Exercícios 3

Funções

Em **R**, as funções são também objectos e podem ser manipuladas como tal. Têm três principais propriedades:

argumentos Lista dos parâmetros que a função vai receber, em que alguns podem ter um valor por defeito e ser opcionais. Uma forma rápida de consultar os argumentos de uma função é através da função `args`.

corpo Ao conjunto de expressões avaliado na execução da função chama-se corpo da função. É possível visualizar o corpo de uma função através da função `body`.

ambiente Uma função é definida num determinado ambiente (*environment*) activo, o que irá determinar os objectos visíveis e acessíveis pela função.

1. Crie uma função que faça a conversão de graus Fahrenheit (°F) para graus Celsius (°C) , sabendo que a sua relação é dada pela Equação 1.

$$^{\circ}C = (^{\circ}F - 32) \div 1,8 \tag{1}$$

   Teste a sua função para saber quantos graus Celsius correspondem a 159,3 °F, uma das temperaturas registadas mais altas de sempre na superfície da Terra, no deserto de Lut, no Irão.

Solução:

```
f2c = function(fhr) {
        cls = (fhr - 32) / 1.8
        return(cls)
}

> f2c(159.3)
[1] 70.72222
```

2. Expanda a sua função para, opcionalmente, também fazer a conversão de graus Celsius para Fahrenheit. A relação inversa é dada pela Equação 2.

$$^{\circ}F = (^{\circ}C \times 1.8) + 32 \tag{2}$$

Solução:

```
converte.temp = function(temp, to="C") {
        if (to == "C") {
                x = (temp - 32) / 1.8
```

```
        } else if (to == "F") {
                x = temp * 1.8 + 32
        } else {
                stop(cat("opção inválida: ", to, "\n"))
        }
        return(x)
}

> converte.temp(100)
[1] 37.77778
> converte.temp(100,"F")
[1] 212
> converte.temp(100,"A")
opção inválida:  A
Error in converte.temp(100, "A") :
```

3. Atente ao seguinte bloco de código:

```
x = 2
z = 56.3
f = function(x) {
      a = 34
      y = x / 4 * a * z
      y
    }
f(21)
a
```

Qual o valor das variáveis `a`, `x` e `z` dentro do ambiente da função? O que aconteceu ao valor da variável `a` após a execução da função?

Solução:

```
f = function(x) {
        a = 34
        y = x / 4 * a * z
        cat("Variável x: ", x, "\nVariável a: ", a, "\nVariável z: ", z, "\n")
        y
}

> x = 2
> z = 56.3
> f(21)
Variável x:  21
Variável a:  34
Variável z:  56.3
[1] 10049.55
```

Os valores visualizados correspondem aos valores que aquelas variáveis têm **dentro** da função:

- A variável `x` toma o valor passado como argumento da função. Não é o mesmo objecto que existe fora da função, embora tenha o mesmo nome (existem em ambientes diferentes).
- A variável `a` só existe dentro da função e tem o valor que recebe dentro desta.
- A variável `z` existe fora da função, mas quando acedida dentro da função, como não existe nenhum outro objecto com o mesmo nome, é procurado um objecto com esse nome no ambiente imediatamente acima do ambiente da função `f`.

4. Escreva três funções que recebam um `data.frame` e devolvam, por cada coluna:

   - o máximo;
   - o mínimo;
   - a média.

   Escreva duas versões para cada caso: uma recorrendo a um ciclo; e outra sem recorrer (explicitamente) a ciclos. Defina ainda o argumento opcional `first.n`, onde o utilizador poderá escolher para encontrar os referidos valores apenas considerando os primeiros n elementos de cada coluna. Teste as suas funções com o conjunto de dados `iris`.

Solução:

Versões recorrendo a ciclos:

```
# máximo, usando um ciclo while
maximo = function(tabela) {
        coluna = 1
        maxs = c()
        while (coluna <= ncol(tabela)) {
                maxs = c(maxs, max(tabela[ , coluna]))
                coluna = coluna + 1
        }
        return(maxs)
}
# mínimo, usando um ciclo for
minimo = function(tabela) {
        mins = vector()
        for (coluna in tabela) {
                mins = c(mins, min(coluna))
        }
        mins
}
# média, usando um ciclo repeat, começando na última coluna
media = function(tabela) {
        meds = NULL
        cols = ncol(tabela)
        repeat {
                if (!cols) {
                        break
                }
                meds = c(meds, mean(tabela[ , cols]))
```

```
                cols = cols - 1
        }
        return(meds)
}
```

Versões segundo o paradigma de programação funcional:

```
# máximo
maximo = function(tabela) {
        apply(tabela, 2, max)
}
# mínimo
minimo = function(tabela) {
        apply(tabela, 2, min)
}
# média
media = function(tabela) {
        apply(tabela, 2, mean)
}
```

Falta agora acrescentar a funcionalidade opcional em que o utilizador poderá escolher para considerar apenas os primeiros n elementos de cada coluna. Segue-se um exemplo para a última função mostrada. A alteração das restantes funções poderá ser feita de forma análoga.

```
# média com argumento opcional
media = function(tabela, first.n=nrow(tabela)) {
        apply(tabela[1:first.n, ], 2, mean)
}
```

Exemplo de aplicação com a tabela `iris`:

```
> media(iris[, -5])
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
5.843333     3.057333     3.758000     1.199333
> media(iris[, -5], 10)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
4.86         3.31         1.45         0.22
```

5. Assuma que gosta de coleccionar flores de Iris (Figura 1). Dispõe de uma tabela onde anota algumas das propriedades destas flores, registando, para cada exemplar, o valor de cada uma destas propriedades: comprimento e largura da pétala; comprimento e largura da sépala. A Tabela 1 mostra um excerto dessa tabela.

Número do examplar	Comprimento da Sépala	Largura da Sépala	Comprimento da Pétala	Largura da Pétala	Espécie
1	5,1	3,5	1,4	0,2	Setosa
51	7,0	3,2	4,7	1,4	Versicolor
101	6,3	3,3	6,0	2,5	Virginica

Tabela 1: Excerto da tabela com os registos das flores Iris, disponível em **R** na variável `iris`.

Após uma ida ao campo, uma nova amostra foi colhida. O novo exemplar tem as características indicadas

Figura 1: *Iris sibirica* (fonte: wikimedia.org).

na Tabela 2.

Número do examplar	Comprimento da Sépala	Largura da Sépala	Comprimento da Pétala	Largura da Pétala
Nova amostra	4,7	3,2	1,7	0,3

Tabela 2: Características da nova amostra de flor Iris.

O problema consiste em descobrir a que espécie pertence.

Este é um problema típico de classificação[1]. Noutra disciplina, aprenderá mais sobre os métodos de classificação. Por agora, pode resolver este problema com uma implementação simples do algoritmo dos k-vizinhos mais próximos[2]:

- Considere que cada característica tem o mesmo peso;
- Cada uma das características pode ser vista como um eixo (ou variável), então é como se tivesse quatro dimensões;
- A nova amostra será classificada como sendo da mesma espécie que o vizinho mais próximo ($k = 1$) no conjunto de dados.
- Considere a distância como a distância Euclidiana[3].

Resolva este problema aplicando uma função sobre o `data.frame` `iris`.

Solução:

Solução com uma linha de código:

```
> iris$Species[which.min(apply(iris[, -5], 1,
+ function(x) sum((x - c(4.7, 3.2, 1.7, 0.3))^2)))]
[1] setosa
Levels: setosa versicolor virginica
```

Agora em mais passos:

```
amostra = c(4.7, 3.2, 1.7, 0.3)
dist.euclidiana = function (vec1, vec2) {
        diff = vec1 - vec2
        square = diff^2
        return(sum(square))
```

[1] https://en.wikipedia.org/wiki/Statistical_classification
[2] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
[3] https://en.wikipedia.org/wiki/Euclidean_distance

```
}
distancias = apply(iris[, -5], 1, dist.euclidiana, vec2=amostra)
indice_do_minimo = which.min(distancias)
especie_amostra = iris$Species[indice_do_minimo]

> especie_amostra
[1] setosa
Levels: setosa versicolor virginica
```

Nota: não é necessário calcular a Distância Euclidiana propriamente dita, basta calcular o seu quadrado – a relação de ordem é preservada e o seu cálculo é menos exigente em termos computacionais. É por esse motivo que a função `sqrt` não foi usada nesta solução.