Aula 2 - O básico de programação em R

Profa. Yana Borges

Fevereiro de 2022

1 Programação em R

R é, no fundo, uma calculadora científica sobrecarregada, por isso tem um conjunto bastante abrangente de capacidades matemáticas incorporadas. Nesta aula iremos aprender a trabalhar com operadores aritméticos, funções matemáticas comuns e operadores relacionais, e mostrar a você como atribuir um valor a uma variável.

1.1 Operações matemáticas e vetores

O operador `+` faz adição, mas tem um truque especial: além de somar dois números, você pode usá-lo para somar dois vetores. Um vetor é um conjunto ordenado de valores. Os vetores são tremendamente importantes em estatística, pois geralmente você deseja analisar um conjunto de dados inteiro em vez de apenas um dado.

O operador de dois pontos, `:`, cria uma sequência de um número para o próximo, e a função `c` concatena valores, neste caso para criar vetores (concatenar é uma palavra latina que significa "conectar em uma cadeia") .

Nomes de variáveis diferenciam maiúsculas de minúsculas em R, portanto, a função `C` faz algo completamente diferente de `c`:

```
1:5 + 6:10 # olha, sem laços!
```

```
## [1]  7  9 11 13 15
```

```
c(1, 3, 6, 10, 15) + c(0, 1, 3, 6, 10)
```

```
## [1]  1  4  9 16 25
```

Existem alguns outros conflitos de nomes: `filter` e `Filter`, `find` e `Find`, `gamma` e `Gamma`, `nrow/ncol` e `NROW/NCOL`.

Vetorização tem vários significados em R, o mais comum deles é que um operador ou uma função atuará em cada elemento de um vetor sem a necessidade de você escrever explicitamente um loop. (Esse loop implícito embutido sobre elementos também é muito mais rápido do que escrever explicitamente seu próprio loop.) Um segundo significado de vetorização é quando uma função recebe um vetor como entrada e calcula um resumo estatístico:

```
sum(1:5)
```

```
## [1] 15
```

```
mean(1:5)
```

```
## [1] 3
```

Um terceiro caso de vetorização é a vetorização sobre argumentos. É quando uma função calcula um resumo estatístico de vários de seus argumentos de entrada. A função `sum` faz isso.

```r
sum(1, 2, 3, 4, 5)
```

```
## [1] 15
```

Todos os operadores aritméticos em R, não apenas mais (`+`), são vetorizados. Os exemplos a seguir demonstram subtração, multiplicação, exponenciação e dois tipos de divisão, bem como o resto após a divisão:

```r
c(2, 3, 5, 7, 11, 13) - 2 #subtração
```

```
## [1]  0  1  3  5  9 11
```

```r
-2:2 * -2:2 #multiplicação
```

```
## [1] 4 1 0 1 4
```

```r
identical(2 ^ 3, 2 ** 3) #podemos usar ^ ou ** para exponenciação embora ^ seja mais comum
```

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

```r
1:10 / 3
```

```
##  [1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667 2.0000000 2.3333333
##  [8] 2.6666667 3.0000000 3.3333333
```

```r
1:10 %/% 3 #divisão inteira
```

```
##  [1] 0 0 1 1 1 2 2 2 3 3
```

```r
1:10 %% 3 #resto de divisão
```

```
##  [1] 1 2 0 1 2 0 1 2 0 1
```

R também contém uma ampla seleção de funções matemáticas. Você obtém trigonometria (`sin`, `cos`, `tan` e seus inversos `asin`, `acos` e `atan`), logaritmos e expoentes (`log` e `exp`, e suas variantes `log1p` e `expm1` que calculam log(1 + x) e exp(x - 1) mais precisamente para valores muito pequenos de x), e quase qualquer outra função matemática que você possa imaginar. Os exemplos a seguir fornecem uma dica do que está em oferta. Novamente, observe que todas as funções operam naturalmente em vetores em vez de apenas valores únicos:

```r
cos(c(0, pi / 4, pi / 2, pi)) #pi é uma constante interna
```

```
## [1]  1.000000e+00  7.071068e-01  6.123032e-17 -1.000000e+00
```

```r
exp(pi * 1i) + 1 #fórmula de Euler
```

```
## [1] 0+1.224606e-16i
```

```r
factorial(7) + factorial(1) - 71 ^ 2 #
```

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

```r
choose(5, 0:5)
```

```
## [1]  1  5 10 10  5  1
```

Para comparar valores inteiros para igualdade, use `==`. Não use um único `=` já que é usado para outra coisa, como veremos em breve. Assim como os operadores aritméticos, `==` e os demais operadores relacionais são vetorizados. Para verificar a desigualdade, o operador "não é igual" é `!=`. Maior que e menor que são como você poderia esperar: `>` e `<` (ou `>=` e `<=` se a igualdade for permitida). Aqui estão alguns exemplos:

```r
c(3, 4 - 1, 1 + 1 + 1) == 3
```

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

```r
1:3 != 3:1
```

```
## [1]  TRUE FALSE  TRUE
```

```r
exp(1:5) < 100
```

```
## [1]  TRUE  TRUE  TRUE  TRUE FALSE
```

```r
(1:5) ^ 2 >= 16
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE
```

Comparar não inteiros usando == é problemático devido a arredondamentos. Vamos considerar estes dois números, que devem ser os mesmos:

```r
  sqrt(2) ^ 2 == 2 #sqrt é a função de raiz quadrada
```

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

```r
  sqrt(2) ^ 2 - 2  #este pequeno valor é o erro de arredondamento
```

```
## [1] 4.440892e-16
```

R também fornece a função `all.equal` para verificar a igualdade de números. Isso fornece um nível de tolerância (por padrão, cerca de 1,5e-8), para que erros de arredondamento menores que a tolerância sejam ignorados:

```r
all.equal(sqrt(2) ^ 2, 2)
```

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

Se os valores a serem comparados não forem os mesmos, `all.equal` retornará um relatório sobre as diferenças. Se você precisar de um valor TRUE ou FALSE, precisará usar a função `isTRUE` no comando `all.equall`:

```r
all.equal(sqrt(2) ^ 2, 3)
```

```
## [1] "Mean relative difference: 0.5"
```

```r
isTRUE(all.equal(sqrt(2) ^ 2, 3))
```

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

Dica: para verificar se dois números são iguais, não use ==. Em vez disso, use a função ***`all.equal`****.*

Também podemos usar == para comparar strings. Nesse caso, a comparação diferencia maiúsculas de minúsculas, portanto, as strings devem corresponder exatamente. Também é teoricamente possível comparar strings (string é uma sequência de caracteres, geralmente utilizada para representar palavras, frases ou textos de um programa) usando maior ou menor que (> e <):

```r
c(
"Falta", "falta", "faltou", "flauta", "feito", "Flauta",
 "faltar", "afeito", "falso", "feio", "falta?"
) == "falta"
```

```
##  [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```r
c("A", "B", "C", "D") < "C"
```

```
## [1]  TRUE  TRUE FALSE FALSE
```

```r
c("a", "b", "c", "d") < "C" #os resultados podem variar
```

```
## [1]  TRUE  TRUE  TRUE FALSE
```

Na prática, no entanto, a última abordagem é quase sempre uma péssima ideia, já que os resultados dependem da sua localidade (diferentes culturas estão cheias de regras de classificação estranhas para letras; em estoniano, “z” vem entre “s” e “t”).

1.1.1 Principais operadores e funções

As operações matemáticas básicas:

Operador	Função
+	Adição
-	Subtração
	Multiplicação
/	Divisão
^ ou **	Potência

Principais operadores relacionais e operadores lógicos:

Função	Descrição
<	Menor que
<=	Menor ou igual
>	Maior que
>=	Maior ou igual
==	Igual(comparação)
!=	Diferente
&	AND
!	NOT
|	OR
FALSE ou 0	Valor booleano falso (0)
TRUE ou 1	Valor booleano verdadeiro (1)

Outras funções:

Função	Descrição
abs(x)	valor absoluto de x
log(x,b)	logaritmo de x com base b
log(x)	logaritmo natural de x
log10(x)	logaritmo de x com base 10
exp(x)	exponencial de x
sin(x)	seno de x
cos(x)	cosseno de x
tan(x)	tangente de x
round(x, digits=n)	arredonda x com n decimais
ceiling(x)	arredonda x para o maior valor
floor(x)	arredonda x para o menor valor
sqrt(x)	raiz quadrada de x

1.2 Atribuindo variáveis

Na maioria das vezes queremos armazenar os resultados para reutilização. Podemos atribuir caracteres a uma variável (local) usando `<-` ou `=`, embora por razões históricas, prefiram usar `<-`:

```
x <- 1:5
y = 6:10
```

Agora podemos reutilizar esses valores em nossos cálculos adicionais:

```
x + 2 * y - 3
```

```
## [1] 10 13 16 19 22
```

Observe que não precisamos declarar quais tipos de variáveis x e y seriam antes de atribuir caracteres a elas (ao contrário da maioria das linguagens compiladas). Na verdade, não poderíamos ter declarado o tipo, já que tal conceito não existe em R.

Os nomes de variáveis podem conter letras, números, pontos e sublinhados, mas não podem começar com um número ou um ponto seguido por um número (já que se parece muito com um número). Palavras reservadas como "if" e "for" não são permitidas.

Os espaços ao redor dos operadores de atribuição não são obrigatórios, mas ajudam na legibilidade, especialmente com `<-`, para que possamos distinguir facilmente a atribuição do sinal de menos ou de menor que:

```
x <- 3
x < -3
```

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

```
x<-3
```

Observe que quando você atribui uma variável, você não vê o valor que foi dado a ela. Para ver qual valor uma variável contém, basta digitar seu nome no prompt de comando para imprimi-lo:

```
  x
```

```
## [1] 3
```

Atenção: Em alguns sistemas, por exemplo, executando R a partir de um terminal Linux, talvez seja necessário chamar explicitamente a função print para ver o valor. Neste caso, digite ***`print(x)`***.

Se você quiser atribuir um valor e imprimir tudo em uma linha, você tem duas possibilidades. Em primeiro lugar, você pode colocar várias instruções em uma linha, separando-as com um ponto e vírgula, `;`. Em segundo lugar, você pode colocar a atribuição entre parênteses, `()`. Nos exemplos a seguir, rnorm gera números aleatórios de uma distribuição normal e rlnorm os gera de uma distribuição lognormal:

```
z <- rnorm(5); z
```

```
## [1] -0.5983215  1.0815641  0.6982438  0.4029636 -1.9311126
```

```
(zz <- rlnorm(5))
```

```
## [1] 0.7855242 0.3951329 2.5363710 1.4013534 1.3011825
```

1.3 Números especiais

Para ajudar na aritmética, R suporta quatro valores numéricos especiais: `Inf`, `-Inf`, `NaN` e `NA`. Os dois primeiros são, é claro, infinito positivo e negativo, mas o segundo par precisa de um pouco mais de explicação. `NaN` é a abreviação de "not-a-number" e significa que nosso cálculo não fez sentido matemático ou não pôde ser realizado corretamente. `NA` é a abreviação de "not available" e representa um valor ausente – um problema muito comum na análise de dados. Em geral, se nosso cálculo envolver um valor ausente, os resultados também estarão ausentes:

```
c(Inf + 1, Inf - 1, Inf - Inf)
```

```
## [1] Inf Inf NaN
```

```r
c(1 / Inf, Inf / 1, Inf / Inf)
```

```
## [1]   0 Inf NaN
```

```r
c(sqrt(Inf), sin(Inf))
```

```
## Warning in sin(Inf): NaNs produzidos
```

```
## [1] Inf NaN
```

```r
c(log(Inf), log(Inf, base = Inf))
```

```
## Warning: NaNs produzidos
```

```
## [1] Inf NaN
```

```r
c(NA + 1, NA * 5, NA + Inf)
```

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

Quando a aritmética envolve `NA` e `NaN`, a resposta é um desses dois valores, mas qual desses dois, depende do sistema:

```r
c(NA + NA, NaN + NaN, NaN + NA, NA + NaN)
```

```
## [1]  NA NaN NaN  NA
```

Existem funções disponíveis para verificar esses valores especiais. Observe que `NaN` e `NA` não são finitos nem infinitos, e `NaN` está ausente, mas `NA` é um número:

```r
x <- c(0, Inf, -Inf, NaN, NA)
is.finite(x)
```

```
## [1]  TRUE FALSE FALSE FALSE FALSE
```

```r
is.infinite(x)
```

```
## [1] FALSE  TRUE  TRUE FALSE FALSE
```

```r
is.nan(x)
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE
```

```r
is.na(x)
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE
```

sqrt(3,2)

1.4 Vetores lógicos

Além dos números, o cálculo científico geralmente envolve valores lógicos, principalmente como resultado do uso dos operadores relacionais (<, tc.). Muitas linguagens de programação usam lógica booleana, onde os valores podem ser TRUE ou FALSE. Em R, a situação é um pouco mais complicada, pois também podemos ter valores ausentes, NA. Esse sistema de três estados às vezes é chamado de lógica trooleana, embora seja uma piada etimológica ruim, já que o "Bool" em "Boolean" vem de George Bool, em vez de qualquer coisa a ver com a palavra binário.

`TRUE` e `FALSE` são palavras reservadas em R: você não pode criar uma variável com nenhum desses nomes (versões minúsculas ou mistas como `True` são boas, no entanto). Quando você inicia R as variáveis `T` e `F` já estão definidas para você, tomando os valores TRUE e FALSE, respectivamente. Isso pode economizar um pouco de digitação, mas também pode causar grandes problemas. T e F não são palavras reservadas, então os usuários podem redefini-las. Isso significa que não há problema em usar os nomes abreviados se você estiver

digitando na linha de comando, mas não se o seu código for interagir com o de outra pessoa (especialmente se o código envolver Tempos ou Temperaturas ou Funções matemáticas).

Existem três operadores lógicos vetorizados em R:

- `!` é usado para *não*
- `&` é usado para *e*
- `|` é usado para *ou*

```
(x <- 1:10 >= 5 )
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
!x
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
(y <- 1:10 %% 2 == 0)
```

```
##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

```
x & y
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

```
x | y
```

```
##  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Podemos organizar algumas tabelas-verdade para ver como elas funcionam (não se preocupe se este código ainda não faz sentido; apenas concentre-se em entender por que cada valor ocorre na tabela-verdade):

```
x <- c(TRUE, FALSE, NA) #os três valores lógicos
xy <- expand.grid(x = x, y = x) #todas as combinações de x e y
within( #faça as próximas atribuições dentro de xy
xy,
{
e <- x & y
ou <- x | y
nao.y <- !y
nao.x <- !x
}
)
```

```
##       x     y nao.x nao.y    ou     e
## 1  TRUE  TRUE FALSE FALSE  TRUE  TRUE
## 2 FALSE  TRUE  TRUE FALSE  TRUE FALSE
## 3    NA  TRUE    NA FALSE  TRUE    NA
## 4  TRUE FALSE FALSE  TRUE  TRUE FALSE
## 5 FALSE FALSE  TRUE  TRUE FALSE FALSE
## 6    NA FALSE    NA  TRUE    NA FALSE
## 7  TRUE    NA FALSE    NA  TRUE    NA
## 8 FALSE    NA  TRUE    NA    NA FALSE
## 9    NA    NA    NA    NA    NA    NA
```

Duas outras funções úteis para lidar com vetores lógicos são `any` and `all`, que retornam `TRUE` se o vetor de entrada contiver pelo menos um valor `TRUE` ou apenas valores `TRUE`, respectivamente:

```
none_true <- c(FALSE, FALSE, FALSE) #nenhum é verdadeiro
some_true <- c(FALSE, TRUE, FALSE)  #algum é verdadeiro
```

```r
all_true <- c(TRUE, TRUE, TRUE)     #todos são verdadeiros
any(some_true) #algum é verdadeiro?
```

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

```r
any(all_true)  #algum é verdadeiro?
```

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

```r
all(none_true) #todos são verdadeiros?
```

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

```r
all(some_true) #todos são verdadeiros?
```

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

```r
all(all_true) #todos são verdadeiros?
```

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

1.5 Exercício

1. Calcule o arco tangente do inverso de todos os inteiros de 1 a 100.
2. Atribua os números de 1 a 100 a uma variável x. Calcule o arco tangente do inverso de x, como no tópico (1.), e atribua-a a uma variável y. Agora calcule o inverso da tangente de y e atribua esses valores a uma variável z. Arredonde os valores de y com duas casas decimais e guarde em k. Atribua a w o erro entre y e k. Desafio: qual o valor absoluto de w?
3. Compare as variáveis x e z do Exercício (2.) usando `==`, `identical` e `all.equal`. Para `all.equal`, tente alterar o nível de tolerância passando um terceiro argumento para a função. O que acontece se a tolerância for definida como 0 (use `all.equal(argumento1,argumento2, tolerance = 0)`)?