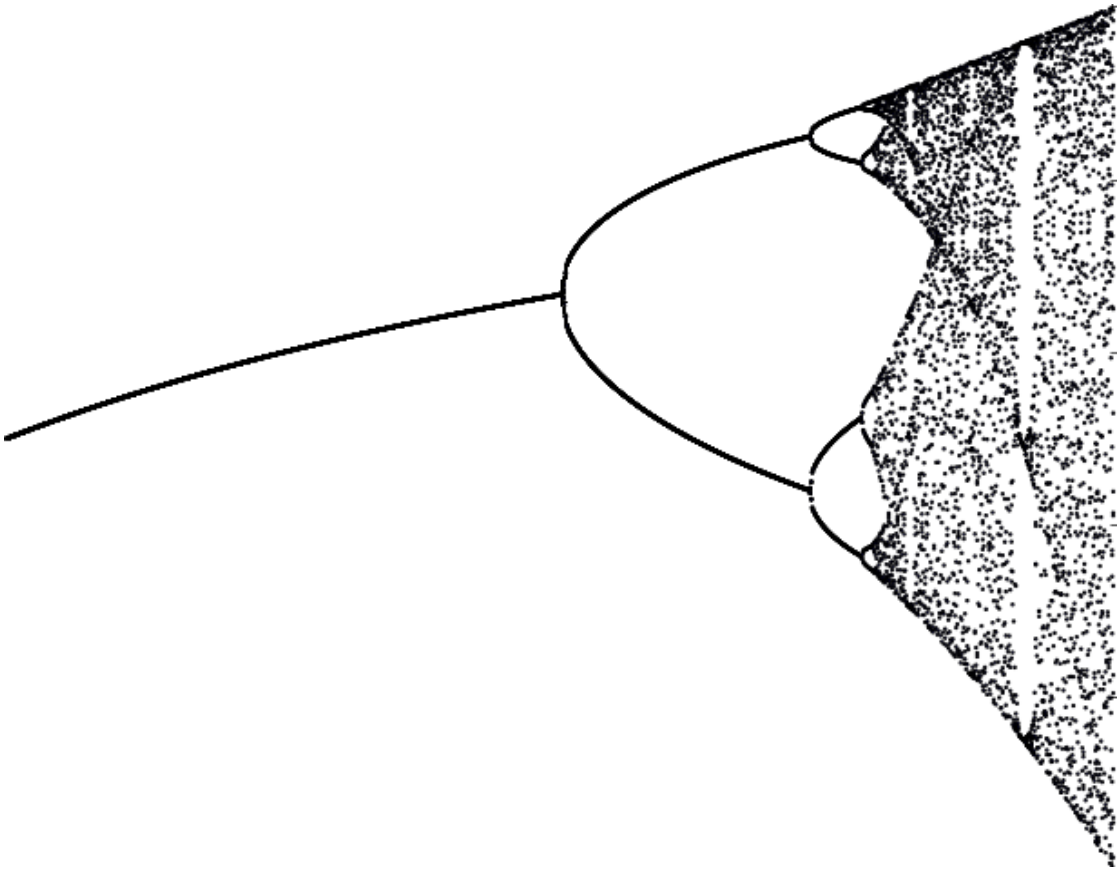
Introdução à linguagem de programação Python

Com aplicações ao cálculo científico



João Luís Silva

Setembro/2008

Conteúdo

Capítulo 1

Introdução

O Python é uma linguagem de programação de alto nível criada por Guido Van Rossum, cuja primeira versão foi publicada em 1991. O Python suporta vários paradigmas de programação, como a programação estruturada, a programação orientada a objectos e também alguns elementos da programação funcional, mas nenhuma destas escolhas é imposta ao utilizador, sendo possível utilizar o paradigma que se achar mais adequado. As principais vantagens de usar Python são:

- A qualidade do código: A sintaxe do Python facilita e encoraja a legibilidade do código, o que o torna mais fácil de manter e reutilizar.
- A produtividade do programador: É mais rápido desenvolver um mesmo código em Python do que noutras linguagens como o C, C++ ou Java. O código é mais compacto, requerendo menos linhas, e não necessita da declaração prévia do tipo das variáveis. Para além disso, não requer um ciclo de compilação, podendo o programa ser executado imediatamente.
- A portabilidade: Muitos programas Python correm sem qualquer alteração em várias plataformas como o Linux e o Windows.
- Uma vasta biblioteca padrão

A linguagem Python é referida usualmente como uma linguagem de "scripting" pois é usualmente aplicada nessas tarefas, mas não existe uma diferença de fundo entre um "script" e um programa, sendo o primeiro entendido como um programa simples. Um uso comum do Python é o de servir de linguagem de controlo de componentes escritos noutra linguagem (como o C++), no entanto o Python não está limitado a tarefas simples, podendo ser usado em programas de qualquer dimensão. Uma possível desvantagem do Python em relação a outras linguagens de programação compiladas é a velocidade de execução, o que dadas as capacidades dos computadores actuais não é um problema para a maior parte dos programas. Para os programas em que essa diferença é importante, é sempre possível desenvolver a maior parte do programa em Python suplementado por um módulo chamado a partir do Python e desenvolvido noutra linguagem.

No caso particular das simulações numéricas utilizam-se usualmente módulos externos à distribuição Python base, como o numpy e scipy.

O sítio `www.python.org` contém vasta documentação (sobretudo em inglês) acerca do Python, para todos os níveis de utilizadores. O nome da linguagem, Python, deve-se ao seu criador ser um fã da série televisiva da BBC "Monty Python Flying Circus".

1.1 Instalação

As distribuições do sistema operativo Linux já incluem o interpretador de linguagem Python, embora algumas bibliotecas que iremos utilizar não sejam instaladas por defeito. No sistema operativo Windows pode instalar a distribuição oficial, de `http://www.python.org/`, e acrescentar posteriormente os módulos em falta.

1.2 Ambientes de desenvolvimento integrados

Embora linguagem Python não imponha a utilização de ferramentas específicas para a criação de programas, é vantajoso utilizar um ambiente de desenvolvimento integrado (IDE[1]) que suporte o Python. Entre as várias escolhas de programas gratuitos de código aberto destacam-se:

- IDLE - "Integrated development for Python", que é incluído com a distribuição Python
- Eric, disponível em `http://www.die-offenbachs.de/eric/index.html`.
- Pydev, que é uma extensão ao IDE eclipse.
- SPE, de *Stani's Python Editor*, disponível em `http://pythonide.stani.be/`.

1.3 Modo interactivo

A forma mais simples de experimentar o Python é usar a linha de comandos interactiva do Python. Alternativamente, pode-se usar o ipython (`http://ipython.scipy.org/`) que tem capacidades de edição e introspecção avançadas, ou o modo imediato de um IDE Python. Usando o ipython:

[1] Abreviatura de "Integrated development environment"

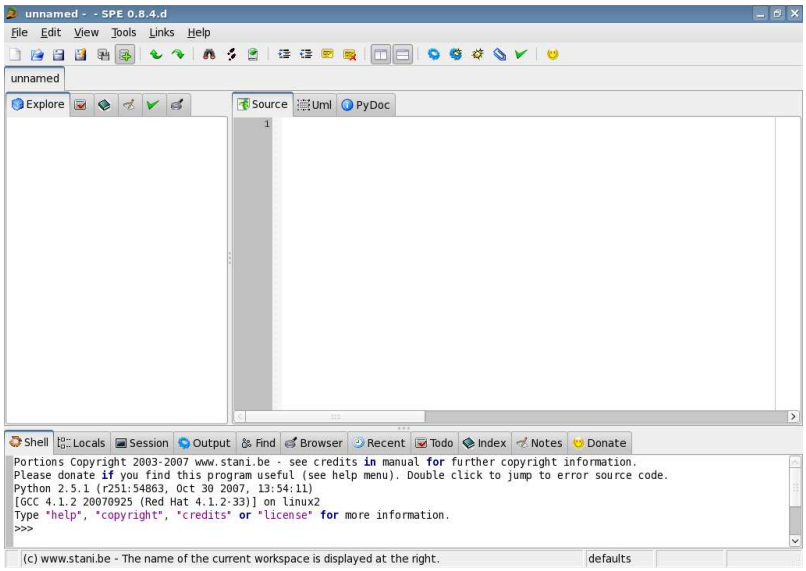


Figura 1.1: O IDE SPE

```
$ ipython
Python 2.5.1 (r251:54863, Jul 10 2008, 17:24:48) Type "copyright", "credits" or
"license" for more information.
IPython 0.8.2 – An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.
In [1]:
```

Na linha de comandos do ipython podemos introduzir qualquer comando Python. Uma das características mais úteis é o "auto-complete", isto é, se a seguir a um objecto escrevemos . e de seguida premir-mos o "tab" obtemos uma lista das possibilidades

```
In [1]: import scipy
In [2]: scipy.
Display all 526 possibilities? (y or n)
```

Outra característica importante é a habilidade de ver a ajuda relativamente a qualquer objecto:

```
In [1]: import scipy
In [2]: help(scipy)
```

e seria mostrada a documentação deste módulo. Também podemos pedir ajuda acerca de funções da própria linguagem

```
In [1]: help(open)
Help on built-in function open in module __builtin__:

open(...)
    open(name[, mode[, buffering]]) -> file object

    Open a file using the file() type, returns a file object.
```

Ou então podemos simplesmente usar o Python como uma calculadora avançada

```
In [1]: import math
In [2]: (1+math.sqrt(5))/2.0
Out[2]: 1.6180339887498949
In [3]: 2**3
Out[3]: 8
```

Quando usado interactivamente, o interpretador mostra o resultado do comando, por exemplo 2**3 imprime 8, mas isso não acontece quando está a executar um ficheiro Python, e nessa situação temos de usar a instrução **print**.

Listagem 1.1:

```
# -*- coding: latin-1 -*-
#Ficheiro print.py
#O cardinal indica que o resto da linha é um comentário
#e será ignorado pelo Python
print 2**3              #Imprime 8
```

Para executar o ficheiro print.py (listagem 1.1), numa linha de comandos:

```
$ python print.py
8
```

A primeira linha do programa (# -*- coding: latin-1 -*-) é necessária porque este contém caracteres acentuados, e é necessário especificar como estão codificados. As codificações usuais são o latin-1 e utf-8. Se esta linha não estivesse presente obteríamos o erro:

```
$ python print.py
File "print.py", line 2
SyntaxError: Non-ASCII character '\xc3' in file t.py on line 2, but no encoding
declared; see http://www.python.org/peps/pep-0263.html for details
```

Capítulo 2

Introdução à programação

Os computadores que utilizamos normalmente seguem a arquitectura de Von Neumann (ver Fig. 2.1) em que o processador executa um programa armazenado na memória, podendo ler ou escrever dados nessa mesma memória e também comunicar com os dispositivos de entrada e saída. As instruções executadas pelo processador designam-se por código máquina, sendo operações muito simples, como por exemplo ler ou escrever um valor de uma dada posição da memória, somar dois números ou mudar o ponto de execução do programa para uma outra posição se o resultado da operação anterior for zero. Cada uma destas operações elementares é identificada por um número, e quando aplicável, seguem-se os números referentes aos parâmetros passados a essa instrução. Como é difícil programar usando apenas números criou-se uma representação intermédia chama *assembly*, em que cada uma destas instruções é agora representada por uma mnemónica, e passa a ser necessário converter desta representação mais útil para nós para os números que o processador consegue executar usando um programa chamado *assembler*. Apesar de o *assembly* ser mais fácil de utilizar, é ainda necessário estar envolvido nos detalhes da máquina, sendo complexo escrever programas extensos, pelo que hoje em dia é apenas utilizado em pequenas rotinas que necessitem de um contacto mais próximo com o *hardware*.

Posteriormente foram desenvolvidas outras linguagens de programação que, por comparação com o *assembly,* são linguagens de alto nível, como o Fortran, Pascal ou C, que utilizam abstracções mais próximas das usadas pelo homem, mas que tal como a primeira têm de poder ser traduzidas de forma não ambígua para código máquina. A maior parte dos programas disponíveis hoje em dia são criados em linguagens deste tipo.

Neste curso vamos estudar a linguagem Python, que é chamada de muito alto nível, pois utiliza abstracções ainda mais afastadas das utilizadas pelo processador, e que nos são mais próximas.

Figura 2.1: A arquitectura de Von Neumann

2.1 Tipos de dados

2.1.1 Valores numéricos

O Python suporta números inteiros, reais e complexos, que podemos manipular com as operações aritméticas convencionais (ver Tab. 2.1). Por exemplo

Listagem 2.1:

```
n = 3*5**2      # Variável inteira
x = 3.14/2.0    # Variável real
z = 1j          # Variável complexa
```

Neste exemplo $n = 3 \cdot 5^2 = 75$ e à variável z é atribuído o valor da unidade imaginária, que em Python é representada por j, mas que tem sempre que se seguir a um número para se distinguir da variável j. As operações que envolvam números inteiros e reais retornam um resultado real e as que envolvam dois inteiros devolvem um inteiro. Uma consequência importante desta última propriedade é que o mesmo operador divisão efectua a divisão inteira para dois valores inteiros, mas a divisão real quando um dos valores é real.

Listagem 2.2:

```
print 1/3          #Imprime 0
print 1/3.0        #Imprime 0.333333333333
print 1.0/3.0      #Imprime 0.333333333333
```

Este comportamento vai mudar na versão 3.0 do Python, sendo então o operador / sempre a divisão real, e utilizando // para a divisão inteira. Podemos forçar esse novo comportamento já na versão actual do Python se a primeira linha do nosso programa for

Listagem 2.3:

```
from __future__ import division
```

2.1.2 Cadeias de caracteres

As cadeias de caracteres são delimitadas em Python por ‘ ou ". Por exemplo

Listagem 2.4:

```
s = "abcdef"
mensagem = 'Olá, Mundo!'
```

Operador	Descrição
+	Soma
-	Subtracção
*	Multiplicação
/	Divisão (real ou inteira)
//	Divisão inteira
**	Elevar a uma potência
%	Módulo (resto da divisão inteira)

Tabela 2.1: Operadores aritméticos

Estas cadeias de caracteres não podem ter mudanças de linha. Se for necessário incluir mudanças de linha pode-se ou colocar explicitamente o código de mudança de linha \n ou usar aspas triplas """.

Listagem 2.5:

```
s2 = "Linha 1\nLinha2"
mensagem_longa = """Ao iniciarmos uma cadeia de caracteres
com aspas triplas podemos usar mudanças de linha sem que
isso constitua um erro de sintaxe"""
```

Podemos aceder a caracteres individuais ou secções de uma cadeia de caracteres usando a notação de parêntesis rectos e sequências:

Listagem 2.6:

```
s = "abcdef"
print len(s) #A função len(s) devolve o tamanho de s,
             # neste caso 6
print s[0]   #Imprime o primeiro caracter de s, 'a'
print s[-1]  #Índices negativos contam a partir do fim,
             #neste caso 'f'

print s[2:4] #Sequência de índices, o primeiro é 2 e
             #menor que 4 (2 e 3), neste caso imprime 'cd'
```

A sintaxe geral de uma sequência é inicio:fim:incremento, em que cada um destes valores é opcional.

Listagem 2.7:

```
s = "abcdef"
print s[::2]  #Todos os elementos, incremento 2,
              #imprime 'ace'

print s[::-1] #Todos os elementos em ordem inversa,
              #imprime 'fedcba'

print s[3:]   #Início no índice 3, imprime 'def'
```

Método	Descrição
insert(i,v)	Insere o valor v na posição i
append(v)	Acrescenta o valor v ao fim da lista
remove(v)	Remove da lista o primeiro item com valor v
pop()	Retorna o último valor da lista, removendo-o.
index(v)	Retorna o índice da primeira ocorrência de v. Se não existir retorna um erro.
count(v)	Número de vezes que v está na lista
sort()	Ordena a lista
reverse()	Inverte a ordem dos elementos da lista

Tabela 2.2: Métodos das listas

```
print s[4:2]   #Início no índice 4, termina no 2,
               #como não corresponde a nenhum índice
               #possível, retorna uma cadeia de
               #caracteres vazia, ''
```

Em Python as cadeias de caracteres são imutáveis e embora possamos aceder a caracteres individuais não os podemos alterar:

```
In [1]: s = "abc"
In [2]: s[0] = "z"
TypeError: 'str' object does not support item assignment
```

Podemos no entanto atribuir à mesma variável um novo valor construído com base no anterior:

```
In [1]: s = "abc"
In [2]: s = "z" + s[1:]
In [3]: print s
zbc
```

2.1.3 Listas

Uma lista é um conjunto ordenado de objectos. Uma lista pode ser alterada, inserindo, apagando ou substituindo um ou mais elementos. Em Python as listas são delimitadas por parêntesis rectos.

Listagem 2.8:

```
l = [3.14,"pi",7]
print l[0]          #Imprime 3.14
l[2] += 3           # Agora l é [3.14,"pi",10]

l.append(1j)        #Insere o número complexo 1j
                    #no fim da lista
```

As listas, tal como as cadeias de caracteres e tuplos suportam indexação usando sequências.

Listagem 2.9:

```
l = [1,"a",3.14,2.718]
print l[:2]      #Imprime [1, 'a']
```

2.1.4 Tuplos

Os tuplos são sequências ordenadas de objectos, semelhantes às listas, mas imutáveis. Os tuplos são delimitados por parêntesis.

Listagem 2.10:

```
t = (1,"abc",5.0,7.4)
print t[1:3]     #Imprime ('abc', 5.0)
t2 = (10,)       #Um tuplo com apenas um elemento
```

Para criar um tuplo com apenas um elemento é necessário por uma vírgula a seguir esse elemento, de outra forma seria apenas um valor entre parêntesis. Os próprios parêntesis são opcionais onde tal não é ambíguo:

Listagem 2.11:

```
a = 10
b = 20
a,b = b,a  #Operação entre tuplos, apesar de não ter
           #parentesis. Agora a=20 e b=10
```

2.1.5 Dicionários

Um dicionário associa valores a chaves. As chaves têm de ser valores imutáveis (números, cadeias de caracteres ou tuplos destes), sendo associadas um valor que pode ser qualquer objecto Python.

Listagem 2.12:

```
d = {}              #Um dicionário vazio
d[1] = "abc"        #Associa o valor "abc" à chave 1
```

Podemos ver quais são as chaves de um dicionário com o método keys()

Listagem 2.13:

```
d = {1:"um",2:"dois"}
print d.keys()  #Imprime [1,2]
if 2 in d:
    print "O dicionário tem a chave 2, com o valor",d[2]
```

Capítulo 3

Instruções de controlo do fluxo do programa

O fluxo de execução de um programa que normalmente segue de uma linha para a seguinte pode ser alterado, executando um dado bloco apenas se uma condição for verdadeira, ou executando repetidamente blocos de código. Estas operações são realizadas com as instruções **if**, **for** e **while**. Antes de estudar estas instruções necessitamos primeiro definir o que é uma condição.

3.1 Condições

Em Python qualquer objecto pode ser testado para determinar se é uma condição verdadeira ou falsa. São condições falsas **None**, **False**, um valor numérico nulo, uma lista, cadeia de caracteres, tuplo, ou dicionário vazios. Os restantes são verdadeiros. Estas condições podem ser combinadas com os operadores lógicos **or** (ou lógico), **and** (e lógico) e **not** (não lógico). O exemplo seguinte vai imprimir a mensagem, pois uma lista vazia é uma condição falsa, e negação de uma proposição falsa é uma proposição verdadeira.

Listagem 3.1:

```
lista = []
if not lista:
    print "A lista está vazia"
```

3.2 Blocos de código

As instruções **if**, **while**, **for** e **def** são seguidas por um bloco de código. Em Python, um bloco de código é um conjunto de uma ou mais linhas de código que estão espaçadas da mesma forma a partir da margem esquerda. Isto é, em Python o alinhamento do código é significativo. Por exemplo o seguinte programa vai falhar com um erro:

Operador	Descrição
<	Menor
>	Maior
<=	Menor ou igual
>=	Maior ou igual
==	Igual
!=	Diferente

Tabela 3.1: Operadores lógicos

Listagem 3.2:

```
x = 5
if x<10:
    x += 1
      print "x foi incrementado"
```

```
File "bloco.py", line 4
    print "x foi incrementado"
    ^ IndentationError: unexpected indent
```

A instrução **print** deveria estar alinhada com a anterior. Como neste ponto o Python não espera um novo bloco falha com o erro "IndentationError". É importante espaçar da mesma forma os blocos de código, e a convenção recomendada é usar 4 espaços. Isto é tornado mais fácil usando um editor que suporte as convenções Python, o que assegura a uniformidade do código. Se não estiver a usar um editor que suporte estas convenções é importante usar sempre 4 espaços e não misturar espaços com tabulações.

Como um bloco de código tem de ter sempre pelo menos uma linha existe uma instrução que não faz nada, **pass**. Desta forma, se por exemplo quisermos deixar temporariamente um dado bloco vazio esta instrução satisfaz o requisito de lá ter pelo menos uma linha.

3.3 Decisões: if

A instrução **if** permite-nos executar um bloco de código se se verificar uma dada condição.

Listagem 3.3:

```
if a>=b and a>=c:
    print "a é valor máximo"
elif b>=a and b>=c:
    print "b é valor máximo"
else:
    print "c é valor máximo"
```

Este exemplo ilustra a sintaxe: Se a primeira condição da primeira linha for verdadeira é executado o primeiro bloco de código. Uma instrução **if** pode ser

opcionalmente seguida por uma ou mais instruções **elif**, abreviatura de "else if", isto é, se a condição do **if** inicial não for verdadeira então executar o bloco seguinte se esta nova condição se verificar. No final, pode existir opcionalmente uma clausula **else**, cujo bloco é executado se nenhuma das outras condições tiver sido verificada.

Listagem 3.4:

```
if x>=10:
    print "x é maior ou igual a 10"
else:
    print "x é menor que 10"
```

3.4 Executar um bloco de código um dado número de vezes: for

Quando queremos que um dado bloco de código seja executado um número fixo de vezes (podendo esse número de vezes ser dado por uma variável) utilizamos a instrução **for**.

Listagem 3.5:

```
print "A tabuada do 9 é:"
for i in range(1,11):
    print "9*",i,"=",9*i
```

A sintaxe é "**for** variável **in** expressão:", em que expressão deve ser iterável.

3.4.1 A função range

A função **range** devolve uma lista com uma sequência de inteiros. O valor inicial é por defeito zero, o valor de paragem tem sempre de ser especificado, e por defeito o valor do incremento é 1.

Listagem 3.6:

```
range(5)        #Devolve [0,1,2,3,4]. O valor inicial é
                #zero, o incremento 1.

range(2,4)      #Devolve [2,3].
range(2,15,3)   #Devolve [2, 5, 8, 11, 14].
                #O valor do incremento é 3.

range(5,0,-1)   #Devolve [5, 4, 3, 2, 1]. O valor do
                #incremento pode ser negativo.

range(0,5,-1)   #Devolve [] (uma lista vazia).
```

3.5 Executar um bloco de código enquanto uma condição for verdadeira: while

A instrução **while** permite-nos repetir um bloco de código enquanto uma condição for verdadeira. Pode opcionalmente ser seguida de uma clausula **else** que é executada se o bloco dentro da instrução **while** não tiver saído devido à instrução **break**.

Listagem 3.7:

```
x = 10
while x>0:
    if x % 2 != 0: #Se x é impar
        print x,
    x -= 1          #Equivalente a x = x - 1
```

O programa anterior imprime 9 7 5 3 1.

3.6 As instruções break e continue

As instruções **break** e **continue** podem ser usadas dentro de ciclos **for** e **while**. A instrução **break** causa uma saída imediata do ciclo, o **continue** interrompe o processamento do resto do bloco de código do ciclo, passando de imediato à próxima iteração.

Capítulo 4

Estruturação e organização do código

Neste momento já vimos as instruções principais que constituem um programa, mas se quisesse-mos aplicar as mesmas operações em várias partes do programa teríamos de as escrever de novo. Ora, não é de todo isso que se deve fazer. Vamos primeiro ver a estrutura mais simples, as funções.

4.1 Funções

Uma função é um bloco de código ao qual são opcionalmente passados parâmetros e pode retornar um único valor. Por exemplo podemos re-escrever o exemplo anterior da tabuada como sendo

Listagem 4.1:

```
def tabuada(n):
    print "A tabuada do",n,"é:"
    for i in range(1,11):
        print n,"*",i,"=",n*i

tabuada(9) #Chama a função tabuada que
           #acabamos de definir para n=9
```

Uma função é definida pela instrução **def**, que é seguida pelo nome da função e por parêntesis, que contêm a lista de parâmetros. Opcionalmente uma função pode usar a instrução **return**, que termina a função e retorna ao chamador. A instrução **return** pode ser seguida de um valor, sendo esse o valor da chamada da função.

Listagem 4.2:

```
def soma(a,b):
    return a+b

c = soma(2,3)
print soma("abc","def")    # Imprime abcdef
```

Neste exemplo definimos uma função que retorna a soma de dois números, sendo o resultado atribuído à variável c. Mesmo que uma função retorne um valor, não é obrigatório atribuir esse valor a uma variável, podendo este ser ignorado se assim o desejarmos. A função soma não tem de ser usada apenas com números, mas pode receber qualquer tipo de dados que possam ser somados com o operador "+" , como cadeias de caracteres.

À primeira linha de definição de uma função pode opcionalmente seguir-se uma cadeia de caracteres. Nesse caso essa cadeia de caracteres é a cadeia de documentação da função ("docstring"). Essa cadeia de caracteres é então associada à própria função, e pode ser consultada usando os mecanismos de introspecção do Python ou por programas (como IDEs) que as reconheçam.

Listagem 4.3:

```
def soma(a,b):
    """A função soma devolve a soma dos seus pâraметros"""
    return a+b
```

4.1.1 Funções base do Python

O Python têm algumas funções pré-definidas, que fazem parte da linguagem. Estas incluem:

- abs(x) - Retorna o valor absoluto de x
- all(iteravel) - Retorna verdade se todos os elementos da expressão iterável são verdadeiros.
- any(iteravel) - Retorna verdade se algum dos elementos da expressão iterável for verdadeiro.
- bool(x) - Converte x para booleano, isto é, retorna **True** ou **False** de acordo com as regras expostas na Sec. 3.1.
- chr(x) - Retorna o caracter com código ASCII x.
- cmp(x,y) - Retorna um valor inteiro que é negativo se x<y, zero se x=y e positivo se x>y.
- complex(s) - Converte a cadeia de caracteres s para um número complexo.
- complex(re,im) - Retorna o número complexo re + 1j*im onde re e im são números reais.
- dir(objecto) - O parâmetro objecto é opcional. Se for passado retorna uma lista dos atributos válidos do objecto (como funções e variáveis), caso contrário retorna o conteúdo da tabela de símbolos local.
- divmod(a,b) - Sendo a e b números retorna um par de números que são o resultado da divisão inteira e o resto da divisão inteira de a por b.

- enumerate(iteravel) - Retorna um objecto iterável que vai devolver tuplos com dois elementos, sendo o primeiro um valor que começa em zero e vai sendo incrementado, o segundo é o valor do iteravel que lhe foi passado como argumento.

- eval(expressao) - Avalia a expressão contida na cadeira de caracteres expressao. Por exemplo eval("2+3") retorna 5.

- filter(f,iteravel) - Retorna uma lista constituída por todos os elementos de iteravel para os quais a função f retornou **True**.

- float(x) - Converte para número real o número ou cadeia de caracteres x.

- globals() - Retorna um dicionário contendo a tabela de símbolos global.

- help() - Activa o sistema interactivo de ajuda. Se for passado um parâmetro, mostra ajuda acerca desse objecto.

- hex(n) - Retorna uma cadeia de caracteres com a representação hexadecimal do número inteiro n.

- int(x) - Converte para inteiro o número ou cadeia de caracteres x.

- len(s) - Retorna o tamanho da cadeia de caracteres, tuplo, lista ou dicionário s.

- list(iteravel) - Retorna uma lista com todos os elementos de iteravel.

- locals() - Retorna um dicionário contendo a tabela de símbolos local.

- map(f,iteravel1,...) - Recebe uma função f e um ou mais iteráveis. A função f tem de receber tantos parâmetros quantos iteráveis forem passados. Retorna uma lista constituída pelo resultado de f aplicada aos elementos dos iteráveis.

- max(iteravel) - Retorna o valor máximo de entre os elementos de iteravel.

- max(valor1,valor2,...) - Quando a função max() é chamada com mais do que um parâmetro, devolve o máximo desses parâmetros.

- min() - Semelhante à função max(), mas retorna o valor mínimo.

- open(ficheiro,[modo]) - Abre um ficheiro. Para mais detalhes ver o Cap. 5.

- ord(c) - Retorna o código ASCII ou código Unicode do caracter c.

- pow(x,y) - Retorna x elevado a y. Equivalente a x**y.

- range() - Cria uma lista contendo uma sequência. Para mais detalhes ver a Sec. 3.4.1.

- raw_input(prompt) - Imprime prompt e devolve uma cadeia de caracteres que leu da entrada padrão, que é normalmente o teclado. Esta função é útil para pedir valores ao utilizador.
- reduce(f,iteravel) - Aplica a função de dois parâmetros f cumulativamente aos valores de iterável, de forma a reduzir iteravel a um único valor. Por exemplo, usando a função soma da Listagem 4.3, reduce(soma,[1,2,3,4]) retorna 10, pois primeiro chama a função soma com 1 e 2, depois com 3 e 3, e finalmente com 6 e 4.
- reload(modulo) - Volta a carregar o módulo especificado. Ver a Sec. 4.2 para mais detalhes.
- repr(x) - Retorna uma cadeia de caracteres com uma representação do objecto x, que é o mesmo que seria impresso com "**print** x".
- reversed(seq) - Retorna um iterável constituído pelos elementos de seq pela ordem inversa.
- round(x) - Arrendonda o valor real x. Opcionalmente pode receber um segundo parâmetro inteiro que indica o número de casas decimais a manter.
- sorted(iteravel) - Retorna uma lista ordenada constituída pelos valores de iteravel.
- str(x) - Converte o objecto x para cadeia de caracteres.
- sum(iteravel) - Retorna a soma dos valores de iterável.
- tuple(iteravel) - Retorna um tuplo constituído pelos elementos de iteravel, mantendo a ordem destes.
- unichr(n) - Retorna o caracter Unicode correspondente ao código n.
- xrange() - Semelhante a range, mas retorna um iterável e não uma lista, e está optimizado para a velocidade e menor consumo de memória do que o range(). A diferença entre o xrange() e o range() só é importante para sequências com muitos elementos.
- zip(iteravel1, iteravel2, ...) - Retorna uma lista de tuplos, em que o n-ésimo elemento da lista consiste num tuplo dos n-ésimos valores dos iteráveis.

4.2 Módulos

Um módulo é simplesmente um ficheiro com extensão .py que contem código Python. É importante manter conjuntos lógicos de funções em módulos diferentes, não só por uma questão de organização, mas por ficarem em espaços de nomes ("namespaces") diferentes. Isto é, uma variável global pertence ao módulo de onde foi definida, não interferindo com outros módulos diferentes. Em programas extensos é essencial compartimentar as várias partes, de forma a limitar a sua complexidade.

Podemos num dado módulo aceder à funcionalidade de outro com a instrução **import**:

Listagem 4.4:

```
import sys  #Importa o módulo da biblioteca padrão sys
from math import sin,sqrt #Importa as funções sin e
                          #sqrt do módulo math
import numpy as np #Importa o módulo numpy, mas
                   #passamos a referi-lo como np
print sys.platform
print sqrt(25.0)
x = np.arange(0.0,10.0,0.1) #Função arange do módulo numpy
```

Ao usar a forma "**import** modulo1", para usar as variáveis e métodos definidos no módulo escrevemos modulo1.metodo ou modulo1.variavel. Na forma "**from** modulo1 **import** func1,func2" ou "**from** modulo1 **import** *" os métodos e variáveis do modulo1 ficam disponíveis no mesmo espaço de nomes que já estamos. Se importar-mos desta forma uma função ou variável com o mesmo nome de uma outra já existente só podemos aceder à última que foi importada, perdendo o acesso à que já existia. A instrução **import** tem várias características importantes:

- Primeiro o módulo tem de ser encontrado. Este é procurado primeiro na directoria actual do programa. Se não for encontrado é de seguida procurado nas directorias especificadas na variável de sistema PYTHONPATH, e não tendo sido encontrado é procurado nas directorias que contêm a biblioteca padrão do Python.
- Se o módulo for encontrado, é verificado se existe uma versão compilada ("bytecode") do código actual (ficheiros .pyc). Se tal não se verificar, o módulo é então compilado, e o "bytecode" gerado é guardado num ficheiro com extensão .pyc.
- O módulo é então executado, do início até ao fim, o que define as funções e variáveis desse módulo. Se houver o módulo possuir instruções fora de funções ou classes essas são também executadas.
- Esta sequência só ocorre na primeira que importarmos um módulo num programa. Se o mesmo módulo for importado outra vez apenas é utilizada a cópia que já está em memória. Para forçar a execução de todos estes passos podemos usar a função *reload*(), que volta a carregar o módulo especificado.

4.3 Classes

A programação orientada a objectos (OOP) é suportada em Python, sendo uma classe criada com a instrução **class**. Apesar de a OOP ser opcional em Python, é importante estar familiarizado com este tipo de abordagem.

Na sua forma mais simples uma classe é apenas uma forma diferente de agrupar dados com o código que manipula esses dados:

Listagem 4.5:

```
# -*- coding: latin1 -*-
class Temperatura:
    """Conversões de temperatura"""
    def __init__(self,temp_Celsius=0.0):
        self.graus_Celsius = temp_Celsius
    def C2F(self,c):
        """Converte de graus Celsius para Fahrenheit"""
        return c * (9.0/5.0) + 32.0
    def F2C(self,f):
        """Converte de graus Fahrenheit para Celsius"""
        return (f-32)*(5.0/9.0)
    def Fahrenheit(self):
        return self.C2F(self.graus_Celsius)
    def Celsius(self):
        return self.graus_Celsius
    def Kelvin(self):
        return self.graus_Celsius + 273.15

#Cria uma instância da classe Temperatura
temp1 = Temperatura(20.0) # 20º C
print temp1.Celsius(),"ºC correspondem a", \
      temp1.Fahrenheit(),"ºF e a",temp1.Kelvin(),"K"
```

Neste exemplo a classe Temperatura contem várias funções, que na OOP são chamadas de métodos. Os métodos de uma classe em Python têm obrigatoriamente pelo menos um parâmetro, a que por convenção se chama self que recebe a instância da classe que o chamou. Isto é necessário pois uma mesma classe pode ter muitas instâncias, cada uma com os seus valores, não existindo assim ambiguidade entre as variáveis da classe ou de uma instância particular. Uma classe pode ter um método __init__ que é chamado automaticamente a quando da instanciação da classe. Se este método tiver mais parâmetros para além do self estes são recebidos dos argumentos que foram passados na criação da instância, neste exemplo 20.0.

Uma descrição pormenorizada de classes e OOP está fora do âmbito deste texto. Para mais detalhes consulte a documentação indicada no Cap. 9.

Capítulo 5

Operações de entrada / saída

Para abrir um ficheiro, seja para leitura ou escrita usa-se a função **open**(nome_ficheiro,modo) onde o primeiro parâmetro é o nome do ficheiro, e o segundo o modo, "r" para leitura, "w" para escrita (se o ficheiro já existir é apagado), "a" para acrescentar informação a um ficheiro existente.

Listagem 5.1:

```
f = open("teste.dat","w") #Abre o ficheiro teste.dat
                          #para escrita
f.write("teste\n")
f.close()                 #Fecha o ficheiro
```

Um ficheiro é um objecto iterável, e podemos ler todas a linhas com um ciclo **for**. O próximo exemplo escreve para o ecrã o conteúdo do ficheiro t.dat:

Listagem 5.2:

```
f = open("t.dat","r")
for linha in f:
        print linha,

f.close()
```

O exemplo seguinte tabela a função seno, escrevendo os resultados no ficheiro seno.dat, em duas colunas separadas por um "tab":

Listagem 5.3:

```
import numpy as np

x = np.arange(0.0,10.0,0.05)
y = np.sin(x)

f = open("seno.dat","w")
n = len(x)
for i in range(n):
```

```
    f.write("%g\t%g\n" % (x[i],y[i]))

f.close()
```

A sintaxe "formato" % tuplo tem a mesma funcionalidade do printf da linguagem C. O código \t é o "tab" e %g é um formato para imprimir um número real.

Capítulo 6

Controlo de erros

Durante a execução de um programa podem surgir erros, como por exemplo uma divisão por zero. Estes erros são chamados de excepções. Se estas excepções não forem interceptadas são reportados pelo Python e terminam a execução do nosso programa:

Listagem 6.1:

```
print 1/0
```

```
$ python erro.py
  File "erro1.py", line 1, in <module>
    print 1/0
ZeroDivisionError: integer division or modulo by zero
```

Neste caso é despoletada a excepção ZeroDivisionError.

Listagem 6.2:

```
# -*- coding: latin-1 -*-

def dividir(a,b):
    try:
        return a/b
    except ZeroDivisionError:
        return "Divisão por zero!"

print dividir(1,0)
```

Neste exemplo uma instrução que pode falhar está dentro uma instrução **try** / **except**, e ao verificar-se a excepção ZeroDivisionError em vez de o programa falhar, a função dividir retorna uma mensagem.

Capítulo 7

A biblioteca padrão

O Python é uma linguagem "com as pilhas incluídas" no sentido em que inclui uma extensa lista de módulos. Vamos apenas apresentar aqui uma pequena parte dos módulos disponíveis, para ver a lista completa consultar a "Python Library Reference" em `http://docs.python.org/lib/lib.html`.

7.1 Os módulos math e cmath

O módulo **math** disponibiliza as operações matemáticas base, como as funções trigonométricas usuais (*sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *atan*2), funções trigonométricas hiperbólicas (*sinh*, *cosh*, *tanh*), raiz quadrada (*sqrt*), exponenciação (*exp*, *pow*), logaritmos (*log*, *log*10), bem como as constantes *e* e *pi*.

O módulo **cmath** é semelhante ao **math**, mas enquanto que o **math** apenas suporta números reais, o **cmath** suporta números imaginários, pelo que

Listagem 7.1:

```
import math
import cmath

print math.sqrt(-1.0)   #Erro ValueError, pois não existe
                        #a raiz quadrada real de -1

print cmath.sqrt(-1.0) #Imprime 1j
```

7.2 O módulo sys

A variável **sys.argv** contem uma lista com os parâmetros passados ao programa Python, sendo o primeiro parâmetro o nome do próprio script. As variáveis **sys.stdin**, **sys.stdout** e **sys.stderr** são objectos Python de tipo ficheiro que estão ligados à entrada padrão, saída padrão e saída de erros padrão respectivamente. A função **sys.exit()** permite a saída imediata do programa.

7.3 O módulo os

O módulo **os** fornece acesso a funções subjacentes do sistema operativo de uma forma portável.

7.4 O módulo csv

Este módulo assiste na leitura de ficheiro de dados delimitados por um separador. O exemplo seguinte mostra como ler o ficheiro "file.dat" que consiste em duas colunas de valores reais separados por uma tabulação, armazenando cada coluna numa lista.

Listagem 7.2:

```
import csv
f = open("file.dat","r")
x = []
y = []
reader = csv.reader(f,delimiter="\t")
for r in reader:
    x.append(float(r[0]))
    y.append(float(r[1]))
f.close()
```

Capítulo 8

Python e cálculo numérico

Os tipos de dados incluídos na linguagem Python são muito flexíveis, mas podem tornam-se lentos e ineficientes para manipular grandes quantidades de informação. Por exemplo, um vector de números reais é sempre constituído por elementos do mesmo tipo mas se os armazenássemos numa lista essa lista teria também de suportar todas as operações de qualquer outra lista potencialmente heterógenea. Para responder a estas dificuldades existem módulos especializados para cálculo numérico como o numpy e scipy. Estes módulos não fazem parte da distribuição oficial do Python, embora estejam normalmente disponíveis nas distribuições de Linux, e na distribuição de Python para Windows da Enthought, podendo também ser obtidos a partir de `http://www.scipy.org/`.

8.1 Numpy

O numpy é uma extensão do Python que adiciona suporte para vectores e matrizes multi-dimensionais, bem como uma série de funções que operam nesses objectos. O numpy está documentado exaustivamente num livro gratuito, disponível em `http://www.tramy.us/guidetoscipy.html`.

O objecto base do numpy é o array. Um array é um conjunto de elementos, todos do mesmo tipo, com um dado número de dimensões.

Listagem 8.1:

```
import numpy as np
a = np.array([1,2,3]) #Converte uma lista para array

#Array 1D com os valores de 0 a 10, passo 0.1
#armazenados como reais de precisão dupla
b = np.arange(0.0,10.0,0.1,dtype=np.float64)
```

Um array tem vários atributos básicos. Por exemplo para um array a:

- a.ndim: Número de dimensões
- a.shape: Tuplo com o número de elementos de cada dimensão

Tipo de dados	Descrição
bool	Booleano (Verdadeiro / Falso)
uint8	Inteiro sem sinal, 8 bits (1 byte).
int32	Inteiro 32 bits
int64	Inteiro 64 bits
float32	Real de precisão simples
float64	Real de precisão dupla
complex64	Complexo de precisão simples
complex128	Complexo de precisão dupla

Tabela 8.1: Tipos de dados mais comuns do numpy

- a.size: Número total de elementos do array
- a.dtype: Tipo de dados. Ver a tabela 8.1 para uma lista de tipos de dados.

Podemos criar um array inicializado a zero com a função zeros:

Listagem 8.2:

```
import numpy as np
#Array de reais de precisão dupla
#com 3 linhas e 4 colunas, inicializado a zero
x = np.zeros((3,4),dtype=np.float64)

#Podemos mudar a forma do array,
#neste caso para 6 linhas e 2 colunas.
x.reshape((6,2))
```

O numpy suporta um grande número de operações sobre os arrays, que são aplicadas elemento a elemento. Podemos utilizar as operações aritméticas usuais (+, - , *, /, **), funções trignométricas (sin, cos, tan, sinh, cosh, tanh, etc.)., entre outras (abs, conj, power, etc.) (ver tabela 8.2).

Para além destas funções o numpy suporta funções lógicas, que retornam um array de valores booleanos com o resultado da respectiva operação. Estas funções são greater, greater_equal, less, less_equal, not_equal, equal, logical_and, logical_or, logical_xor e logical_not.

Listagem 8.3:

```
import numpy as np
x = np.array([10,2,3,5])
y = np.array([2,4,15,2])
print np.greater(x,y) #Imprime [ True False False  True]
```

Pode encontrar uma lista de outras funções do numpy na tabela 8.3.

8.1.1 Diferenças finitas

O numpy suporta nos arrays as mesmas operações de sequências que as listas python. Usando esta funcionalidade podemos implementar algoritmos de dife-

Função	Descrição
abs	Valor absoluto
conj	Complexo conjugado
exp(x)	e^x
log	Logaritmo natural
log10	Logaritmo base 10
sqrt	Raiz quadrada
square(x)	x^2
reciprocal(x)	$1/x$
sin, cos, tan	Funções trignométricas
arcsin, arccos, arctan	Funções trignométricas inversas
arctan2(x1,x2)	arctan(x1/x2) considerando os sinais de x1 e x2
hypot(x1,x2)	$\sqrt{x_1^2 + x_2^2}$
sinh,cosh, tanh	Funções trignométricas hiperbólicas
arcsinh, arccosh, arctanh	Funções trignométricas hiperbólicas inversas
maximum(x1,x2)	Elemento máximo
minimum(x1,x2)	Elemento mínimo
real, imag	Parte real ou imaginária

Tabela 8.2: Algumas funções do numpy

renças finitas de forma eficiente. O exemplo seguinte resolve a equação do calor 1D

$$\frac{dU}{dt} = k\frac{d^2U}{dx^2}$$

usando o algoritmo de diferenças finitas FTCS ("Forward Time Centered Space"). Este método não o ideal para resolver a equação do calor devido às suas limitações em termos de critérios de convergência, mas é uma forma simples de ilustrar o uso de diferenças finitas. Variando o índice j sobre a coordenada t, e o índice i sobre a coordenada x, temos

$$\frac{U_i^{j+1} - U_i^j}{\Delta t} = k\frac{U_{i-1}^j - 2U_{i-1}^j + U_{i+1}^j}{\Delta x^2},$$

ou seja,

$$U_i^{j+1} = U_i^j + \frac{k\Delta t}{\Delta x^2}\left(U_{i-1}^j - 2U_{i-1}^j + U_{i+1}^j\right).$$

Listagem 8.4:

```
# -*- coding: latin-1 -*-
#Equação do calor, dU/dt = k d^2U/dx^2
import numpy as np
import pylab

dt = 0.05
dx = 0.5
```

Função	Descrição
all(x)	Retorna verdade se todos os elementos de x forem verdadeiros
angle(z)	Ângulo do número complexo
any	Verdade se algum elemento for verdadeiro
append	Acrescenta elementos a um array
arange	Semelhante ao range, mas devolve um array
array	Converte uma lista para um array
array_split, hsplit, vsplit	Divide um array em várias partes
astype	Converte o tipo de dados de um array
average	Média de um array
clip	Força os valores a estarem num intervalo
concatenate	Concatena arrays
copy	Retorna uma cópia de um dado array
cross	Produto externo (vectorial)
delete	Remove elementos do array
diag	Matriz diagonal
diff(x)	Diferenças entre os elementos de x
eye	Matriz identidade
flat	Iterador que percorre todos os elementos do array
flatten	Converte para 1D, retorna um novo array
frombuffer	Converte uma string para array
fromfile	Lê um array a partir de um ficheiro
inner	Produto interno
insert	Insere novos elementos numa dada posição
inv	Matriz inversa
linspace(inicio,fim,N)	Array com N elementos, com os valores de inicio a fim
mat, matrix	Converte para matriz
max	Valor máximo
min	Valor mínimo
newaxis	Acrescenta uma nova dimensão
nonzero	Índices dos elementos diferentes de zero
ravel	Versão 1D do array
reshape	Altera as dimensões do array, mantendo o nº de elementos
resize	Altera o tamanho do array
select	Aplica operações diferentes conforme as condições indicadas
shape	A forma do array
shuffle	Mistura aleatoriamente os elementos
sort	Ordena o array
sum	Soma de todos os elementos
swapaxes	Troca os eixos indicados
take	Retorna os valores do array de índices específicado
tofile	Escreve um array para ficheiro
tolist	Converte um array para uma lista Python
unique(x)	Todos os valores diferentes de x
where(condição,a,b)	Valores de a ou b dependendo de condição
zeros, ones	Array inicializado com o valor zero (ou um)

Tabela 8.3: Outras funções do numpy

```
k = 1.0
#Para k=1, este método é estável se
#dt/dx**2 <= 1/2

x = np.arange(-10.0,10.0,dx)
t = np.arange(0.0,10.0,dt)
nx = len(x)
nt = len(t)
#Distribuição inicial
u = np.exp(-(x+2.0)**2)+np.exp(-(x-2.0)**2)
sol = np.zeros((nt,nx),dtype=u.dtype)
sol[0,:] = u

for i in xrange(1,nt):
    u[1:-1] += (k*dt/dx**2)*( u[:-2] -2.0*u[1:-1] + u[2:])
    sol[i,:] = u

pylab.imshow(sol,origin="lower",aspect="auto", \
             extent=(x[0],x[-1],t[0],t[-1]))
pylab.colorbar()
pylab.xlabel("x")
pylab.ylabel("t")
pylab.show()
```

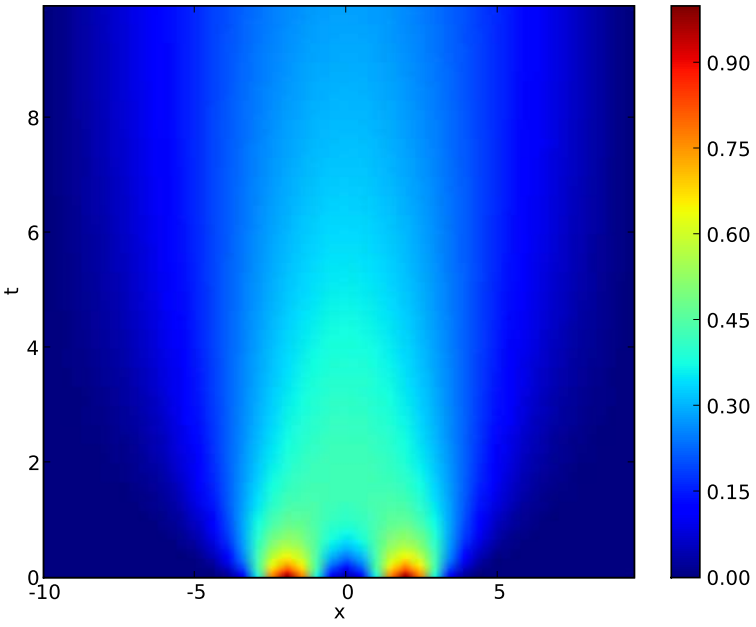


Figura 8.1: Equação do calor

É importante usar operações vectoriais tais como as do exemplo acima, e não ciclos **for** a percorrer cada elemento do array, pois estes últimos podem ser até cerca de 100 vezes mais lentos.

8.2 Scipy

O scipy complementa o suporte de vectores multidimensionais do numpy com módulos com rotinas gerais para cálculo númerico como minimização, integração, transformadas de Fourier rápidas e outras tarefas comuns.

8.2.1 optimize

Rotinas gerais de optimização e minimização.

Método	Descrição
fmin	Algoritmo simplex Nelder-Mead
fmin_powell	Powell's (modified) level set method
fmin_cg	Algoritmo de gradiente conjugado não linear
fmin_bfgs	Método quasi-Newton (Broydon-Fletcher-Goldfarb-Shanno)
fmin_ncg	Gradiente conjugado usando o método de Newton
leastsq	Método dos mínimos quadrados
fsolve	Cálculo de zeros de uma função
brentq	Interpolação quadrática pelo método de Brent
bisect	Método da bissecção
newton	Método de Newton

Tabela 8.4: Alguns métodos de scipy.optimize

8.2.2 stats

Um conjunto de distribuições de probabilidade incluindo por exemplo as distribuições normal, Chi-quadrado, Cauchy, Gamma, Logística, T de Student, as distribuições discretas binomial, Bernoulli, geométrica e Poisson entre outras. Inclui também as funções estatísticas usuais como a media, mediana, moda, etc.

8.2.3 linalg

Rotinas de álgebra linear.

Método	Descrição
inv	Inverte uma matriz quadrada
solve	Resolve um sistema de equações lineares
det	Determinante de uma matriz quadrada
norm	Norma
lstsq	Mínimos quadrados lineares
eig	Valores e vectores próprios de uma matriz quadrada
eigvals	Valores próprios de uma matriz quadrada
lu	Decomposição LU de uma matriz
cholesky	Decomposição Cholesky de uma matriz
qr	Decomposição QR de uma matriz

Tabela 8.5: Alguns métodos de scipy.linalg

8.2.4 signal

Rotinas de processamento de sinal

Método	Descrição
convolve	Convolução
correlate	Correlação
fftconvolve	Convolução usando transformadas de Fourier rápidas
bspline	B-splines
medfilt	Filtro "median"
sawtooth	Onda dentes de serra
square	Onda quadrada
boxcar	Janela boxcar
triang	Janela triangular
gaussian	Janela gaussiana
hann	Janela de Hann
hamming	Janela de Hamming

Tabela 8.6: Alguns métodos de scipy.signal

8.2.5 fftpack

Transformadas de Fourier rápidas

Método	Descrição
fft / ifft	Transformada de Fourier 1D
fftn / ifftn	Transformada de Fourier multidimensional
fftshift	Muda a frequência zero para o centro do espectro
hilbert / ihilbert	Transformada de Hilbert
diff	Derivação de sequências periódicas

Tabela 8.7: Alguns métodos de scipy.fftpack

8.2.6 integrate

Integração e resolução de equações diferenciais ordinárias

Método	Descrição
quad	Rotina geral de integração
dblquad	Rotina geral para cálculo de integrais duplos
tplquad	Rotina geral para cálculo de integrais triplos
fixed_quad	Quadratura Gaussiana
quadrature	Quadratura Gaussiana com tolerância fixa
romberg	Quadratura de uma função pelo método de Romberg
trapz / simps / romb	Quadratura de valores igualmente espaçados
odeint / ode	Resolução de um sistema de equações diferencias ordinárias

Tabela 8.8: Alguns métodos de scipy.integrate

8.2.7 outras

Para além dos componentes do scipy apresentados acima, destacam-se:

- interpolate - Interpolação usando splines
- linsolve - Resolução de sistemas de equações lineares
- sparse - Matrizes esparsas
- special - Funções especiais (Airy, elípticas, erf, Bessel, gamma, Hankel, Legendre, etc.)

Este resumo não é uma descrição exaustiva das funcionalidades do scipy. Para uma descrição completa consulte a documentação desta biblioteca.

8.3 VPython

O VPython, ou "Visual Python" (`http://vpython.org/`), é uma biblioteca Python que permite criar gráficos 3D interactivos particularmente apropriados para demonstrações de princípios físicos.

Listagem 8.5:

```
from visual import *
redbox=box(pos=vector(4,2,3),size=(8.,4.,6.), \
           color=color.red)
greenball=sphere(pos=vector(4,7,3), radius=2, \
                 color=color.green)
```

É possível rodar a figura mantendo o botão direito do rato premido sobre a janela e movendo-o. Procedendo da mesma maneira com o botão do meio permite-nos ampliar ou reduzir. As chamadas *box* e *sphere* criam os respectivos objectos. Se for necessário referir-nos a esses objectos posteriormente é necessário atribuir essas chamadas a uma variável como no exemplo.

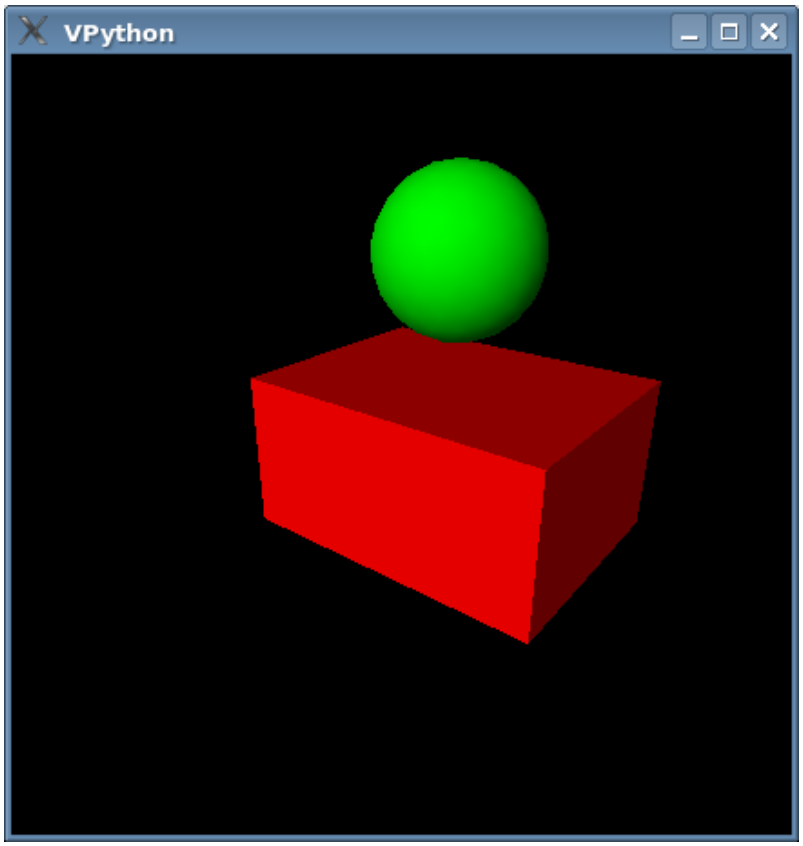


Figura 8.2: Exemplo de VPython

Comando	Descrição
cylinder(pos=<*ponto*>, axis=<*vec*>, radius=<*n*>)	Cilindro
arrow(pos=<*ponto*>, axis=<*vec*>)	Seta
cone(pos=<*ponto*>, axis=<*vec*>, radius=<*n*>)	Cone
pyramid(pos=<*ponto*>, size=<*vec*>)	Pirâmide
sphere(pos=<*ponto*>, radius=<*n*>)	Esfera
box(pos=<*ponto*>, length=<*n*>, height=<*n*>, width=<*n*>)	Caixa
ellipsoid(pos=<*ponto*>, length=<*n*>, height=<*n*>, width=<*n*>)	Elipsóide
curve([<*ponto1*>,<*ponto2*>,...],radius=<*n*>)	Curva
ring(pos=<*ponto*>, axis=<*vec*>, radius=<*n*>, thickness=<*n*>)	Anel
helix(pos=<*ponto*>, axis=<*vec*>, radius=<*n*>)	Hélice
label(pos=<*ponto*>, text="*Texto*")	Texto

Tabela 8.9: Objectos VPython

8.3.1 Vectores

O VPython suporta o tipo vector.

Listagem 8.6:

```
from visual import *
a = vector(1.,2.,3.)
b = vector(4.,5.,6.)
c=a+b
d = mag(c) # d é um escalar
f = cross(a,b) # produto vectorial
g = dot(a,b) # produto escalar
h = norm(a) # Versor segundo a direcção a
```

8.3.2 Objectos

Na tabela 8.9, <*ponto*> deve ser substituído por um dado ponto no espaço 3D como por exemplo (1,2,3) e <*vec*> por um vector, como por exemplo (1,0,0), e sendo <*n*> um número real.

8.4 matplotlib

O matplotlib é uma biblioteca que produz gráficos 2D de elevada qualidade, disponível em `http://matplotlib.sourceforge.net/`, onde também pode encontrar exemplos e um manual de utilizador. Seguem-se alguns exemplos de utilização desta biblioteca.

8.4.1 plot

```
import pylab
import numpy as np
```

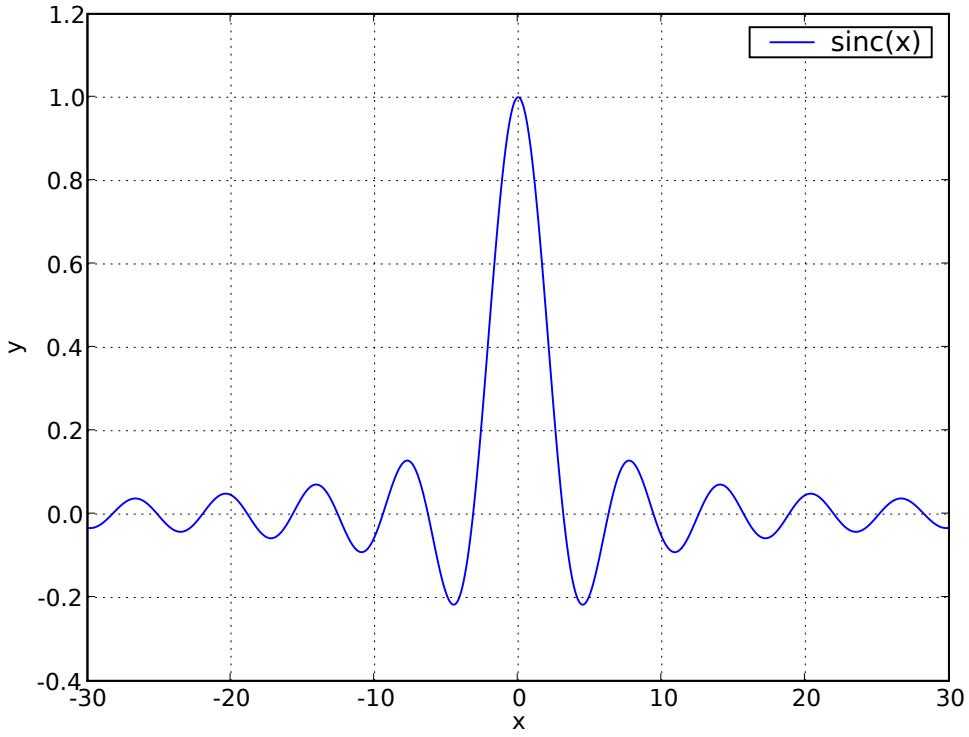


Figura 8.3: plot

```
x = np.arange(-30.0,30.0,0.01)
y = np.sin(x)/x
pylab.plot(x,y,label="sinc(x)")
pylab.xlabel("x")
pylab.ylabel("y")
pylab.legend()
pylab.grid()
pylab.show()
```

8.4.2 subplot

```
import pylab
import numpy as np

def grafico(m):
    x = np.arange(-10.0,10.0,0.01)
    y = np.sin(m*x)
    pylab.plot(x,y)
    pylab.title("m="+str(m))
```

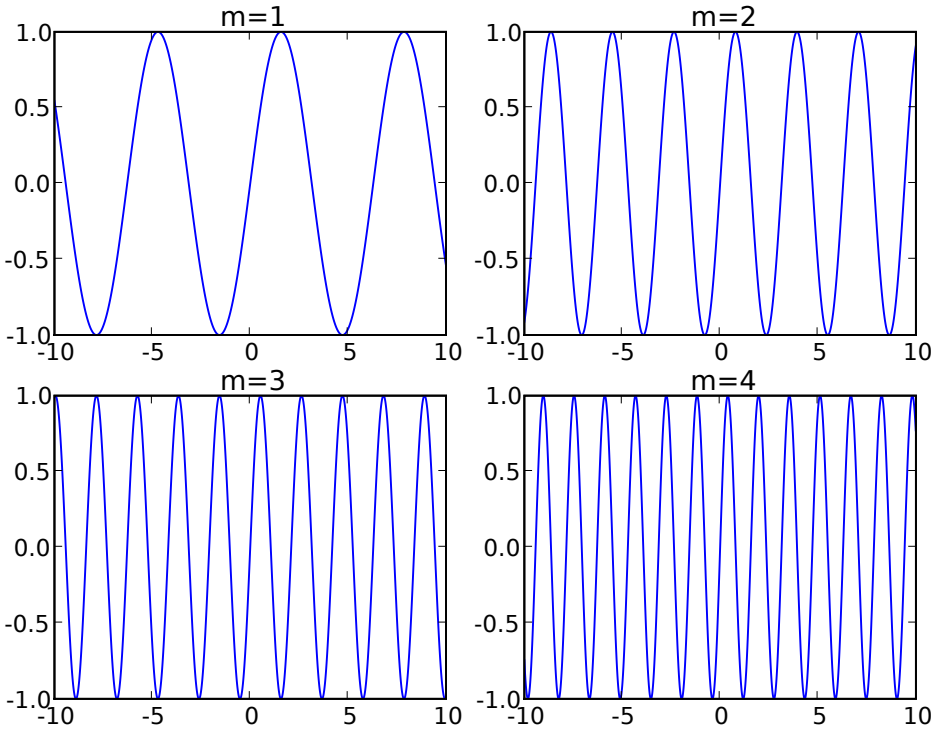


Figura 8.4: subplot

```
for i in range(4):
    pylab.subplot(2,2,i+1)
    grafico(i+1)
pylab.show()
```

8.4.3 hist

```
import pylab
import numpy as np
import numpy.random as random

#Média 0.0, desvio padrão 1.0, 10000 elementos
x = random.normal(0.0,1.0,size=10000)
pylab.hist(x,50)
pylab.show()
```

8.4.4 scatter

```
import pylab
```

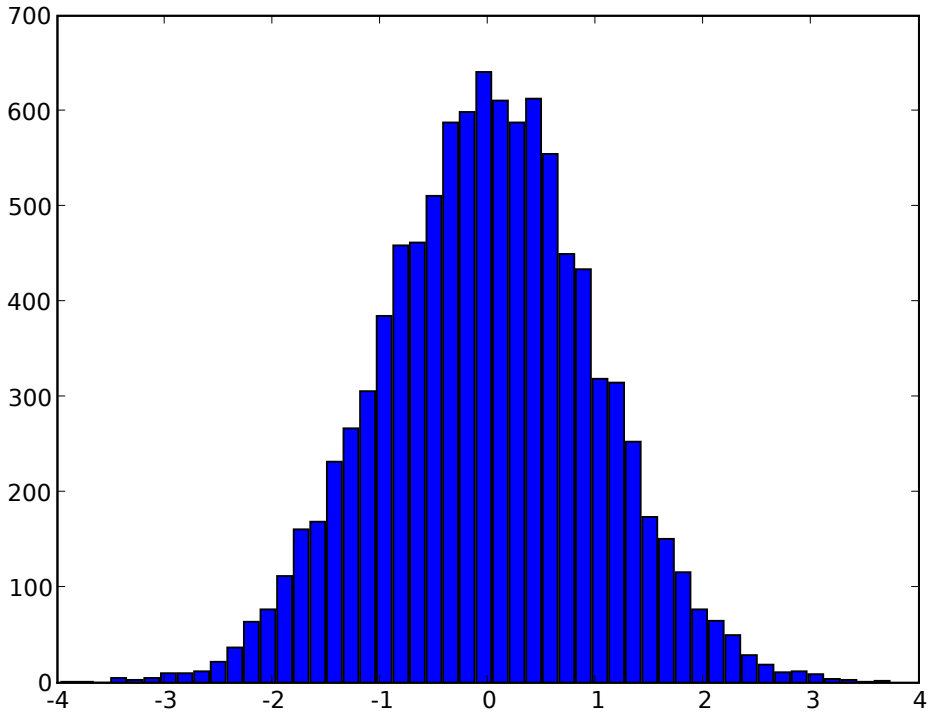


Figura 8.5: subplot

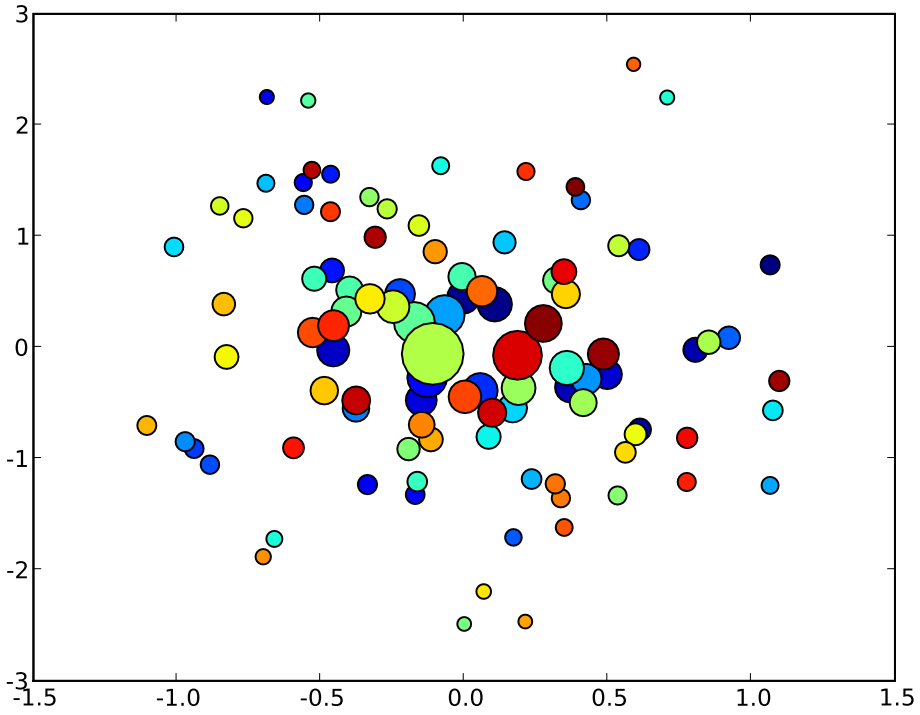


Figura 8.6: scatter

```
import numpy as np
import numpy.random as random

n = 100
x = random.normal(0.0,0.5,size=n)
y = random.normal(0.0,1.0,size=n)
area = 100.0/np.sqrt(x**2+y**2)
cor = np.arange(0,n,1)
pylab.scatter(x,y,c=cor,s=area,alpha=0.75)
pylab.show()
```

8.4.5 imshow

```
import pylab
import numpy as np
x = np.arange(-5.0,5.0,0.01)
y = x
X,Y = np.meshgrid(x,y)
pylab.imshow(np.sin(X)*np.cos(Y), \
        extent=[-5.0,5.0,-5.0,5.0], \
```

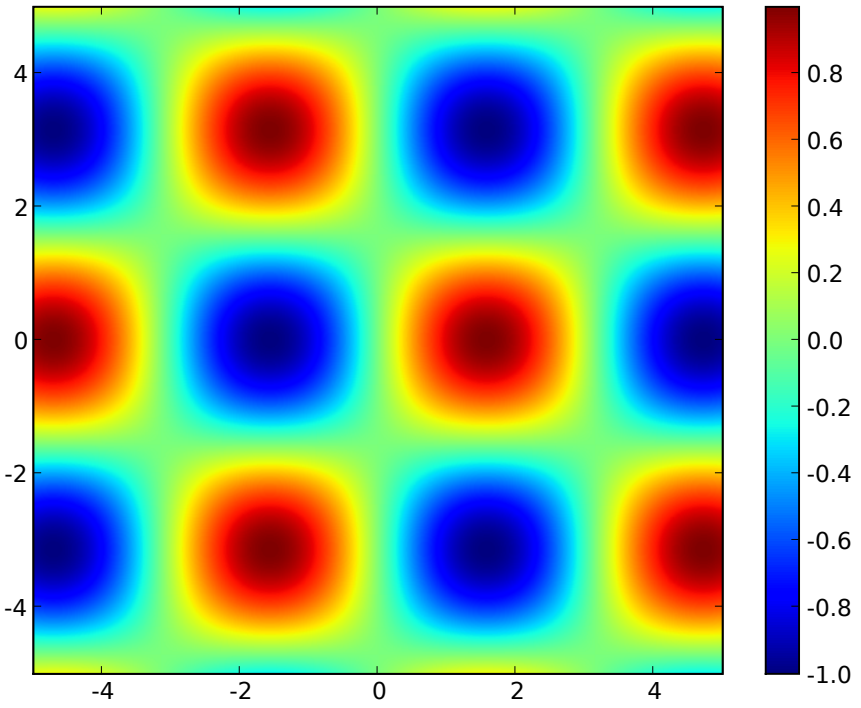


Figura 8.7: imshow

```
        origin="lower",aspect="auto")
pylab.colorbar()
pylab.show()
```

8.5 Armazenamento de dados de simulações

Ao efectuar simulações é normalmente necessário armazenar os dados obtidos. A forma mais simples é escrever cada conjunto de dados como um ficheiro de texto. Este método não é o ideal, pois ocupa mais espaço do que o necessário, espalha a informação de uma mesma simulação por vários ficheiros e não se auto-descreve, isto é, não inclui informação acerca do que é que está armazenado. Para resolver estes problemas foram desenvolvidos formatos standard para armazenamento de dados científicos, como o netCDF e o HDF5. Existem bibliotecas que suportam estes formatos para todas as linguagens de programação mais comuns. No Python temos o PyTables (`http://www.pytables.org/`) para o HDF5 e o ScientificPython (entre outros) para o netCDF. Assim usando estes formatos podemos desenvolver uma simulação numa linguagem de programação, armazenar os dados resultantes, e visualizar ou processar estes dados num outro programa escrito noutra linguagem. Podemos também partilhar es-

tes ficheiros, uma vez que são independentes do sistema operativo que se esteja a usar.

8.5.1 Acesso a ficheiros netCDF com o ScientificPython

O ScientificPython (`http://dirac.cnrs-orleans.fr/plone/software/scientificpython/`) têm uma interface simples para ler e escrever ficheiros netCDF. O exemplo seguinte define uma função que escreve valores $y(x)$ para um ficheiro netCDF.

Listagem 8.7:

```
from Scientific.IO.NetCDF import *
import numpy as np

def store(x_data,y_data,filename):
    n = len(x_data)
    file = NetCDFFile(filename,"w")
    file.createDimension("x",n)
    x = file.createVariable("x","d",("x",))
    y = file.createVariable("y","d",("x",))
    x[:] = x_data
    y[:] = y_data
    file.close()

x = np.arange(0,2.0*np.pi,0.1)
y = np.sin(x)
store(x,y,"test.nc")
```

Os métodos disponibilizados por este interface são:

- NetCDFFile(ficheiro,modo) - Abre um ficheiro para leitura (modo "r"), escrita (modo "w") ou para acrescentar dados (modo "a"). Os ficheiros netCDF têm usualmente a extensão .nc. Para um dado ficheiro aberto com esta função, como por exemplo

  file = NetCDFFile("teste.nc","w")

temos os métodos

- file.createDimension(dimensao,tamanho) - O NetCDF define todos os valores em termos de dimensões, pelo que antes de se poderem criar quaisquer variáveis é necessário criar primeiro as dimensões que elas usam.
- file.dimensions - Dicionário com todas as dimensões e respectivos tamanhos
- file.createVariable(nome,tipo,dimensoes) - Cria uma variável onde vão poder ser armazenados dados. Os parâmetros são o nome da variável, o tipo ("f" - real precisão simples, "d" - real precisão dupla, "i" - inteiro, "l" - inteiro longo, "c" - caracter, "b" - byte) e dimensoes é um tuplo com os nomes das dimensões desta variável.

- file.variables - Dicionário contento todas a variáveis, e respectivos valores. Para obter um array x com os valores da variável x usariamos:

```
x = file.variables["x"][:]
```

- file.sync() - Guarda todas as alterações pendentes no disco.
- file.close() - Fecha o ficheiro

Podemos aceder e criar atributos globais do ficheiro netCDF acedendo à propriedade respectiva de file. Por exemplo, para criar o atributo nome com o valor "teste":

```
file.nome = "teste"
```

Podemos ler os ficheiros resultantes com qualquer programa que suporte o formato netCDF, como o ncview (`http://meteora.ucsd.edu/~pierce/ncview_home_page.html`). O conjunto de programas NCO ("netCDF Operators"), disponível em `http://nco.sourceforge.net/`, é útil para manipular estes ficheiros.

Capítulo 9

Conclusão

A linguagem Python torna a programação mais fácil, não limitando o utilizador. Para aprender mais acerca do Python pode consultar:

- A documentação oficial (`http://python.org/doc/`) é muito completa e legível, incluindo um tutorial de introdução à linguagem, bem como referências exaustivas da linguagem e biblioteca padrão.
- "How to think like a computer scientist", disponível em `http://openbookproject.net/thinkCSpy/` proporciona uma introdução simples à linguagem.
- "Dive into Python", disponível em `http://diveintopython.org/` expõe a linguagem Python de uma introdução simples até algumas aplicações mais avançadas.
- O livro "Learning Python" de Mark Lutz da editora O'Reilly, actualmente na 3ª edição, é um livro introdutório muito completo e repleto de exemplos.

Em relação às outras bibliotecas utilizadas, todas elas estão documentadas nos próprios módulos ("docstrings") e nos respectivos sítios web:

- O numpy possuí um livro gratuito, disponível em `http://www.tramy.us/guidetoscipy.html`.
- O matplotlib (`http://matplotlib.sourceforge.net/`) tem um manual de utilizador extenso, um guia de referência de todas as funcionalidades disponibilizadas e exemplos de utilização.
- A documentação do scipy é mais limitada, estando em curso esforços para a melhorar. Entretando, a melhor documentação deste módulo é a das "docstrings".