Um pouco de Python para matemática

Não é o objetivo deste capítulo servir como introdução ao Python. De facto, partiremos do princípio que o leitor tem familiaridade com os conceitos básicos desta linguagem: entrada e saída de dados, *loops* e condicionais e utilização de pacotes, para dizer o mínimo. É até normal que já tenha noções de manipulação de base de dados.

No entanto, na maior parte dos manuais de programação, os conceitos necessários para definir uma função (matemática, não no sentido informático...) e posteriormente manipulá-la de forma simbólica e numérica são frequentemente deixados para depois. Serão estes conceitos que, de forma muito breve, queremos mostrar aqui para que o aluno possa, com ajuda dos exemplos no final de cada capítulo e dos códigos disponibilizados na página Internet, aprender a utilizar o Python na resolução de equações diferenciais.

Vamos começar por definir uma função. Este é um momento em que o Python mostra tanto a sua riqueza como suas subtilezas.

Do ponto de vista matemático não faz grande diferença escrever "$f(x) = x^2$ para todo $x \in \mathbb{R}$" ou "$f : \mathbb{R} \to \mathbb{R}$, $f : x \mapsto x^2$ ". No entanto, estas são duas formas distintas de definir uma função

em Python.

```
def f(x):
    return(x**2)
```

```
f=lambda x:x**2
```

Para saber o valor de f num ponto específico basta escrever (em qualquer dos casos)

```
print(f(5))
```

e a resposta encontrada será 25.

Vamos agora fazer alguma manipulação simbólica. Para isto precisamos do pacote `sympy` ("symbolic Python"). O código abaixo funciona com qualquer uma das duas definições de função. Além de incluir o pacote, o que é feito com o comando `import` e dando um prefixo para as funções deste pacote (no exemplo abaixo `sym`, o que permite referenciar de forma única as funções — agora no sentido informático – que este pacote introduz), devemos dizer que `x` será um símbolo a ser manipulado.

```
import sympy as sym
x = sym.Symbol('x')
print(f(x).diff(x))
```

O programa irá responder `2*x`. Podemos, adicionalmente, definir a função derivada de f:

```
def fprime(x):
    return(f(x).diff(x))
```

```
fprime=lambda x:f(x).diff(x)
```

e, nos dois casos, a resposta a `print(fprime(x))` será `2*x`. Para saber o seu valor explícito em um ponto (digamos $x = 2$) utilizamos o comando `subs` e para avaliação numérica, utilizamos a função `evalf`:

```
8 fprime.subs({x:2})
9 fprime(x).evalf(subs={x:2})
```

obtendo como resposta `4` e `4.000000000000`.

Igualmente, podemos calcular a primitiva simbólica e a seguir a imprimir através do comando `print((lambda x:f(x).integrate(x))(x))`. Neste caso, encontramos como resposta `x**3/3`. Note que `lambda x:f(x).integrate(x)` obtém a primitiva de `f` como uma função "lambda" e a expressão `(lambda x:f(x).integrate(x))(x)` avalia esta função no ponto `x`. Para saber o seu valor em um ponto explícito, escrevemos `print((lambda x:f(x).integrate(x))(x).subs({x:2}))`.

É menos elegante, mas seria mais simples, definir uma nova função `h` como a primitiva de `f` e a seguir manipulá-la:

```
10 h=lambda x:f(x).integrate(x)
11 print(h(x))
```

É importante notar que o módulo `sympy` tem várias funções matemáticas pré-definidas, tais como `sym.sin` e `sym.exp`, para o seno e a exponencial, respetivamente.

No entanto, a grande força do Python não é no cálculo simbólico, mas sim no numérico.

Imagine uma função, por exemplo $f(x) = \operatorname{sen}(x^2)$, para a qual não existe uma primitiva explícita e vamos calcular numericamente $I = \int_0^1 \operatorname{sen}(x^2)\mathrm{d}x$.

Começamos por importar o módulo numérico `numpy` ("numeric Python") e definimos a função f que queremos integrar de forma parecida como anteriormente. Antes de explicar como o Python utiliza métodos de excelente qualidade para estimar o integral, vamos calcular de forma ingénua I, utilizando uma soma de Riemann.

Definimos $N = 1000$ e queremos calcular a aproximação de I dada por sua soma de Riemann à esquerda $\sum_{i=0}^{999} f\left(\frac{i}{N}\right)\frac{1}{N}$.

```
# Importa ‘‘numpy’’
import numpy as np
# Define f
f=lambda x:np.sin(x**2)
# Define o tamanho da partição do domínio
N=1000
# Introduz uma variável chamada ‘‘soma’’ e atribui o seu valor inicial a 0
soma=0
# Começa um loop para percorrer os pontos de interesse e
# para cada ponto da forma i/N soma o valor de f neste ponto
# multiplicada pelo tamanho do intervalo 1/N
for i in range(0,N,1):
  soma+=f(i/N)*(1/N)
# Imprime o resultado
print(soma)
```

No código acima, a expressão `range(0,N,1)` equivale a uma lista de números que começa em 0, termina em $N-1$ e salta de 1 em 1. Poderia, equivalentemente, ser escrito apenas `range(N)`. O *loop* `for` percorre todos os números desta lista, um a um. A atribuição `x+=a` atribui à variável `x` o valor corrente de `x` adicionado do valor corrente de `a`.

O resultado obtido é `0.3098476562813816`.

Apesar de não haver nada de intrinsecamente errado com este método (que converge para o valor exato quando $N \to \infty$), há formas implementadas em Python que fazem esta estimativa de forma imediata utilizando métodos muito mais sofisticados do que as somas de Riemann. Para isto utilizamos a função quad, do módulo `scipy` ("scientific Python"), particularmente o conjunto de funções do submódulo `integrate`.

```
import numpy as np
import scipy.integrate as integrate
f=lambda x:np.sin(x**2)
soma=integrate.quad(f,0,1)
print(soma)
```

Neste caso a resposta é `(0.3102683017233811, 3.444670123846428e-15)` onde a primeira entrada é o valor estimado e a segunda o erro.

O mesmo resultado pode ser obtido em apenas uma linha:

```
3 print(integrate.quad(lambda x:np.sin(x**2),0,1))
```

Poderíamos utilizar a função seno do `numpy`, do `scipy` ou do `sympy` indistintamente, escrevendo `np.sin`, `sp.sin` ou `sym.sin`, respetivamente.

Terminamos este minitutorial com uma breve explicação sobre como fazer um gráfico de uma função (mais possibilidades serão vistas no decorrer dos capítulos do livro). Para isto, será utilizado o módulo `matplotlib`, particularmente o submódulo `pyplot`.

O ponto central é que o Python (ou melhor, o `pyplot`) não faz o gráfico de uma função, mas apenas cria uma imagem de uma quantidade finita de pontos (que podem ter tamanho zero) ligados entre si por uma linha. Na prática, é o mesmo que fazer um gráfico de uma função se a malha de pontos for suficientemente densa. Criamos a malha de pontos do eixo x utilizando o comando `xdata=np.linspace(0,1,101)`. Este comando atribui à variável `xdata` uma lista de números da forma $[0, 0.01, 0.02, \ldots, 1]$, ou seja, uma lista de 101 números identicamente espaçados começando em 0 e terminando em 1. É equivalente a escrever `xdata=[i/100 for i in range(101)]`, o que não necessitaria da inclusão do `numpy`.

Os pontos no eixo y são criados a partir da aplicação da função f (definida anteriormente no código) à lista de números `xdata`, criando uma nova lista `ydata`. Note que a aplicação de f a uma lista gera uma nova lista onde o n-ésimo elemento é o resultado da aplicação de f ao n-ésimo elemento da lista original.

O gráfico será então a linha que une sucessivamente os pontos cuja abcissa é o n-ésimo ponto da lista `xdata` e a ordenada o n-ésimo ponto da lista `ydata`.

```
import numpy as np
import matplotlib.pyplot as plt
f=lambda x:np.sin(x)
xdata=np.linspace(0,1,101)
ydata=f(xdata)
plt.plot(xdata,ydata)
plt.show()
```

Vamos agora definir, e a seguir fazer o gráfico, de uma função definida por pedaços. Seja

$$f(x) = \begin{cases} 0\ , & \text{se } x < 0\ , \\ x^2\ , & \text{se } x \in [0,1]\ , \\ 1\ , & \text{se } x > 1\ . \end{cases}$$

A função f pode ser explicitamente definida em Python utilizando os condicionais `if` e `elif`. Neste caso, se quisermos traçar o gráfico da função f, temos que avaliar o valor de f na lista de pontos `xdata` e, a seguir, transformar estes valores numa lista. Para isto, utilizamos os comandos `map`, que avalia uma função em cada um dos valores presentes em uma lista, e `list`, que transforma o resultado obtido numa nova lista.

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
        if x<0:
                return(0)
        elif x>=0 and x<=1:
                return(x**2)
        elif x>1:
                return(1)

xdata=np.linspace(-1,2,201)

plt.plot(xdata,list(map(f,xdata)))
plt.show()
```

Podemos também definir a mesma função utilizando o comando `np.piecewise` e o conectivo lógico `np.logical_and`, ambos do `numpy`, junto com a operação `lambda` tal como já explicado.

```
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(-1,2,201)
plt.plot(x,np.piecewise(x,[x<0,np.logical_and(x>=0,x<=1),x>1],[0,lambda x:x**2,1]))
plt.show()
```

Com isto, creio que o leitor terá familiaridade suficiente com o Python para compreender os códigos disponibilizados no fim de cada capítulo e escrever novos códigos que correspondam às suas necessidades.