hexadécimal

```
> x <- as.hexmode(8364)
> x
[1] "20ac"
> x <- as.integer(x)
> x
[1] 8364
```

Jules César et les messages secrets

La cryptographie est la science des codes secrets. Une des premières formes de codage d'un message provient de Jules César durant la Guerre des Gaules. Ici, on ne code que les MAJUSCULES...

Pour coder un message

- Choisir une clé de codage `k` dans [1,25].
- Initialiser la chaîne résultat à la chaîne vide.
- Pour chaque caractère c du message à coder :
  - Si le caractère c est une lettre majuscule : coder le caractère c en le décalant de k positions à droite insérer le nouveau caractère dans la chaîne résultat.
  - sinon : insérer c dans la chaîne résultat
- Rendre en résultat la chaîne résultat

Code César avec une clé `k=3`
'TOUS au ZOO !' $\Longrightarrow$ 'WRXV au CRR !'

Questions?

Retrouvez ce cours sur le site web

`www.i3s.unice.fr/~malapert/R`

Backup slides

Les diapositives suivantes, issues du cours 7, doivent probablement être déplacées dans un autre cours.
En effet, on utilise les vecteurs à bloc !

Codage d'un entier en binaire I

Coder en binaire un entier consiste à remarquer que ses chiffres (lus de droite à gauche) ne sont autres que les restes des divisions par 2.

$(13)_{10}$ **s'écrit** $(1101)_2$ **en binaire :** $13 = 6 \times 2 + 1$

- Le prefixe 110 correspond au codage du quotient (6).
- Le suffixe 1 correspond au reste (1).

Par récurrence

```
Codebin <- function(n) {
  if(n < 2) {
    return(as.character(n))
  } else {
    return(paste(codebin(n%/%2),n %% 2,sep=''))
  }}
```

```
> codebin(13)
[1] '1101'
> codebin(0)
[1] '0'
```

Codage d'un entier en binaire II

Par itération

```
Codebin <- function(n) {
  res <- c()
  while( n > 0) {
    res <- append(n %% 2, res)
    n <- n %/%2
  }
  return(paste(res,collapse=''))
}
```

Questions

- Que renvoie l'appel `Codebin(13)` ?
- Que renvoie l'appel `Codebin(0)` ?
- Comment peut-on corriger la fonction `Codebin` ?

Décodage du binaire vers le décimal : méthode naïve

Le mauvais algorithme consiste à calculer des puissances de 2.

$$(1101)_2 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$$

```
Decodebin <- function(n) {
  digits <- utf8ToInt(n)- utf8ToInt('0')
  res <- 0
  for(i in seq_along(digits)) {
    res <- res + digits[i]* 2**(length(digits)-i)
  }
  return(res)
}
```

```
> Decodebin('1101')
[1] 13
```

Décodage du binaire vers le décimal : schéma de Horner

Le bon algorithme consiste à implémente un schéma de Hörner : calcul optimal de la valeur d'un polynôme en un point sans calculs de puissances.

$$(1101)_2 = (1 \times 2 + 1) \times 2 + 0) \times 2 + 1$$

```
Decodebin <- function(n) {
  horner <- function(digits) {
    res <- 0
    if(length(digits) > 0) {
      res <- 2* horner(head(digits,-1)) + tail(digits,1)
    }
    return(res)
  }
  digits <- utf8ToInt(n)- utf8ToInt('0')
  return(horner(digits))
}
```

```
> Decodebin('1101')
[1] 13
```

Traitement des vecteurs

Une chaîne de caractères n'est pas mutable, mais on peut la transformer en un vecteur de code de caractères, et un vecteur est mutable ...

Transformation des chiffres en '?'

```
Cacher <- function(str) {
  chars <- utf8ToInt(str)
  c0 <- utf8ToInt('0')
  c9 <- utf8ToInt('9')
  res <- c()
  for(i in seq_along(chars)) {
    if(chars[i]>=c0 && chars[i]<=c9) {
      res <- append(res,utf8ToInt('?'))
    } else {
      res <- append(res, chars[i])
    }
  }
  return(intToUtf8(res))
}
```

```
> Cacher('324DAC75')
[1] '???DAC??'
```

Inconvénient : construction incrémentale de `res` par `append` est coûteuse.

Traitement vectorisé des vecteurs

La solution consiste à basculer la chaîne dans un vecteur en un vecteur de code de caractères, puis à effectuer un traitement de vecteur mutable, puis à rebasculer le vecteur liste en une chaîne. Intellectuel non ?

```
Cacher <- function(str) {
  chars <- utf8ToInt(str)
  c0 <- utf8ToInt('0')
  c9 <- utf8ToInt('9')
  numbers <- which( chars >= c0 & chars <= c9)
  chars[numbers] <- utf8ToInt('?')
  return(intToUtf8(chars))
}
```

concaténer un vecteur de chaînes : une opération fréquente

```
> paste(c('a','bb','c'),collapse='')
[1] "abbc"
```