Funções de Mathematica 101

Maria Madrugo

Funções Simples

Começamos por definir uma função simples, que apenas duplica um número. A forma usual de fazer isto é

```
f[x_]:=2 x;
```

Esta expressão simples esconde algumas subtilezas. Para começar, os parâmetros de funções estão entre parentesis rectos. Isto verifica-se quer na definição de funções quer na avaliação:

```
g[x_,y_]:=x+y; g[1,3]
output: 4
```

Olhemos agora para o resto da sintaxe, nomeadamente para o uso de **:=**, também chamado de SetDelayed. Este comando é responsável por manter o 2 x não-avaliado, até que a função seja chamada. Isto pode à partida parecer uma distinção fraca, mas eis um exemplo que ajuda a perceber a relevância:

```
r=RandomReal[]; {r,r,r}
output: {0.964064, 0.964064, 0.964064}
```
[1]

Neste código, definimos o r como sendo um número real gerado aleatoriamente, neste caso 0.964064. Assim sendo, r tornou-se um número perfeitamente normal, e ao chamarmos r três vezes numa lista obtivémos uma lista com esse elemento três vezes.

```
r:=RandomReal[]; {r,r,r}
output: {0.964064, 0.797772, 0.472486}
```

Neste caso, ao usar o SetDelayed, a avaliação do lado direito foi adiada até o r ser chamado. Assim sendo, em vez de ser um número qualquer, o r ficou uma 'máquina de gerar números aleatórios', que de cada vez que é chamado gera um número novo.

Finalmente, notemos o _ a seguir ao x, que é muito importante: é este underscore que é responsável por informar o Mathematica que deve interpretar x como uma variável, que será substituida pelo argumento da função. Vejamos o que acontece com e sem o uso do underscore:

```
f[x_]:= 2 x;
f[x]
output: 2 x
f[y]
output: 2 y
f[2]
output: 4
```

```
f[x]= 2 x;
f[x]
output: 2 x
f[y]
output: f[y]
f[2]
output: f[2]
```

Tal como esperado, tudo correu bem no lado esquerdo. Por outro lado, no lado direito, podemos ver que só para x é que obtivémos o que queríamos, uma vez que o Mathematica não foi informado que o x é uma variável e por isso assume que estamos a definir a imagem de f especificamente para x[2].

[1] Se ainda não estás familiar com Mathematica, eis uma explicação da notação: as chavetas delimitam uma lista, cujos elementos são separados por vírgulas. Nota a falta de ambiguidade entre isso e as casas decimais de um número, que são delimitadas por um ponto final.

[2] Este comportamento pode parecer estranho, mas pode ser usado para definir funções em inputs específicos, por exemplo: not[True] := False; not[False] := True.

Variáveis Auxiliares e Module

Ao definir funções, é muitas vezes útil usar variáveis auxiliares, que são essencialmente de dois tipos:

- Variáveis Globais, que existem fora da função, podem ser chamadas dentro da função sem cuidados especiais e cujas mudanças dentro da função afectam o exterior (isto é, o restante notebook);
- Variáveis Locais, que apenas existem dentro da função e são o foco principal desta secção.

No Mathematica, a forma de trabalhar com variáveis locais é usar o comando Module. Este comando é uma função que recebe dois parâmetros:

- Uma lista com as variáveis locais, às quais podem ou não ser atribuidos valores (por exemplo, {y,z}, {y=2,z=3}, {y=2,z} e {y,z=2} são todas escolhas legítimas);
- Uma sequência de código, indicando o que queremos fazer dentro do Module, tal que
  - Os comandos estejam separados por pontos e vírgulas;
  - O último comando seja o que queremos que seja retornado pelo Module. Se for necessário retornar várias coisas basta colocar todas numa lista, e para não retornar nada basta terminar num ponto e vírgula[3].

A título de exemplo, o comando Module[{x=3,y=4,z}, x=x+2; z=x+y] retornará 9, visto que começará por adicionar 2 ao x, obtendo 5, seguido de definir o z como sendo a soma de x e y, que é 9. Como este é o último comando, será isso que o Module retorna. Nota que no resto do notebook nada acontece às variáveis x, y nem z, uma vez que são definidas dentro do Module.

Definir funções

Agora que vimos como funcionam variáveis auxiliares, estamos prontos para definir funções mais complicadas. Em geral, definimos uma função como sendo uma sequência de comandos locais, isto é, usando Module. Por exemplo, a seguinte função retorna o factorial de n:

fact[n_]:=Module[{prod=1}, Do[prod=prod j,{j,1,n}]; prod][4]

- Há um detalhe importante na definição de funções que vale a pena mencionar: não é possível modificar os argumentos dentro da função. Por exemplo, se no caso acima, não poderíamos modificar o n. Se precisássemos de o fazer, teriamos de definir outra variável no Module igual a n e modificar essa. Por exemplo, se quisermos implementar um código para a conjectura de Collatz [5]
  - f[n_]:=Module[{iter=0}, While[n!=1, If[EvenQ[n], n=n/2, n=3 n +1]; iter=iter+1]; iter] daria erro, uma vez que estamos a tentar modificar n no Module.
  - f[n_]:=Module[{iter=0, m=n}, While[m!=1, If[EvenQ[m], m=m/2, m=3 m +1]; iter=iter+1]; iter] já funcionaria perfeitamente, uma vez que estamos a modificar a variável local m, em vez do argumento n.

Funções Por Ramos e Indução/Recursão

Por vezes queremos definir funções por ramos, e o Mathematica tem um comando, Piecewise, de propósito para o efeito. No entanto, existe uma alternativa simples e com a qual o Mathemática sabe lidar surpreendentemente bem: usar If.

- A função modulo[x_]:=If[x>0, x, -x] está bem definida, e podemos por exemplo fazer Plot[modulo[x],{x,-1,1}], obtendo exactamente o que esperaríamos do módulo.

[3] Isto é um facto geral do Mathematica, pontos e virgulas servem (entre outras coisas) para suprimir o output de um dado comando.

[4] Nota que o comando Do recebe uma instrução, neste caso multiplicar prod por j, e irá efectuar a instrução para j no range pretendido, neste caso entre 1 e n. Nota também que o mathemática intrepreta espaços como produtos, daí o uso de prod j em vez de prod*j, que também estaria correcto.

[5] Dado um n, queremos retornar quantas vezes temos de fazer a operação de Collatz ($m \mapsto m/2$ se m for par ou $m \mapsto 3m+1$ se for ímpar) até chegar a 1

- Podemos também calcular a derivada fazendo modulo'[x], e obtemos If[x>0, 1, -1]. Excepto o comportamento estranho no 0 (explicado por estarmos a pedir ao Mathematica algo que não faz sentido), podemos ver que funções definidas com If são consideradas "funções a sério" pelo Mathematica.

Um uso especial deste tipo de definição de funções é quando queremos usar recursão ou indução. Por exemplo, podemos redefinir o factorial como

```
fact2[n_]:=If[n==1, 1, n fact2[n-1]]
```

Funções Puras

Até agora, vimos como definir funções às quais damos um nome. Isto tem vantagens para além de podermos referir-nos à função, relacionadas com o Mathematica ser uma linguagem funcional: funções podem ser dadas como parâmetros de funções.

- Em Mathematica, o seguinte código é perfeitamente válido (embora redundante):

```
Avaliador[f_,x_]:=f[x]
```

No entanto, por vezes pode dar jeito passar uma função como argumento sem ter de a definir antes. Existem duas formas principais de o fazer, que na verdade correspondem apenas a sintaxes diferentes para a mesma coisa:

- Usar o comando Function. Este recebe dois argumentos, o primeiro sendo uma lista[6] das variáveis e o segundo o valor da função em função dessas variáveis:
  - Para duplicar um número, podemos definir a função f=Function[x,2 x].
  - Para somar o dobro de dois números, podemos definir a função g=Function[{x,y}, 2 x+2 y].
  - Se quisermos avaliar as funções acima, podemos usar a notação habitual, nomeadamente f[5] e g[2,3], se quisermos avaliar nestes valores.
  - Se quisermos estas funções com variáveis x ou x e y, basta fazer f[x] e g[x,y], até lá "a função não sabe quais são os nomes das suas variáveis"
  - De facto, se tentares avaliar f ou g, terás como output 2 # & e 2 #1 + 2 #2 &, cujo significado ficará claro com a segunda forma de definir:
- Definir a função com # a representar variáveis e & para indicar que a função acabou. No caso de querermos mais que uma variável, podemos numerar os #, isto é, usar #1 para a primeira variável, #2 para a segunda, etc.
  - Para duplicar um número, podemos definir a função f=2 #&.
  - Para somar o dobro de dois números, podemos definir a função g= 2 #1 + 2 #2 &.
  - Tal como no caso anterior, e quisermos avaliar as funções acima, podemos usar a notação habitual, nomeadamente f[5] e g[2,3], se quisermos avaliar nestes valores.
  - Se quisermos estas funções com variáveis x ou x e y, basta fazer f[x] e g[x,y], até lá "a função não sabe quais são os nomes das suas variáveis", exactamente como no caso acima.

[6] Para funções de uma só variável não é necessário colocá-la numa lista, como podes ver no exemplo.