Performance

Techniques avancées en programmation statistique R

Patrick Fournier

Automne 2019

Université du Québec à Montréal

Introduction

Idées reçues

$\rightsquigarrow$ “Il ne faut pas faire de boucles en R car elles sont lentes.”

Idées reçues

- ⇝ “Il ne faut pas faire de boucles en R car elles sont lentes.”
- ⇝ “R est un mauvais language de programmation car il n’est pas performant.”

Idées reçues

- ⇝ “Il ne faut pas faire de boucles en R car elles sont lentes.”
- ⇝ “R est un mauvais language de programmation car il n’est pas performant.”
- ⇝ “Python, Julia, C++, SAS, ... est meilleur que R car il est plus performant.”

Idées reçues

- ⇝ “Il ne faut pas faire de boucles en R car elles sont lentes.”
- ⇝ “R est un mauvais language de programmation car il n’est pas performant.”
- ⇝ “Python, Julia, C++, SAS, ... est meilleur que R car il est plus performant.”
- ⇝ “Il faut implémenter les parties critiques du programme en C/C++/Fortran et le reste en R.”

Idées reçues

- ⇝ “Il ne faut pas faire de boucles en R car elles sont lentes.”
- ⇝ “R est un mauvais language de programmation car il n’est pas performant.”
- ⇝ “Python, Julia, C++, SAS, ... est meilleur que R car il est plus performant.”
- ⇝ “Il faut implémenter les parties critiques du programme en C/C++/Fortran et le reste en R.”
- ⇝ ...

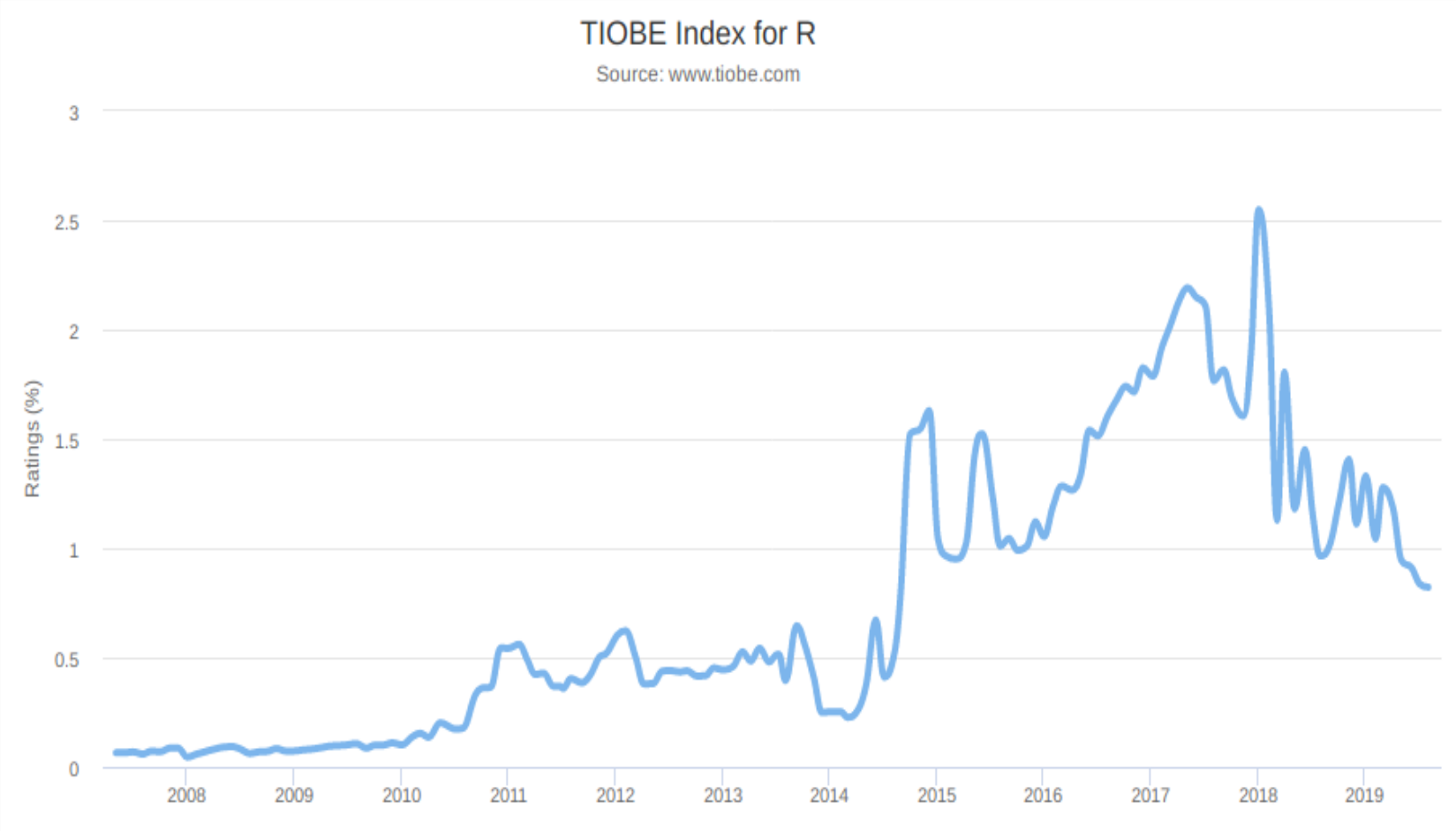
R vs. autres languages [2]

Aug 2019	Aug 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.028%	-0.85%
2	2		C	15.154%	+0.19%
3	4	↑	Python	10.020%	+3.03%
4	3	↓	C++	6.057%	-1.41%
5	6	↑	C#	3.842%	+0.30%
6	5	↓	Visual Basic .NET	3.695%	-1.07%
7	8	↑	JavaScript	2.258%	-0.15%
8	7	↓	PHP	2.075%	-0.85%
9	14	⇈	Objective-C	1.690%	+0.33%
10	9	↓	SQL	1.625%	-0.69%
11	15	⇈	Ruby	1.316%	+0.13%
12	13	↑	MATLAB	1.274%	-0.09%
13	44	⇈	Groovy	1.225%	+1.04%
14	12	↓	Delphi/Object Pascal	1.194%	-0.18%
15	10	⇊	Assembly language	1.114%	-0.30%
16	19	↑	Visual Basic	1.025%	+0.10%
17	17		Go	0.973%	-0.02%
18	11	⇊	Swift	0.890%	-0.49%
19	16	↓	Perl	0.860%	-0.31%
20	18	↓	R	0.822%	-0.14%

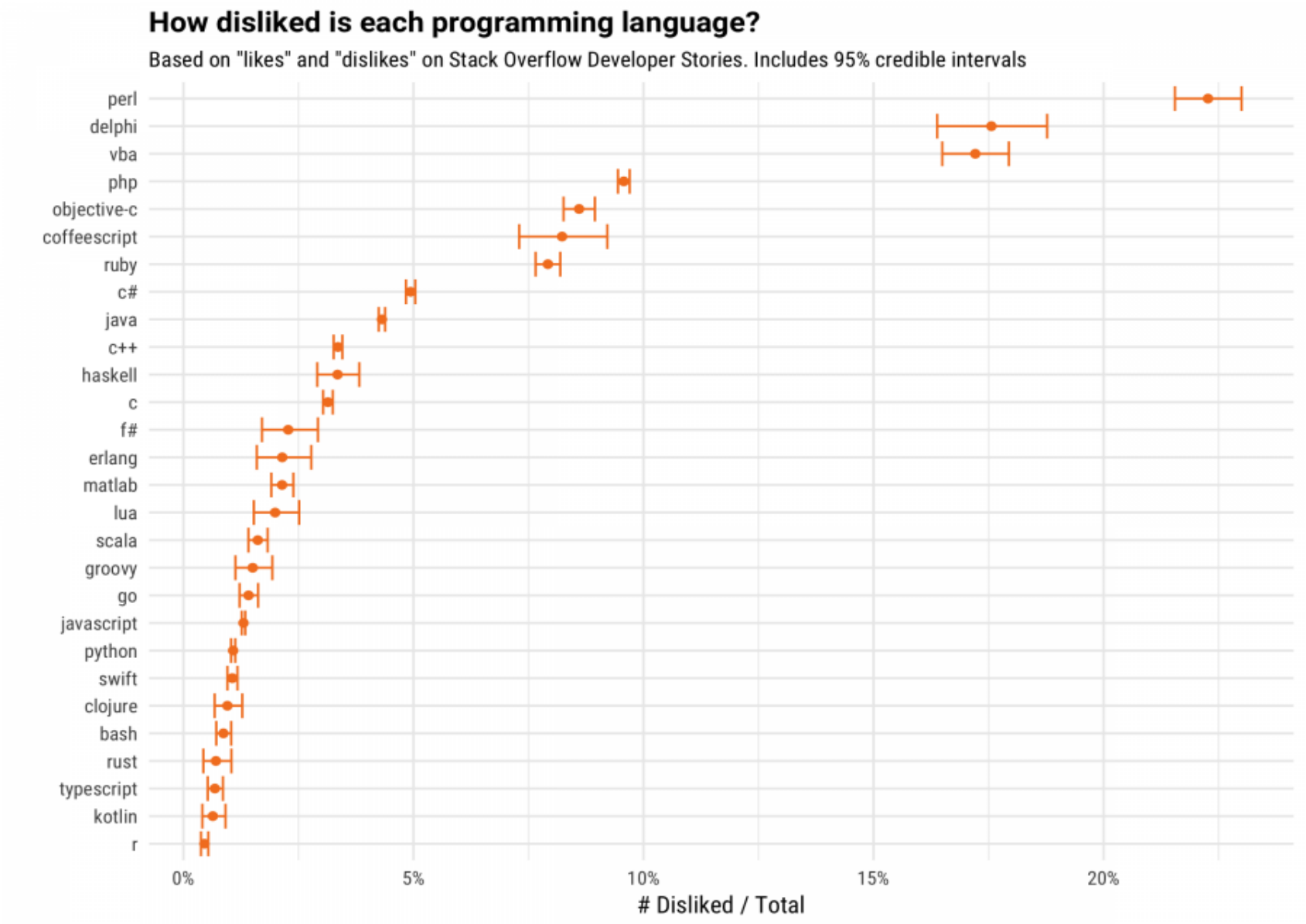
R à travers le temps [2]

Highest Position (since 2001): #8 in Jan 2018

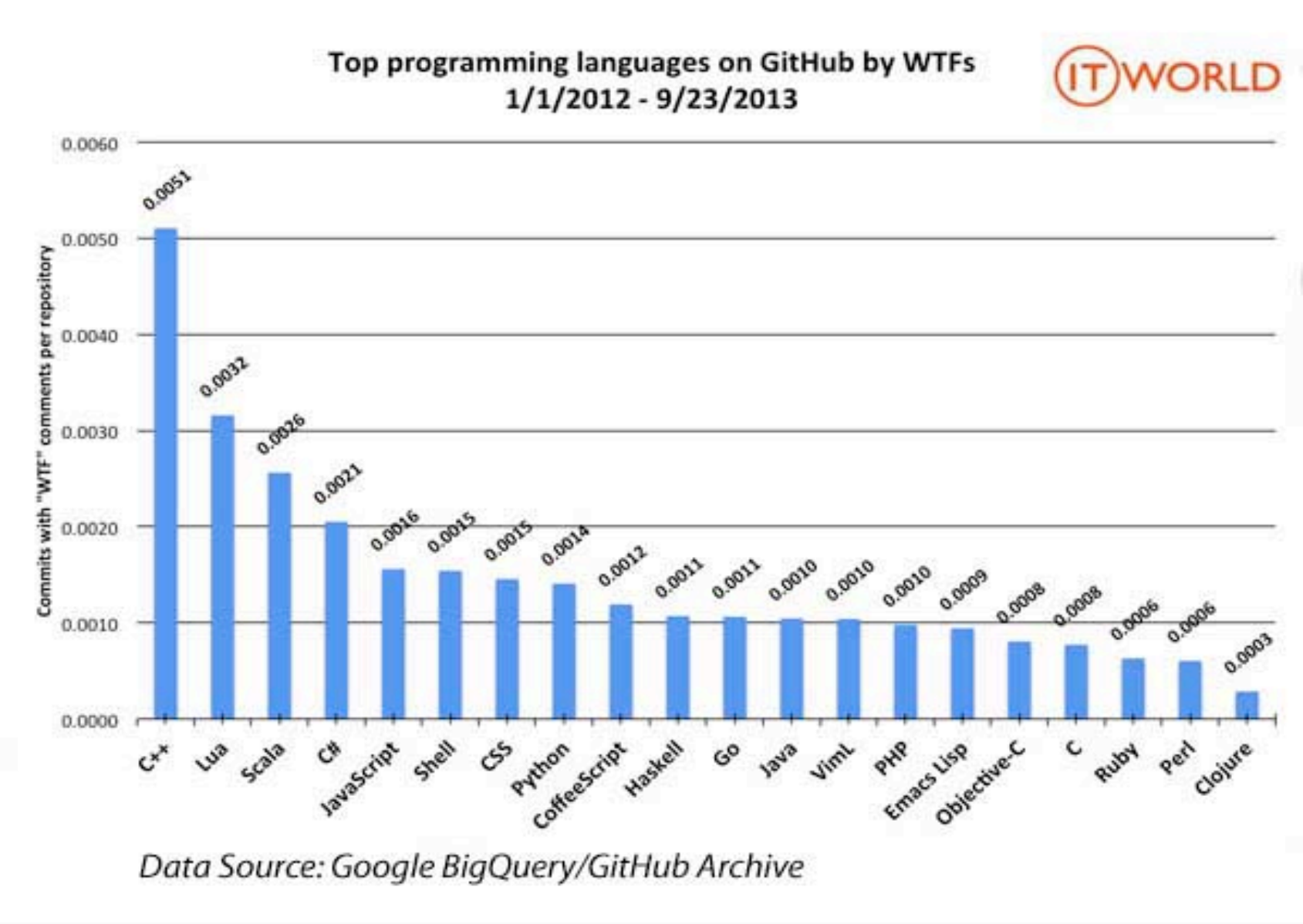
Lowest Position (since 2001): #73 in Dec 2008



Langages détestés [3]



Langage par proportion de WTFs [1]



Le terrible secret des variables en R

Qu’est-ce qu’une variable ?

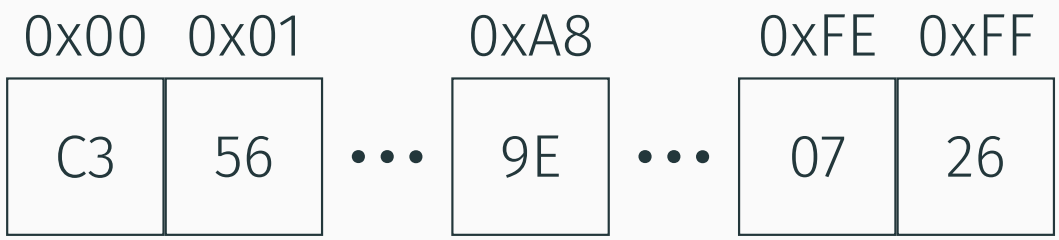
Variable

Symbole représentant un espace mémoire.

Qu’est-ce qu’une variable ?

Variable

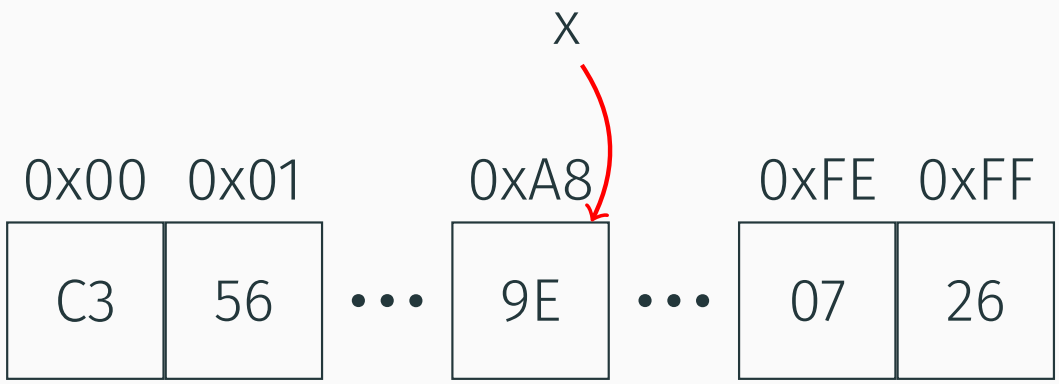
Symbole représentant un espace mémoire.



Qu’est-ce qu’une variable ?

Variable

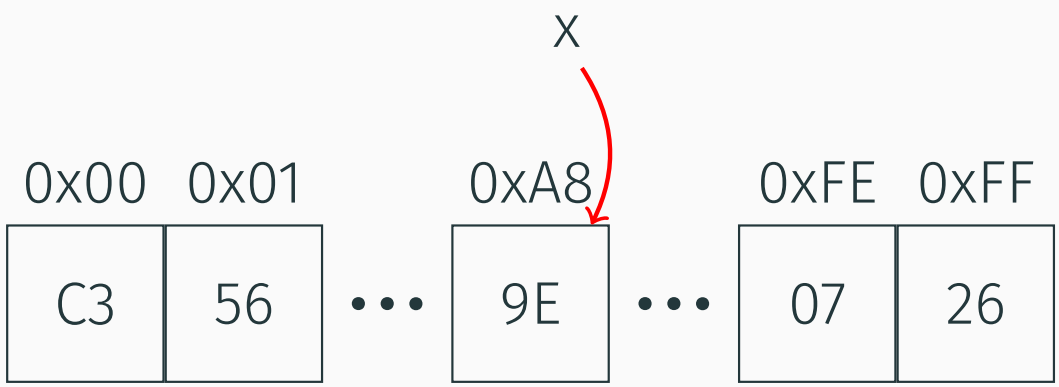
Symbole représentant un espace mémoire.



Qu’est-ce qu’une variable ?

Variable

Symbole représentant un espace mémoire.



Remarque

Selon cette définition, une variable n’a pas besoin de pouvoir varier !

Une petite expérience

```
> x <- 158
> pryr::address(x)
[1] "0x5582f3f626f8"
> y <- x
> pryr::address(y)
[1] "0x5582f3f626f8"
> identical(pryr::address(x), pryr::address(y))
[1] TRUE
> x <- 86
> pryr::address(x)
[1] "0x5582f3f62260"
> pryr::address(y)
[1] "0x5582f3f626f8"
> identical(pryr::address(x), pryr::address(y))
[1] FALSE
```

Une petite expérience

```
> x <- 158
> pryr::address(x)
[1] "0x5582f3f626f8"
> y <- x
> pryr::address(y)
[1] "0x5582f3f626f8"
> identical(pryr::address(x), pryr::address(y))
[1] TRUE
> x <- 86
> pryr::address(x)
[1] "0x5582f3f62260"
> pryr::address(y)
[1] "0x5582f3f626f8"
> identical(pryr::address(x), pryr::address(y))
[1] FALSE
```

???

Types de variables

Mutable

Il est possible de changer la valeur contenue dans la case mémoire associée à une variable.

Types de variables

Mutable

Il est possible de changer la valeur contenue dans la case mémoire associée à une variable.

$\rightsquigarrow$ Approche associée à la programmation *impérative*.

Types de variables

Mutable

Il est possible de changer la valeur contenue dans la case mémoire associée à une variable.

$\rightsquigarrow$ Approche associée à la programmation *impérative*.

Immutable

Une fois qu'une valeur est stockée en mémoire, il n'est pas possible de modifier cette valeur.

Types de variables

Mutable

Il est possible de changer la valeur contenue dans la case mémoire associée à une variable.

⇝ Approche associée à la programmation *impérative*.

Immutable

Une fois qu'une valeur est stockée en mémoire, il n'est pas possible de modifier cette valeur.

⇝ Pour donner l'illusion de la modification, on associe simplement la variable à une autre case mémoire.

Types de variables

Mutable

Il est possible de changer la valeur contenue dans la case mémoire associée à une variable.

- ⇝ Approche associée à la programmation *impérative*.

Immutable

Une fois qu'une valeur est stockée en mémoire, il n'est pas possible de modifier cette valeur.

- ⇝ Pour donner l'illusion de la modification, on associe simplement la variable à une autre case mémoire.
- ⇝ Approche associée à la programmation *fonctionnelle*.

Types de variables

Mutable

Il est possible de changer la valeur contenue dans la case mémoire associée à une variable.

- ⇝ Approche associée à la programmation *impérative*.

Immutable

Une fois qu'une valeur est stockée en mémoire, il n'est pas possible de modifier cette valeur.

- ⇝ Pour donner l'illusion de la modification, on associe simplement la variable à une autre case mémoire.
- ⇝ Approche associée à la programmation *fonctionnelle*.

Et R ?

C'est compliqué.

Compteur de références

$\rightsquigarrow$ R associe à (presque) chaque objet un compteur.

Compteur de références

- ⇝ R associe à (presque) chaque objet un compteur.
  - ⇝ Exception : les objets de type environment, qui sont *toujours mutables*.

Compteur de références

- ⇝ R associe à (presque) chaque objet un compteur.
  - ⇝ Exception : les objets de type environment, qui sont *toujours mutables*.
- ⇝ Déclaration de la variable : compteur initialisé à 1.

Compteur de références

- ⇝ R associe à (presque) chaque objet un compteur.
  - ⇝ Exception : les objets de type environment, qui sont *toujours mutables*.
- ⇝ Déclaration de la variable : compteur initialisé à 1.
- ⇝ Deux manières d'incrémenter le compteur :

Compteur de références

- ⇝ R associe à (presque) chaque objet un compteur.
  - ⇝ Exception : les objets de type environment, qui sont *toujours mutables*.
- ⇝ Déclaration de la variable : compteur initialisé à 1.
- ⇝ Deux manières d'incrémenter le compteur :
  - ⇝ ajouter une référence vers la variable et

Compteur de références

- ⇝ R associe à (presque) chaque objet un compteur.
  - ⇝ Exception : les objets de type environment, qui sont *toujours mutables*.
- ⇝ Déclaration de la variable : compteur initialisé à 1.
- ⇝ Deux manières d'incrémenter le compteur :
  - ⇝ ajouter une référence vers la variable et
  - ⇝ utiliser la variable comme argument pour une fonction non-primitive (et certaines fonctions primitives).

Compteur de références

- ⇝ Lorsque vient le temps de modifier une variable, deux possibilités :

Compteur de références

- ⇝ Lorsque vient le temps de modifier une variable, deux possibilités :
  - ⇝ Si le compteur vaut 1, la variable est mutée.

Compteur de références

- ⇝ Lorsque vient le temps de modifier une variable, deux possibilités :
  - ⇝ Si le compteur vaut 1, la variable est mutée.
  - ⇝ Sinon, une nouvelle variable est crée.

Compteur de références

- ⇝ Lorsque vient le temps de modifier une variable, deux possibilités :
  - ⇝ Si le compteur vaut 1, la variable est mutée.
  - ⇝ Sinon, une nouvelle variable est crée.
- ⇝ Tout cela devient vite compliqué !

Compteur de références

- ⇝ Lorsque vient le temps de modifier une variable, deux possibilités :
  - ⇝ Si le compteur vaut 1, la variable est mutée.
  - ⇝ Sinon, une nouvelle variable est crée.
- ⇝ Tout cela devient vite compliqué !
- ⇝ De manière générale, un code qui suppose que les variables sont immutables sera plus performant.

Compteur de références

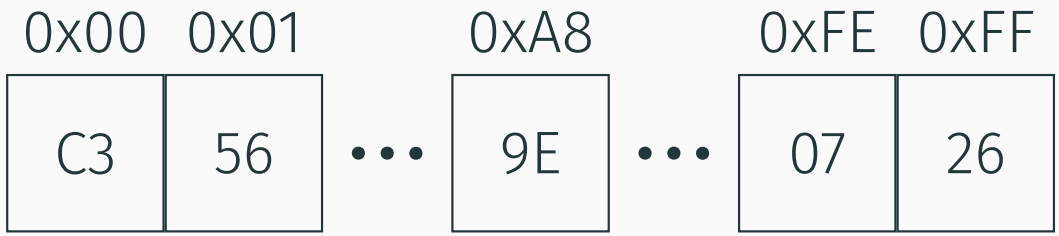
- ⇝ Lorsque vient le temps de modifier une variable, deux possibilités :
  - ⇝ Si le compteur vaut 1, la variable est mutée.
  - ⇝ Sinon, une nouvelle variable est crée.
- ⇝ Tout cela devient vite compliqué !
- ⇝ De manière générale, un code qui suppose que les variables sont immutables sera plus performant.
- ⇝ De plus, bien qu'il soit possible d'exploiter la mutabilité pour obtenir des gains de performances, ils sont habituellement marginaux.

Compteur de références

- ⇝ Lorsque vient le temps de modifier une variable, deux possibilités :
  - ⇝ Si le compteur vaut 1, la variable est mutée.
  - ⇝ Sinon, une nouvelle variable est crée.
- ⇝ Tout cela devient vite compliqué !
- ⇝ De manière générale, un code qui suppose que les variables sont immutables sera plus performant.
- ⇝ De plus, bien qu'il soit possible d'exploiter la mutabilité pour obtenir des gains de performances, ils sont habituellement marginaux.
- ⇝ Cela étant dit, un cas important où la mutabilité peut être exploitée est la *préallocation de mémoire*

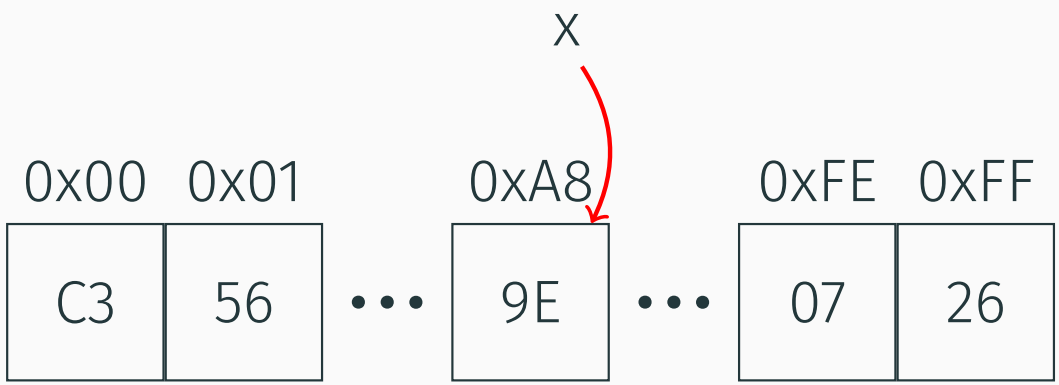
Retour sur l'expérience

```
1   > x <- 158
2   > y <- x
3   > x <- 86
```



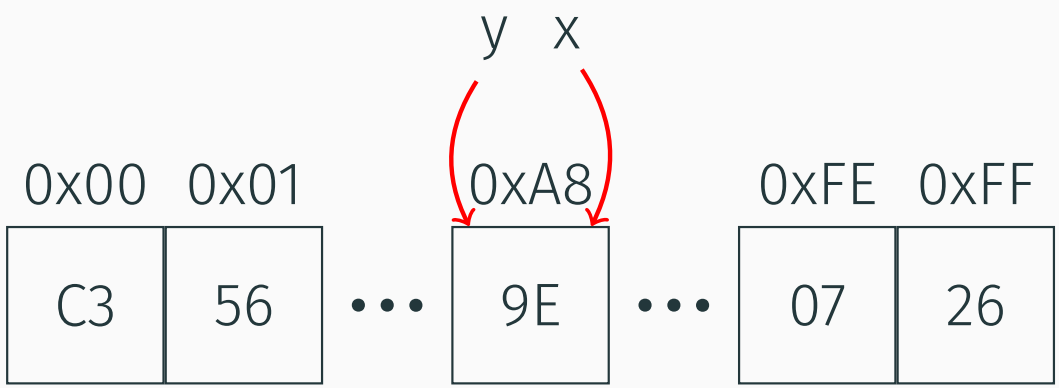
Retour sur l'expérience

```
1   > x <- 158
2   > y <- x
3   > x <- 86
```



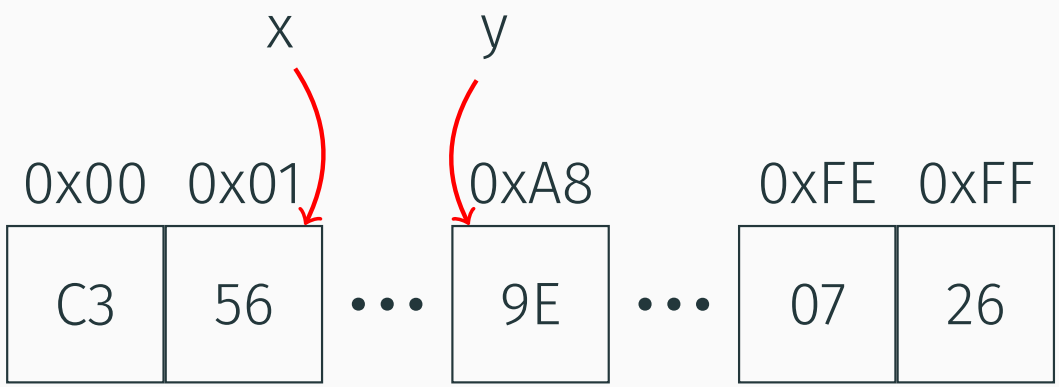
Retour sur l'expérience

```
1   > x <- 158
2   > y <- x
3   > x <- 86
```



Retour sur l'expérience

```
1   > x <- 158
2   > y <- x
3   > x <- 86
```



Exemple : préallocation

```
1  lcg <- function(a, b, m,
2                  seed = 0)
3    function(){
4      ret <- (a * seed + b) %% m
5      seed <<- ret
6      ret / m
7    }
```

Exemple

```r
## LCG de numerical recipes.
nr_lcg <- lcg(1664525, 1013904223, 2^32)

rand_prealloc <- function(n){
    res <- numeric(n)
    for (kk in 1:n)
        res[kk] <- nr_lcg()

    res
}

rand_vec <- function(n){
    replicate(n, nr_lcg())
}
```

Exemple

```
> microbenchmark(rand_vec(1e4), rand_prealloc(1e4))
Unit: milliseconds
                  expr   min    lq  mean median    uq  max neval
       rand_vec(10000) 12.31 13.45 14.72  14.65 15.64 27.5   100
 rand_prealloc(10000)  8.48  8.73  9.91   9.29  9.92 46.0   100
```

Que s'est-il passé ?

Exemple

```
> microbenchmark(rand_vec(1e4), rand_prealloc(1e4))
Unit: milliseconds
                  expr   min    lq  mean median    uq  max neval
       rand_vec(10000) 12.31 13.45 14.72  14.65 15.64 27.5   100
 rand_prealloc(10000)  8.48  8.73  9.91   9.29  9.92 46.0   100
```

Que s'est-il passé ?

for

Assigne directement une valeur à un emplacement mémoire bien déterminé.

Exemple

```
> microbenchmark(rand_vec(1e4), rand_prealloc(1e4))
Unit: milliseconds
                  expr   min    lq  mean median    uq  max neval
       rand_vec(10000) 12.31 13.45 14.72  14.65 15.64 27.5   100
 rand_prealloc(10000)  8.48  8.73  9.91   9.29  9.92 46.0   100
```

Que s'est-il passé ?

`for`

Assigne directement une valeur à un emplacement mémoire bien déterminé.

`replicate`

-> `sapply` -> `lapply` -> code compilé

Exemple

```
> microbenchmark(rand_vec(1e4), rand_prealloc(1e4))
Unit: milliseconds
                  expr   min    lq  mean median    uq  max neval
       rand_vec(10000) 12.31 13.45 14.72  14.65 15.64 27.5   100
 rand_prealloc(10000)  8.48  8.73  9.91   9.29  9.92 46.0   100
```

Que s'est-il passé ?

`for`

Assigne directement une valeur à un emplacement mémoire bien déterminé.

`replicate`

-> `sapply` -> `lapply` -> code compilé

Le code à exécuté est tellement simple (`r()`) que l'overhead associé à l'appel de code externe tue la performance !

Fonctions

Fonctions

Fonction

Ensemble d'instructions permettant l'exécution d'une tâche.

Fonctions

Fonction

Ensemble d'instructions permettant l'exécution d'une tâche.

$\rightsquigarrow$ En R, fonction = objet de première classe (comme un vecteur, une liste...)

Fonctions

Fonction

Ensemble d'instructions permettant l'exécution d'une tâche.

$\rightsquigarrow$ En R, fonction = objet de première classe (comme un vecteur, une liste...)

$\rightsquigarrow$ On peut assigner un symbole à une fonction...

Fonctions

Fonction

Ensemble d'instructions permettant l'exécution d'une tâche.

- $\rightsquigarrow$ En R, fonction = objet de première classe (comme un vecteur, une liste...)
- $\rightsquigarrow$ On peut assigner un symbole à une fonction...
- $\rightsquigarrow$ Ou pas ! On parle alors de *fonction lambda* ou de *fonction anonyme*.

Fonctions

Fonction

Ensemble d'instructions permettant l'exécution d'une tâche.

- ⇝ En R, fonction = objet de première classe (comme un vecteur, une liste...)
- ⇝ On peut assigner un symbole à une fonction...
- ⇝ Ou pas ! On parle alors de *fonction lambda* ou de *fonction anonyme*.
  - ⇝ Souvent utilisées dans les fonctionnelle : `apply`, `Reduce`, `Filter`...

Fonctions

Fonction

Ensemble d'instructions permettant l'exécution d'une tâche.

⇝ En R, fonction = objet de première classe (comme un vecteur, une liste...)

⇝ On peut assigner un symbole à une fonction...

⇝ Ou pas ! On parle alors de *fonction lambda* ou de *fonction anonyme*.

⇝ Souvent utilisées dans les fonctionnelle : `apply`, `Reduce`, `Filter`...

```
> Filter(function(x) identical(x %% 2, 0), 1:10)
```

Fonction en R [4]

⇝ En R, une fonction est composée de 3 éléments :

Fonction en R [4]

⇝ En R, une fonction est composée de 3 éléments :

formals

une liste d'arguments,

Fonction en R [4]

⇝ En R, une fonction est composée de 3 éléments :

formals

une liste d'arguments,

body

du code et

Fonction en R [4]

⇝ En R, une fonction est composée de 3 éléments :

formals
: une liste d'arguments,

body
: du code et

environment
: un ensemble de symboles associés à des valeurs.

Fonction en R [4]

- ⇝ En R, une fonction est composée de 3 éléments :

  formals
  : une liste d'arguments,

  body
  : du code et

  environment
  : un ensemble de symboles associés à des valeurs.

- ⇝ Ces trois composantes peuvent être examinées à l'aide des fonctions `formals`, `body` et `environment` respectivement.

Fonction en R [4]

```
> formals(lcg)
$a

$b

$m

$seed
[1] 0
> body(lcg)
{
    function() {
        ret <- (a * seed + b)%%m
        seed <<- ret
        ret/m
    }
}
> environment(lcg)
<environment: R_GlobalEnv>
```

Environment

- ⇝ À quelques détails près, un environment peu être vu comme une simple liste.

Environment

- ⇝ À quelques détails près, un environment peu être vu comme une simple liste.
- ⇝ En particulier, presque tout environment possède un environment parent.

Environment

- ⇝ À quelques détails près, un environment peu être vu comme une simple liste.
- ⇝ En particulier, presque tout environment possède un environment parent.
  - ⇝ Le seul orphelin est `R_EmptyEnv`, *l'ancêtre de tous les environments*.

Environment

- ⇝ À quelques détails près, un environment peu être vu comme une simple liste.
- ⇝ En particulier, presque tout environment possède un environment parent.
  - ⇝ Le seul orphelin est `R_EmptyEnv`, *l'ancêtre de tous les environments*.
  - ⇝ Un autre environment important est `R_GlobalEnv` où ont lieu toutes les opérations situées à l'extérieur d'une fonction.

Environment

- ⇝ À quelques détails près, un environment peu être vu comme une simple liste.
- ⇝ En particulier, presque tout environment possède un environment parent.
  - ⇝ Le seul orphelin est `R_EmptyEnv`, *l'ancêtre de tous les environments*.
  - ⇝ Un autre environment important est `R_GlobalEnv` où ont lieu toutes les opérations situées à l'extérieur d'une fonction.
- ⇝ Lorsqu'un symbole est appelé,

Environment

- ⇝ À quelques détails près, un environment peu être vu comme une simple liste.
- ⇝ En particulier, presque tout environment possède un environment parent.
  - ⇝ Le seul orphelin est `R_EmptyEnv`, *l'ancêtre de tous les environments*.
  - ⇝ Un autre environment important est `R_GlobalEnv` où ont lieu toutes les opérations situées à l'extérieur d'une fonction.
- ⇝ Lorsqu'un symbole est appelé,
  - ⇝ R le cherche dans l'environment courrant.

Environment

- ⇝ À quelques détails près, un environment peu être vu comme une simple liste.
- ⇝ En particulier, presque tout environment possède un environment parent.
  - ⇝ Le seul orphelin est `R_EmptyEnv`, *l'ancêtre de tous les environments*.
  - ⇝ Un autre environment important est `R_GlobalEnv` où ont lieu toutes les opérations situées à l'extérieur d'une fonction.
- ⇝ Lorsqu'un symbole est appelé,
  - ⇝ R le cherche dans l'environment courrant.
  - ⇝ S'il ne le trouven pas, il le cherche dans le parent de cet environment et ainsi de suite.

Environment

- ⇝ À quelques détails près, un environment peu être vu comme une simple liste.
- ⇝ En particulier, presque tout environment possède un environment parent.
  - ⇝ Le seul orphelin est `R_EmptyEnv`, *l'ancêtre de tous les environments*.
  - ⇝ Un autre environment important est `R_GlobalEnv` où ont lieu toutes les opérations situées à l'extérieur d'une fonction.
- ⇝ Lorsqu'un symbole est appelé,
  - ⇝ R le cherche dans l'environment courrant.
  - ⇝ S'il ne le trouven pas, il le cherche dans le parent de cet environment et ainsi de suite.
  - ⇝ Arrivé à `R_EmptyEnv`, R sait que le symbole n'est pas défini.

Environment

`<-` (Assignment)

Crée un symbole dans l'environment dans lequel il est appelé.

Environment

`<-` (Assignment)
: Crée un symbole dans l'environment dans lequel il est appelé.

`<<-` (Super assignment)
: "Modifie" la valeur d'un symbole situé dans un des environment parent.

Environment

`<-` (Assignment)
: Crée un symbole dans l'environment dans lequel il est appelé.

`<<-` (Super assignment)
: "Modifie" la valeur d'un symbole situé dans un des environment parent.

⇝ Si un symbole n'est pas défini, `<<-` le créera dans `R_GlobalEnv`.

Environment

`<-` (Assignment)
: Crée un symbole dans l'environment dans lequel il est appelé.

`<<-` (Super assignment)
: "Modifie" la valeur d'un symbole situé dans un des environment parent.

- ⇝ Si un symbole n'est pas défini, `<<-` le créera dans `R_GlobalEnv`.
- ⇝ Si le symbole est une variable, on parle alors de *variable globale*.

Portée lexicale

- ⇝ À chaque fois qu'une fonction est appelée, un nouvel environment est construit.

Portée lexicale

- ⇝ À chaque fois qu'une fonction est appelée, un nouvel environment est construit.
- ⇝ Comme les environments sont ordonnés, cela permet l'implémentation du concept de *portée lexicale*.

Portée lexicale

- ⇝ À chaque fois qu'une fonction est appelée, un nouvel environment est construit.
- ⇝ Comme les environments sont ordonnés, cela permet l'implémentation du concept de *portée lexicale*.

```
> x <- 42
> f <- function(){
+     print(x)
+     x <- 666
+     print(x)
+ }
> f()
[1] 42
[1] 666
> print(x)
[1] 42
```

Variables globales

⇝ En général, l'utilisation de variables globales devrait être évitée.

Variables globales

- ⇝ En général, l'utilisation de variables globales devrait être évitée.
  - ⇝ Moins portable.

Variables globales

- ⇝ En général, l'utilisation de variables globales devrait être évitée.
  - ⇝ Moins portable.
  - ⇝ Plus difficile à comprendre.

Variables globales

- ⇝ En général, l'utilisation de variables globales devrait être évitée.
  - ⇝ Moins portable.
  - ⇝ Plus difficile à comprendre.
  - ⇝ Plus difficile à déboguer.

Variables globales

- ⇝ En général, l'utilisation de variables globales devrait être évitée.
  - ⇝ Moins portable.
  - ⇝ Plus difficile à comprendre.
  - ⇝ Plus difficile à déboguer.
  - ⇝ Moins performant.

Variables globales

- ⇝ En général, l'utilisation de variables globales devrait être évitée.
  - ⇝ Moins portable.
  - ⇝ Plus difficile à comprendre.
  - ⇝ Plus difficile à déboguer.
  - ⇝ Moins performant.
- ⇝ Variables globales : souvent utilisées pour modifier le comportement d'une fonction.

Variables globales

- ⇝ En général, l'utilisation de variables globales devrait être évitée.
  - ⇝ Moins portable.
  - ⇝ Plus difficile à comprendre.
  - ⇝ Plus difficile à déboguer.
  - ⇝ Moins performant.
- ⇝ Variables globales : souvent utilisées pour modifier le comportement d'une fonction.
- ⇝ Il existe de meilleures alternatives.

Closure

- ⇝ Par défaut, les fonctions sont déclarées dans `R_GlobalEnv`.

Closure

- ⇝ Par défaut, les fonctions sont déclarées dans `R_GlobalEnv`.
- ⇝ La seule manière de leur donner un état est de faire appel à des variables globales.

Closure

- ⇝ Par défaut, les fonctions sont déclarées dans `R_GlobalEnv`.
- ⇝ La seule manière de leur donner un état est de faire appel à des variables globales.
- ⇝ Pour éviter cela, il faudrait déclarer une fonction dans son propre environment.

Closure

- ⇝ Par défaut, les fonctions sont déclarées dans `R_GlobalEnv`.
- ⇝ La seule manière de leur donner un état est de faire appel à des variables globales.
- ⇝ Pour éviter cela, il faudrait déclarer une fonction dans son propre environment.
- ⇝ Une manière de procéder est de faire appel a une *closure*.

Closure

- ⇝ Par défaut, les fonctions sont déclarées dans `R_GlobalEnv`.
- ⇝ La seule manière de leur donner un état est de faire appel à des variables globales.
- ⇝ Pour éviter cela, il faudrait déclarer une fonction dans son propre environment.
- ⇝ Une manière de procéder est de faire appel a une *closure*.
- ⇝ De manière informelle, closure $\simeq$ fonction dans une fonction.

Closure

- ⇝ Par défaut, les fonctions sont déclarées dans `R_GlobalEnv`.
- ⇝ La seule manière de leur donner un état est de faire appel à des variables globales.
- ⇝ Pour éviter cela, il faudrait déclarer une fonction dans son propre environment.
- ⇝ Une manière de procéder est de faire appel a une *closure*.
- ⇝ De manière informelle, closure $\simeq$ fonction dans une fonction.
- ⇝ On exploite le fait qu'un nouvel environment est créé lors de l'appel d'une fonction.

Variable globale vs. closure

Variable globale

```
seed <- 42
lcg_bad <- function(a, b, m)
  function(){
    ret <- (a * seed + b) %% m
    seed <<- ret
    ret / m
  }
```

Closure

```
lcg <- function(a, b, m,
                seed = 0)
  function(){
    ret <- (a * seed + b) %% m
    seed <<- ret
    ret / m
  }
```

```
> nr_lcg <- lcg(1664525, 1013904223, 2^32)
> identical(environment(lcg), globalenv())
[1] TRUE
> identical(environment(nr_lcg), globalenv())
[1] FALSE
```

Fonction et performance

Pour beaucoup de langages de programmation (particulièrement les langages compilés),

$\rightsquigarrow$ Le coût de l'appel d'une fonction est négligeable.

Fonction et performance

Pour beaucoup de langages de programmation (particulièrement les langages compilés),

- $\rightsquigarrow$ Le coût de l'appel d'une fonction est négligeable.
- $\rightsquigarrow$ Bien diviser le code en fonction = plus de possibilités d'optimisation par le compilateur.

Fonction et performance

Pour beaucoup de langages de programmation (particulièrement les langages compilés),

- $\rightsquigarrow$ Le coût de l'appel d'une fonction est négligeable.
- $\rightsquigarrow$ Bien diviser le code en fonction $=$ plus de possibilités d'optimisation par le compilateur.

Donc, on ne se pose pas de question. Pour R,

Fonction et performance

Pour beaucoup de langages de programmation (particulièrement les langages compilés),

- ⇝ Le coût de l'appel d'une fonction est négligeable.
- ⇝ Bien diviser le code en fonction = plus de possibilités d'optimisation par le compilateur.

Donc, on ne se pose pas de question. Pour R,

- ⇝ L'appel de fonction est coûteux (construction de l'environment...).

Fonction et performance

Pour beaucoup de langages de programmation (particulièrement les langages compilés),

- ⇝ Le coût de l'appel d'une fonction est négligeable.
- ⇝ Bien diviser le code en fonction = plus de possibilités d'optimisation par le compilateur.

Donc, on ne se pose pas de question. Pour R,

- ⇝ L'appel de fonction est coûteux (construction de l'environment...).
- ⇝ Il n'y a pas de possibilité d'optimisation.

Fonction et performance

Pour beaucoup de langages de programmation (particulièrement les langages compilés),

- ⇝ Le coût de l'appel d'une fonction est négligeable.
- ⇝ Bien diviser le code en fonction = plus de possibilités d'optimisation par le compilateur.

Donc, on ne se pose pas de question. Pour R,

- ⇝ L'appel de fonction est coûteux (construction de l'environment...).
- ⇝ Il n'y a pas de possibilité d'optimisation.
- ⇝ Faut-il quand même écrire des fonctions ? *Absolument !*

Fonction et performance

Pour beaucoup de langages de programmation (particulièrement les langages compilés),

- ⇝ Le coût de l'appel d'une fonction est négligeable.
- ⇝ Bien diviser le code en fonction = plus de possibilités d'optimisation par le compilateur.

Donc, on ne se pose pas de question. Pour R,

- ⇝ L'appel de fonction est coûteux (construction de l'environment...).
- ⇝ Il n'y a pas de possibilité d'optimisation.
- ⇝ Faut-il quand même écrire des fonctions ? *Absolument !*
- ⇝ Toutefois, dans certaines parties critiques du code, il peut être utile d'appeler le moins de fonctions possible.

Exemple

Approche impérative

```
add1 <- function(x) x + 1

add2 <- function(x) x + 2

for_add1_twice <- function(n){
    res <- integer(n)
    for (kk in 1:n)
        res[n] <- add1(add1(rpois(1, 1)))

    res
}

for_add2 <- function(n){
    res <- integer(n)
    for (kk in 1:n)
        res[n] <- add2(rpois(1, 1))

    res
}
```

Exemple

```
for_noCall <- function(n){
    res <- integer(n)
    for (kk in 1:n)
        res[n] <- rpois(1, 1) + 2

    res
}
```

```
> microbenchmark(for_add1_twice(1e4),
                 for_add2(1e4),
                 for_noCall(1e4))
Unit: milliseconds
                expr   min    lq  mean median    uq   max neval
 for_add1_twice(1e4) 20.19 20.95 22.93  23.73 24.44 28.83   100
       for_add2(1e4) 16.95 17.88 19.42  18.50 21.18 25.24   100
    for_noCall(1e4) 14.30 14.67 16.70  15.32 18.22 49.97   100
```

Exemple

Approche fonctionnelle

```
1  sapply_add1_twice <- function(n)
2      sapply(rpois(n, 1), function(x) add1(add1(x)))
3
4  sapply_add2 <- function(n) sapply(rpois(n, 1), add2)
5
6  sapply_noCall <- function(n) sapply(rpois(n, 1), `+`, y = 1)
```

```
7  > microbenchmark(sapply_add1_twice(1e4),
8                   sapply_add2(1e4),
9                   sapply_noCall(1e4))
10 Unit: milliseconds
11                    expr   min    lq  mean median    uq   max neval
12  sapply_add1_twice(1e4) 10.55 12.07 13.64  12.72 14.80 37.13   100
13        sapply_add2(1e4)  5.28  5.83  6.69   6.26  6.97 15.53   100
14     sapply_noCall(1e4)  3.57  3.89  4.41   4.16  4.45  9.22   100
```

Exemple

Approche array

```
add2_correct <- function(n) rpois(n, 1) + 2
```

```
> microbenchmark(for_add1_twice(1e4),
                 for_add2(1e4),
                 for_noCall(1e4),
                 sapply_add1_twice(1e4),
                 sapply_add2(1e4),
                 sapply_noCall(1e4),
                 add2_correct(1e4))
Unit: microseconds
                   expr   min    lq  mean median    uq   max neval
    for_add1_twice(1e4) 20049 21202 23115  21889 25029 30717   100
          for_add2(1e4) 16993 18182 20276  20327 22165 28140   100
        for_noCall(1e4) 14192 14747 17172  15402 18683 56314   100
 sapply_add1_twice(1e4) 10392 11379 12213  11721 12402 16501   100
       sapply_add2(1e4)  5015  5587  5983   5866  6129 10029   100
     sapply_noCall(1e4)  3341  3680  4004   3891  4132  8193   100
      add2_correct(1e4)   366   370   394    375   380  1641   100
```

Conclusion

$\rightsquigarrow$ Les boucles ne sont pas particulièrement lentes en R.

Conclusion

- ⇝ Les boucles ne sont pas particulièrement lentes en R.
  - ⇝ Toutefois, le type d'algorithme implicite à l'utilisation de boucles n'est pas bien géré par R.

Conclusion

- ⇝ Les boucles ne sont pas particulièrement lentes en R.
  - ⇝ Toutefois, le type d'algorithme implicite à l'utilisation de boucles n'est pas bien géré par R.
- ⇝ Les performances d'un programme écrit en R sont limitées.

Conclusion

- ⇝ Les boucles ne sont pas particulièrement lentes en R.
  - ⇝ Toutefois, le type d'algorithme implicite à l'utilisation de boucles n'est pas bien géré par R.
- ⇝ Les performances d'un programme écrit en R sont limitées.
  - ⇝ Toutefois, l'objectif premier de R n'est pas la performance ; on ne juge pas un poisson rouge à sa capacité à grimper dans les arbres !

Conclusion

- ⇝ Les boucles ne sont pas particulièrement lentes en R.
  - ⇝ Toutefois, le type d'algorithme implicite à l'utilisation de boucles n'est pas bien géré par R.
- ⇝ Les performances d'un programme écrit en R sont limitées.
  - ⇝ Toutefois, l'objectif premier de R n'est pas la performance ; on ne juge pas un poisson rouge à sa capacité à grimper dans les arbres !
  - ⇝ De plus, il est relativement facile d'appeler des fonctions contenues dans des librairies partagées à partir de R.

Conclusion

- ⇝ Les boucles ne sont pas particulièrement lentes en R.
  - ⇝ Toutefois, le type d'algorithme implicite à l'utilisation de boucles n'est pas bien géré par R.
- ⇝ Les performances d'un programme écrit en R sont limitées.
  - ⇝ Toutefois, l'objectif premier de R n'est pas la performance ; on ne juge pas un poisson rouge à sa capacité à grimper dans les arbres !
  - ⇝ De plus, il est relativement facile d'appeler des fonctions contenues dans des librairies partagées à partir de R.
- ⇝ Un langage n'est pas meilleur qu'un autre parce qu'il est plus performant.

Conclusion

- ⇝ Les boucles ne sont pas particulièrement lentes en R.
  - ⇝ Toutefois, le type d'algorithme implicite à l'utilisation de boucles n'est pas bien géré par R.
- ⇝ Les performances d'un programme écrit en R sont limitées.
  - ⇝ Toutefois, l'objectif premier de R n'est pas la performance ; on ne juge pas un poisson rouge à sa capacité à grimper dans les arbres !
  - ⇝ De plus, il est relativement facile d'appeler des fonctions contenues dans des librairies partagées à partir de R.
- ⇝ Un langage n'est pas meilleur qu'un autre parce qu'il est plus performant.
  - ⇝ On choisit un langage pour un problème particulier en fonction des objectifs qu'on souhaite atteindre.

Conclusion

- ⇝ Il peut être utile d'implémenter certaines parties critique d'un programme R dans un autre language.

Conclusion

- ⇝ Il peut être utile d'implémenter certaines parties critique d'un programme R dans un autre language.
  - ⇝ Toutefois, l'appel à une librairie partagée ou l'exécution d'une commande externe a un coût qu'il ne faut pas négliger.

Conclusion

- ⇝ Il peut être utile d'implémenter certaines parties critique d'un programme R dans un autre language.
  - ⇝ Toutefois, l'appel à une librairie partagée ou l'exécution d'une commande externe a un coût qu'il ne faut pas négliger.
  - ⇝ De plus, le temps de développement peut augmenter *drastiquement*.

Conclusion

- ⇝ Il peut être utile d'implémenter certaines parties critique d'un programme R dans un autre language.
  - ⇝ Toutefois, l'appel à une librairie partagée ou l'exécution d'une commande externe a un coût qu'il ne faut pas négliger.
  - ⇝ De plus, le temps de développement peut augmenter *drastiquement*.
- ⇝ Quelques conseils généraux pour du code performant :

Conclusion

- ⇝ Il peut être utile d'implémenter certaines parties critique d'un programme R dans un autre language.
  - ⇝ Toutefois, l'appel à une librairie partagée ou l'exécution d'une commande externe a un coût qu'il ne faut pas négliger.
  - ⇝ De plus, le temps de développement peut augmenter *drastiquement*.
- ⇝ Quelques conseils généraux pour du code performant :
  - ⇝ Utiliser des opérations vectorisées.

Conclusion

- ⇝ Il peut être utile d'implémenter certaines parties critique d'un programme R dans un autre language.
  - ⇝ Toutefois, l'appel à une librairie partagée ou l'exécution d'une commande externe a un coût qu'il ne faut pas négliger.
  - ⇝ De plus, le temps de développement peut augmenter *drastiquement*.
- ⇝ Quelques conseils généraux pour du code performant :
  - ⇝ Utiliser des opérations vectorisées.
  - ⇝ Éviter de modifier les variables.

Conclusion

- ⇝ Il peut être utile d'implémenter certaines parties critique d'un programme R dans un autre language.
  - ⇝ Toutefois, l'appel à une librairie partagée ou l'exécution d'une commande externe a un coût qu'il ne faut pas négliger.
  - ⇝ De plus, le temps de développement peut augmenter *drastiquement*.
- ⇝ Quelques conseils généraux pour du code performant :
  - ⇝ Utiliser des opérations vectorisées.
  - ⇝ Éviter de modifier les variables.
  - ⇝ Écrire des boucles courtes appelant peu de fonctions.

Conclusion

- ⇝ Il peut être utile d'implémenter certaines parties critique d'un programme R dans un autre language.
  - ⇝ Toutefois, l'appel à une librairie partagée ou l'exécution d'une commande externe a un coût qu'il ne faut pas négliger.
  - ⇝ De plus, le temps de développement peut augmenter *drastiquement*.
- ⇝ Quelques conseils généraux pour du code performant :
  - ⇝ Utiliser des opérations vectorisées.
  - ⇝ Éviter de modifier les variables.
  - ⇝ Écrire des boucles courtes appelant peu de fonctions.
  - ⇝ Éviter les fonctions récursives.

Références

[1] Phil Johnson. *The most WTF-y programming languages*. en. Sept. 2013. URL : https://www.itworld.com/article/2833252/the-most-wtf-y-programming-languages.html.

[2] *TIOBE Index | TIOBE - The Software Quality Company*. URL : https://www.tiobe.com/tiobe-index/.

[3] *What are the Most Disliked Programing Languages ? | Stack Overflow*. en-US. Oct. 2017. URL : https://stackoverflow.blog/2017/10/31/disliked-programming-languages/.

[4] Hadley Wickham. *Advanced R*. URL : https://adv-r.hadley.nz/.