2

Nomes e Valores[*]

2.1 Intodução

No R, é importante entender a distinção entre um objeto e seu nome. Isso te ajudará a:

- Prever mais acertadamente o desempenho e o uso da memória do seu código.
- Escrever códigos mais rápido evitando cópias acidentais, que é o principal motivo de códigos lentos.
- Entender melhor as ferramentas de programação funcional do R.

O objetivo deste capítulo é ajudá-lo a entender a diferença entre nomes e valores, e quando R copiará um objeto.

Quiz

Responda as seguintes questões para ver se você pode pular esse capítulo tranquilamente. Você encontrará as respostas no final destr capítulo na **Seção 2.7**.

1. Dado o seguinte data frame, como eu crio uma nova coluna chamada "3" que contenha a soma de 1 e de 2? Você pode usar apenas $, e não [[. O que faz de 1, 2 e 3 nomes de variáveis desafiadoras?

```
df <- data.frame(runif(3), runif(3))
names(df <- c(1, 2))
```

2. No código abaixo, quanta memória y ocupa?

```
x <- runif(1e6)
y <- list(x, x, x)
```

3. Em qual linha do exemplo abaixo a é copiada?

```
a <- c(1, 5, 3, 2)
b <- a
b[[1]] <- 10
```

[*]Esta é uma tradução livre de autoria de Antônio Fernando Costa Pella do Capítulo 2 do livro *Advanced R* de Hadley Wickham, 2ed.

Resumo

- A **Seção 2.2** introduz a diferença entre nomes e valores e discute como <- cria a ligação entre um nome e um valor.
- A **Seção 2.3** descreve quando R faz uma cópia: sempre que você modifica um vetor, você está certamente criando um vetor modificado. Você aprenderá como usar `tracemem()` para descobrir quando uma nova cópia realmente acontece. Você então explorará como esses implicações se aplicam a funções, listas, data frames e vetores de caracteres.
- A **Seção 2.4** explora as implicações das duas seções anteriores em quanto de memória um objeto ocupa. Desde que sua intuição esteja profundamente errada e `utils::object.size()` é impreciso, você aprenderá como usar `lobster::obj_size()`.
- A **Seção 2.5** apresenta as duas importantes exceções do *copy-on-modify*: com ambientes e valores com um único nome, objetos são na verdade modificados no local.
- Por fim, a **Seção 2.6** conclui o capítulo com uma discussão do *garbage collector*, que limpa a memória usada por objetos que não tem mais um nome.

Pré-requisitos

Usaremos o pacote lobstr (`https://github.com/r-lib/lobstr`) para entrar na representação interna dos objetos do R.

```
library(lobstr)
```

Fontes

Os detalhes do gerenciamento de memória no R não estão documentados em um único lugar. Muito da informação deste capítulo foi obtida de uma leitura profunda da documentação (particularmente `?Memory` e `?gc`), da seção *memory profiling* (`http://cran.r-project.org/doc/manuals/R-exts.html#Profiling-R-code-for-memory-use`) do *Writing R extentions* [R Core Team, 2018a] e da seção SEXPs (`http://cran.r-project.org/doc/manuals/R-ints.html#SEXPs`) do *R internals* [R Core Team, 2018b]. O resto eu descobri lendo o código fonte em C, fazendo pequenos experimentos e fazendo perguntas no R-devel. Qualquer erro é inteiramente meu.

2.2 O Básico sobre Atribuição

Considere o seguinte código:

```
x <- c(1, 2, 3)
```

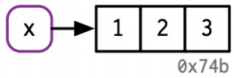
É fácil lê-lo como: "cria um objeto chamado 'x' que contém os valores 1, 2 e 3". Infelizmente, essa é uma simplificação que levará a uma imprecisão sobre o que o R realmente está fazendo por trás. É mais preciso dizer que este código está fazendo duas coisas:

- Está criando um objeto, um vetor de valores, `c(1, 2, 3)`.

- E está vinculando este objeto a um nome, `x`.

Em outras palavras, o objeto, ou valor, não tem um nome; na verdade é o nome que tem um valor.

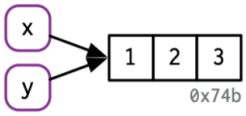
Para esclarecer ainda mais essa diferença, desenharei diagramas como este:



O nome, `x`, está desenhado com um retângulo arredondado. Ele tem uma seta que aponta para (ou vincula, ou referencia) o valor, o vetor `c(1, 2, 3)`. A seta aponta para a direção oposta ao sinal de atribuição: `<-` cria uma vinculação do nome na no lado esquerdo com o objeto no lado direto.

Assim, você pode pensar no nome como uma referência ao valor. Por exemplo, se você executa este código, você não faz uma cópia do valor `c(1, 2, 3)`, você faz uma nova atribuição para o objeto já existente:

```
y <- x
```



Você pode ter percebido que o valor `c(1, 2, 3)` tem um rótulo: `0x74b`. Enquanto o vetor não tiver um nome, eu irei ocasionalmente precisar me referir a um objeto independentemente se ele tiver um nome. Para tornar isso possível, eu irei rotular os valores com um identificador único. Esses identificadores têm uma forma especial que parece com o "endereço" de memória do objeto, isto é, a localização na memória onde o objeto está armazenado. Porém, como o endereço na memória muda toda vez que o código é executado, precisaremos identificá-los.

Você pode acessar um identificador de um objeto usando `lobstr::obj_addr()`. Fazendo isso permite você ver que tanto `x` quanto `y` têm o mesmo identificador:

```
obj_addr(x)
## [1] "0x18b1b588"
obj_addr(y)
## [1] "0x18b1b588"
```

Esses identificadores são longos, e mudam toda vez que você reinicia o R.

Pode levar um tempo para fazer sua cabeça entender a diferença entre nomes e valores, mas entender isto é realmente útil na programação funcional, onde funções podem ter diferentes nomes em diferentes contexos.

2.2.1 Nomes não-sintáticos

O R regras estritas sobre o que constitui um nome válido. Um nome **sintático** deve consistir de letras[1], números, "." e "_", mas não pode começar com "_" e nem com um

[1]Surpreendentemente, o que constitui precisamente uma palavra é determinado pelo local. Isto significa que a sintaxe do R pode na verdade ser diferente de computador para computador, e é possível que um arquivo funcione em um computador e nem seja entendido em outro. Evite este problema configurando para caracteres ASCII (isto é, A-Z) o máximo possível.

número.Adicionalmente, você não pode usar nenhuma das **palavras reservadas**, como `TRUE`, `NULL`, `if` e `function` (ver a lista completa em `?Reserved`). Um nome que não segue estas rgras é um nome **não-sintático**; se você tentar usá-los, você terá um erro:

```
_abc <- 1
#> Error: unexpected input in "_"

if <- 10
#> Error: unexpected assignment in "if <-"
```

É possível sobrepor essas regras e usar um nome, isto é, uma sequência de caracteres, colocando-o entre crases:

```
`_abc` <- 1
`_abc`
#> [1] 1

`if` <- 10
`if`
#> [1] 10
```

Mesmo que seja improvável que você deliberadamente crie esses nomes confusos, é preciso entender como eles funcionam, pois você poderá se deparar com eles, e ocorre mais frequentemente quando você carrega dados que foram criados fora do R.

> Você *pode* também criar atribuições não-sintáticas usando aspas simples ou duplas (e.g `"_abc" <- 1`) ao invés de crase, mas não deve, pois você terá que usar diferentes sintaxes para recuperar os valores. A capacidade de usar *strings* no lado esquerdo do símbolo de atribuição tem raizes históricas, antes de o R aderir as aspas.

2.2.2 Exercícios

1. Explique a relação entre `a, b, c` e `d`no seguinte código:

```
a <- 1:10
b <- a
c <- b
d <- 1:10
```

2. O seguinte código acessa a função `mean` de várias maneiras. Todos eles apintam para o mesmo objeto de função subjascente? Verifique isto com `lobstr::obj_addr()`.

```
mean
base::mean
get("mean")
evalq(mean)
match.fun("mean")
```

3. Por *default*, as funções de importação de dados do base do R, como o `read.csv()`, irão automaticamente converter nomes não-sintáticos em nomes sintáticos. Por quê isso pode vir a ser um problema? Qual opção nos permite contornar este problema?

4. Quais regras o `make.names()` usa para converter nomes não-sintáticos em nomes sintáticos?

5. Eu simplifiquei um pouco as regras que guiam os nomes sintáticos. Por quê `.123e1` não é um nome sintático? Leia `?make.names` para maiores detalhes.

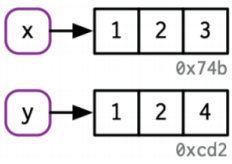
2.3 *Copy-on-modify*

Considere o seguinte código. Ele atribui `x` e `y` ao mesmo valor subjascente e então altera `y`[2].

```
x <- c(1, 2, 3)
y <- x

y[[3]] <- 4
x
#> [1] 1 2 3
```

Modificar `y` claramente não modifica `x`. Então o que aconteceu com o vetor compartilhado? Enquanto o valor associado a `y` mudou, o objeto original não. Ao invés disso, o R criou um novo objeto, `0xcd2`, uma cópia de `0x74b` com um valor diferente, então reatribuiu `y` a esse objeto.



Esse comportamento é chamado de ***copy-on-modify***. Entender isso melhorará radicalmente a sua intuição sobre o desempenho da programação em R. Uma forma parecida de descrever esse comportamento é dizer que os objetos do R são **imutáveis**. Contudo, irei geralmente evitar esse termo pois existem algumas exceções importantes no *copy-on-modify* que você aprenderá na **Seção 2.5**.

Quando estiver explorando o comportamento do *copy-on-modify*, saiba que você terá alguns resultados diferentes no RStudio. Isso ocorre pois o painel *environment* deve fazer a referência a cada objeto de forma a formar mostrar essa informação. Isto distorce a sua exploração interativa mas não afeta o código interno das funções, e por isso não afeta o desempenho durante a análise dos dados. Para experimentar, recomendo tanto executar o código diretamento do terminal ou usando o RMarkdown (como este livro).

2.3.1 `tracemem()`

Você pode ver um objeto ser copiado com a ajuda de `base::tracemem()`. Uma vez chamada essa função, você terá o o endereço atual do objeto:

[2] Você pode estar surpreso em ver `[[` sendo usado em um subconjunto de um vetor numérico. Nós iremos retornar a isto na **Seção 4.3**, mas resumidamente, eu acho que você deva sempre usar `[[` quando você estiver obtendo um único elemento.

```
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
#> <0x7f80c0e0ffc8>
```

Daí em diante, toda vez que esse objeto for copiado, `tracemem()` mostrará uma mensagem dizendo qual objeto foi copiado, seu novo endereço na memória e a sequência de requisições que levaram à cópia:

```
y <- x
y[[3]] <- 4L
#> tracemem[0x7f80c0e0ffc8 -> 0x7f80c4427f40]:
```

Se você modificar `y` de novo, ele não será copiado. Isso ocorre pois o novo objeto agora só tem um único nome vinculado a ele, então o R aplicada *modify-in-place*. Voltaremos a isto na **Seção 2.5**.

```
y[[3]] <- 5L

untracemem(y)
```

O `untrecemem()`é o oposto de `tracemem()`; ele para de rastrear.

2.3.2 Chamada de funções

As mesmas regras para copiar também se aplicam para a chamada de funções. Observe este código:

```
f <- function(a){
  a
}

x <- c(1, 2, 3)
cat(tracemem(x), "\n")
#> <0x7f8e10fd5d38>

z <- f(x)
# there's no copy here!

untracemem(x)
```

Enquanto que `f()` está sendo executado, a variável `a` dentro da funções aponta para o mesmo valor