

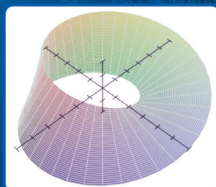
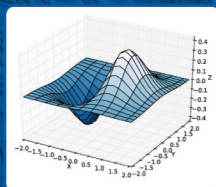
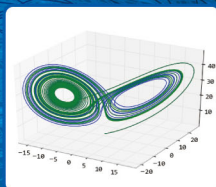
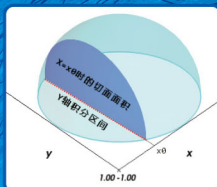
从 $e^{i\pi}+1$ 开始



Scientific Computing With Python

Python

科学计算



张若愚 著

- ★ NumPy——快速处理数据
- ★ SymPy——符号运算好帮手
- ★ Traits——为Python添加类型定义
- ★ Chaco——交互式图表
- ★ Mayavi——更方便的可视化
- ★ OpenCV——图像处理和计算机视觉
- ★ 数字信号、滤波器、频域处理
- ★ 用C语言提高计算效率

- ★ SciPy——数值计算库
- ★ matplotlib——绘制精美的图表
- ★ TraitsUI——轻松制作用户界面
- ★ TVTK——数据的三维可视化
- ★ VPython——制作3D演示动画
- ★ 声音与视频数据处理
- ★ 动画模拟、分形几何

清华大学出版社

Python 科学计算

张若愚 著

清华大学出版社

北 京

内 容 简 介

本书介绍如何用 Python 开发科学计算的应用程序，除了介绍数值计算之外，还着重介绍如何制作交互式的 2D、3D 图像，如何设计精巧的程序界面，如何与 C 语言编写的高速计算程序结合，如何编写声音、图像处理算法等内容。书中涉及的 Python 扩展库包括 NumPy、SciPy、SymPy、matplotlib、Traits、TraitsUI、Chaco、TVTK、Mayavi、VPython、OpenCV 等，涉及的应用领域包括数值运算、符号运算、二维图表、三维数据可视化、三维动画演示、图像处理以及界面设计等。

书中以大量实例引导读者逐步深入学习，每个实例程序都有详尽的解释，并都能在本书推荐的运行环境中正常运行。此外，本书附有大量的图表和插图，力求减少长篇的理论介绍和公式推导，以便读者通过实例和数据学习并掌握理论知识。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

Python 科学计算 / 张若愚 著. —北京：清华大学出版社，2012.1

ISBN 978-7-302-27360-8

I. P… II. 张… III. 软件工具—程序设计 IV. TP311.56

中国版本图书馆 CIP 数据核字(2011)第 232994 号

责任编辑：王 军 李维杰

装帧设计：牛艳敏

责任校对：成凤进

责任印制：

出版发行：清华大学出版社

地 址：北京清华大学学研大厦 A 座

<http://www.tup.com.cn>

邮 编：100084

社 总 机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 刷 者：

装 订 者：

经 销：全国新华书店

开 本：185×260

印 张：39.75

字 数：941 千字

附光盘 1 张

版 次：2012 年 1 月第 1 版

印 次：2012 年 1 月第 1 次印刷

印 数：1~3000

定 价：98.00 元

产品编号：

前言

Python 是一种面向对象的、动态的程序设计语言，具有非常简洁而清晰的语法，既可以用于快速开发程序脚本，也可以用于开发大规模的软件，特别适合于完成各种高层任务。

随着 NumPy、SciPy、matplotlib、ETS^①等众多程序库的开发，Python 越来越适合于做科学计算。与科学计算领域最流行的商业软件 MATLAB 相比，Python 是一门真正的通用程序设计语言，比 MATLAB 所采用的脚本语言的应用范围更广泛，有更多程序库的支持，适用于 Windows 和 Linux 等多种平台，完全免费并且开放源码。虽然 MATLAB 中的某些高级功能目前还无法替代，但是对于基础性、前瞻性的科研工作和应用系统的开发，完全可以用 Python 来完成。

本书介绍如何用 Python 开发科学计算的应用程序，除了介绍数值计算之外，还着重介绍了如何制作交互式二维、三维图像，如何设计精巧的程序界面，如何与 C 语言编写的高速计算程序结合，如何编写声音、图像处理算法等内容。

由于 Python 的相关资源非常多，本书不可能全部涉及，相信读者在掌握本书所介绍的一些相关知识之后，只要充分利用互联网的搜索功能，就一定能够很快地找到合适的 Python 解决方案。此外，由于绝大多数 Python 资源都开放源代码，因此读者将会很容易地对感兴趣的内容进行深度挖掘和研究。

本书适合于工科高年级本科生、研究生、工程技术人员以及计算机开发人员阅读。实例篇以信号处理为主，通过简单易懂的 Python 源程序，实际演示信号处理的一些基础知识和原理，因此特别适合于相关专业的学生作为扩展视野的补充阅读教材。

阅读本书的读者需要掌握 Python 语言的一些基础知识，下面是一个“自我检测列表”，如果读者熟悉下述内容，阅读本书的实例源代码就应该没有困难。此外由于 Python 程序简单易读，即使读者没有接触过 Python，也可以边阅读本书边通过其他书籍或免费教程学习 Python。

- 基本语法：库的载入(import)、循环(for、while)、判断(if)、函数定义(def)
- 基本数据类型的用法：列表(list)、字典(dict)、元组(tuple)、字符串
- 面向对象的基本语法：类(class)、继承
- C 语言编程的基础知识^②

有关 Python 语言的基础知识，可以参考啄木鸟社区的 Python 图书简介。



<http://wiki.woodpecker.org.cn/moin/PyBooks>

啄木鸟社区的 Python 图书概览

本书所有演示程序，均在 Windows XP 系统下采用 Python(x,y)通过测试。如果读者觉得安装众多的 Python 程序库很麻烦，不妨下载安装 Python(x,y)，或者直接使用本书所附光盘中的 Python(x,y)安装程序。

^① 全称为 Enthought Tool Suite，是 Enthought 公司开发的开源科学计算应用程序开发包。

^② 为了提高程序的运行效率，有时需要使用 C 语言编写 Python 的扩展模块，第 16 章“用 C 语言提高计算效率”中介绍的内容需要读者熟悉 C 语言编程。

Preface

Python is rightfully viewed as a general purpose language, well suited for web development, system administration, and general purpose business applications. It's has earned this reputation well by powering web sites such as YouTube, installation tools integral to Red Hat's operating system, and large corporate IT systems from cloud cluster management to investment banking. Python has also established itself firmly in the world of scientific computing covering a wide range of applications from seismic processing for oil exploration to quantum physics. This breadth of applicability is significant because these seemingly disparate uses often overlap in important ways. Applications that can easily connect to databases publish information to the web, and efficiently carry out complex calculations are now critical in many industries. Python's primary strength is that it allows developers to build such tools quickly.

Python's scientific computing roots actually go quite deep. Guido van Rossum created the language while at CWI, the Center for Mathematics and Computer Science, in the Netherlands. As interest developed outside the center, others began to contribute. The first several Python workshops, starting in 1994, were held an ocean away at scientific institutions such as NIST (National Institute of Instruments and Technology), the US Geological Society, and LLNL (Lawrence Livermore National Laboratories), all science centric institutions. At the time, Python 1.0 had recently been released and the attendees were just beginning to hammer out the design of its mathematical tools. A decade and a half later, it is gratifying to see how far we have come both in the amazing capabilities of the tool set and the diversity of the community. It is somehow fitting that the first comprehensive book (that I know of) covering the primary scientific computing tools for Python is composed and published, another ocean away, in Chinese. Looking forward a decade and a half, I can hardly wait to see what we will all build together.

Guido, himself, was not a scientist or engineer. He sat squarely in the computer science branch of CWI and created Python to ease the pain of building system administration tools for the Amoeba operating system. At the time, the tools were being written in C. Python was to be the tool that "bridged the gap between shell scripting and C." Operating system tools are not even in the same neighborhood as matrix inversions or fast Fourier transforms, but, as the language emerged, scientists around the world were some of its earliest adopters. Guido had succeeded in creating an elegantly expressive language that coupled nicely with their existing C and Fortran code. And, in Guido, they had a language designer willing to listen and add critical features, such as complex numbers, specifically for the scientific community. With the creation of Numeric, the precursor to NumPy, Python gained a fast and powerful number crunching tool that solidified Python's role as a leading computational language in the coming decades.

For some, the term “scientific programming” conjures up visions of intricate algorithms described from “Numerical Recipes in C” or forged in late night programming sessions by graduate students. But the reality is the domain encompasses a much wider range of programming tasks from low level algorithms to GUI development with advanced graphics. This latter topic is too often underestimated in terms of importance and effort. Fortunately, Ruoyu Zhang has done us the service of covering all facets of the scientific programming in this book. Beginning with the foundational Numpy library the algorithmic toolboxes in SciPy he provides the fundamental tools for any scientific application. He then aptly covers the 2D plotting and 3D visualization libraries provided by matplotlib, chaco, and mayavi. Application and GUI development with Traits and Traits UI, and coupling to legacy C libraries through Cython, Weave, ctypes, and SWIG are well covered as well. These core tools are rounded out by coverage of symbolic mathematics with SymPy and various other useful topics.

It’s truly gratifying to see all of these topics aggregated into a single volume. It provides a one-stop shop that can lead you from the beginning steps to a polished and full featured application for analysis and simulation.

Eric Jones
2011/12/8

序

Python 理所当然地被视为一门通用的程序设计语言，非常适合于网站开发、系统管理以及通用的业务应用程序。它为诸如 YouTube 这样的网站系统、Red Hat 操作系统中不可或缺的安装工具以及从云管理到投资银行等大型企业的 IT 系统提供技术支持，从而赢得了如此高的声誉。Python 还在科学计算领域建立了牢固的基础，覆盖了从石油勘探的地震数据处理到量子物理等范围广泛的应用场景。Python 这种广泛的适用性在于，这些看似不同的应用领域通常在某些重要的方面是重叠的。易于与数据库连接、在网络上发布信息并高效地进行复杂计算的应用程序对于许多行业是至关重要的，而 Python 最主要的长处就在于它能让开发者迅速地创建这样的工具。

实际上，Python 与科学计算的关系源远流长。吉多·范罗苏姆创建这门语言，还是在他在荷兰阿姆斯特丹的国家数学和计算机科学研究学会(CWI)的时候。当时只是作为“课余”的开发，但是很快其他人也开始为之做出贡献。从 1994 年开始的头几次 Python 研讨会，都是在大洋彼岸的科研机构举行的。例如国家标准技术研究所(NIST)、美国地质学会以及劳伦斯利福摩尔国家实验室(LLNL)，所有这些都是以科研为中心的机构。当时 Python 1.0 刚刚发布，与会者们就已经开始打造 Python 的数学计算工具。10 多年过去了，我们欣喜地看到，我们在开发具有惊人能力的工具集以及建设多彩的社区方面做出了如此多的成绩。很合时宜的是，就我所知的第一本涵盖了 Python 的主要科学计算工具的综合性著作，在另一个海洋之遥的中国编著并出版了。展望今后的十几年，我迫不及待地想看到我们能共同创建出怎样的未来。

吉多他本人并不是科学家或工程师。他在 CWI 的计算机科学部门时，为了缓解为阿米巴(Amoeba)操作系统创建系统管理工具的痛苦，他创建了 Python。当时那些系统管理工具都是用 C 语言编写的。于是 Python 就成了填补 shell 脚本和 C 语言之间空白的工具。操作系统工具与计算逆矩阵或快速傅立叶变换是完全不同的领域，但是从 Python 诞生开始，世界各地的许多科学家就成了它最早期的采用者。吉多成功地创建了一门能与他们的 C 和 Fortran 代码完美结合的、具有优雅表现力的程序语言。并且，吉多是一位愿意听取建议并添加关键功能的语言设计师，例如支持复数就是专门针对科学领域的。随着 NumPy 的前身——Numeric 的诞生，Python 获得了一个高效且强大的数值运算工具，它巩固了在未来几十年中，Python 作为领先的科学计算语言的地位。

对于一些人来说，“科学计算编程”会让人联想起 *Numerical Recipes in C* 中描述的那些复杂算法，或是研究生们在深夜中努力打造程序的场景。但是真实情况所涵盖的范围更广泛——从底层的算法设计到具有高级绘图功能的用户界面开发。而后者的重要性却常常被忽视了。幸运的是在本书中，作者为我们介绍了科学计算编程所需的各个方面。从 NumPy 库和 SciPy 算法工具箱的基础开始，介绍了任何科学计算应用程序所需的基本工具。然后，本书很恰当地介绍了二维绘图以及三维可视化库——matplotlib、Chaco、Mayavi。用 Traits 和 TraitsUI 进行应用程序和界面开发，以及用 Cython、Weave、ctypes 和 SWIG 等与传统的 C 语言库相互结合等

内容在书中也有很好的介绍。除了这些核心的工具之外，本书还介绍了使用 SymPy 进行数学符号运算以及其他各种有用的主题。

所有这些主题都被汇编到一本书中真是一件令人欣喜的事情。本书所提供的一站式服务，能够指导读者从最初的入门直到创建一个漂亮的、全功能的分析与模拟应用程序。

Eric Jones

2011 年 12 月 8 日

关于序言作者

Eric Jones 是 Enthought 公司的 CEO，他在工程和软件开发领域拥有广泛的背景，指导 Enthought 公司的产品工程和软件设计。在共同创建 Enthought 公司之前，他在杜克大学电机工程学系从事数值电磁学以及遗传优化算法方面的研究，并获得了该系的硕士和博士学位。他教授过许多用 Python 做科学计算的课程，并且是 Python 软件基金会的成员。

关于 Enthought 公司

Enthought 是一家位于美国得克萨斯州首府奥斯汀的软件公司，主要使用 Python 从事科学计算工具的开发。本书中介绍的 NumPy、SciPy、Traits、TraitsUI、Chaco、TVTK 以及 Mayavi 均为该公司开发或维护的开源程序库。

目 录

第 1 章 软件包的安装和介绍	1		
1.1 Python 简介	1		
1.2 安装软件包	2		
1.2.1 Python(x,y)	2		
1.2.2 Enthought Python Distribution (EPD)	3		
1.3 方便的开发工具	3		
1.3.1 IPython	4		
1.3.2 Spyder	8		
1.3.3 Wing IDE 101	12		
1.4 函数库介绍	13		
1.4.1 数值计算库	13		
1.4.2 符号计算库	14		
1.4.3 界面设计	14		
1.4.4 绘图与可视化	14		
1.4.5 图像处理和计算机视觉	15		
第 2 章 NumPy——快速处理数据	16		
2.1 ndarray 对象	16		
2.1.1 创建数组	16		
2.1.2 存取元素	21		
2.1.3 多维数组	24		
2.1.4 结构数组	29		
2.1.5 内存结构	32		
2.2 ufunc 运算	35		
2.2.1 四则运算	37		
2.2.2 比较和布尔运算	39		
2.2.3 自定义 ufunc 函数	40		
2.2.4 广播	42		
2.2.5 ufunc 函数的方法	46		
2.3 多维数组的下标存取	48		
2.3.1 下标对象	48		
2.3.2 整数数组作为下标	49		
2.3.3 一个复杂的例子	51		
2.3.4 布尔数组作为下标	53		
2.4 庞大的函数库	54		
2.4.1 求和、平均值、方差	54		
2.4.2 最值和排序	55		
2.4.3 多项式函数	57		
2.4.4 分段函数	60		
2.4.5 统计函数	62		
2.5 线性代数	65		
2.5.1 各种乘积运算	65		
2.5.2 解线性方程组	67		
2.6 掩码数组	69		
2.7 文件存取	72		
2.8 内存映射数组	75		
第 3 章 SciPy——数值计算库	79		
3.1 常数和特殊函数	79		
3.2 优化——optimize	81		
3.2.1 最小二乘拟合	81		
3.2.2 函数最小值	84		
3.2.3 非线性方程组求解	86		
3.3 插值——interpolate	88		
3.3.1 B 样条曲线插值	88		
3.3.2 外推和 Spline 拟合	90		
3.3.3 二维插值	91		
3.4 数值积分——integrate	93		
3.4.1 球的体积	93		
3.4.2 解常微分方程组	95		
3.5 信号处理——signal	97		

3.5.1	中值滤波	97	5.2	Artist 对象	152
3.5.2	滤波器设计	98	5.2.1	Artist 对象的属性	154
3.6	图像处理——ndimage	100	5.2.2	Figure 容器	155
3.6.1	膨胀和腐蚀	101	5.2.3	Axes 容器	156
3.6.2	Hit 和 Miss	102	5.2.4	Axis 容器	159
3.7	统计——stats	105	5.2.5	Artist 对象的关系	163
3.7.1	连续和离散概率分布	105	5.3	坐标变换和注释	164
3.7.2	二项、泊松、伽玛分布	108	5.3.1	4 种坐标系	167
3.8	嵌入 C 语言程序——weave	112	5.3.2	坐标变换的步骤	169
第 4 章	SymPy——符号运算好帮手	115	5.3.3	制作阴影效果	173
4.1	从例子开始	115	5.3.4	添加注释	174
4.1.1	封面上的经典公式	115	5.4	绘图函数简介	177
4.1.2	球体体积	117	5.4.1	对数坐标图	177
4.2	数学表达式	119	5.4.2	极坐标图	178
4.2.1	符号	119	5.4.3	柱状图	179
4.2.2	数值	121	5.4.4	散列图	180
4.2.3	运算符和函数	122	5.4.5	图像	181
4.3	符号运算	125	5.4.6	等值线图	184
4.3.1	表达式变换和化简	125	5.4.7	三维绘图	187
4.3.2	方程	128	第 6 章	Traits——为 Python 添加类型	
4.3.3	微分	129		定义	190
4.3.4	微分方程	130	6.1	开发背景	190
4.3.5	积分	131	6.2	Trait 属性的功能	192
4.4	其他功能	133	6.3	Trait 类型对象	196
4.4.1	平面几何	133	6.4	Trait 的元数据	198
4.4.2	绘图	135	6.5	预定义的 Trait 类型	200
第 5 章	matplotlib——绘制精美 的图表	139	6.6	Property 属性	204
5.1	快速绘图	139	6.7	Trait 属性监听	206
5.1.1	使用 pyplot 模块绘图	139	6.8	Event 和 Button 属性	210
5.1.2	以面向对象方式绘图	142	6.9	Trait 属性的从属关系	211
5.1.3	配置属性	143	6.10	动态添加 Trait 属性	213
5.1.4	绘制多个子图	145	6.11	创建自己的 Trait 类型	215
5.1.5	配置文件	147	6.11.1	从 TraitType 继承	215
5.1.6	在图表中显示中文	149	6.11.2	使用 Trait()	217
			6.11.3	定义 TraitHandler 类	219

第 7 章 TraitsUI——轻松制作用户

界面	221
7.1 默认界面	221
7.2 用 View 定义界面	222
7.2.1 外部视图和内部视图	222
7.2.2 多模型视图	226
7.2.3 Group 对象	228
7.2.4 配置视图	231
7.3 用 Handler 控制界面和模型	232
7.3.1 用 Handler 处理事件	233
7.3.2 Controller 和 UIInfo 对象	237
7.3.3 响应 Trait 属性的事件	238
7.4 属性编辑器	240
7.4.1 编辑器演示程序	241
7.4.2 对象编辑器	243
7.4.3 字符串列表编辑器	248
7.4.4 对象列表编辑器	250
7.5 菜单、工具条和状态栏	252
7.6 设计自己的编辑器	255
7.6.1 Trait 编辑器的工作原理	255
7.6.2 制作 matplotlib 的编辑器	259
7.6.3 CSV 数据绘图工具	262

第 8 章 Chaco——交互式图表

8.1 面向脚本绘图	264
8.2 面向应用绘图	265
8.2.1 多条曲线	267
8.2.2 Plot 对象的结构	271
8.2.3 编辑绘图属性	275
8.2.4 容器(Container)	276
8.3 添加交互工具	279
8.3.1 平移和缩放	279
8.3.2 选取范围	282
8.3.3 选取数据点	284
8.3.4 套索工具	287
8.4 二次开发	289
8.4.1 用 Kiva 库在数组上绘图	290

8.4.2 Enable 库的组件	292
8.4.3 设计圆形选择工具	297
8.4.4 制作动画演示	301

第 9 章 TVTK——数据的三维可视化

9.1 流水线(Pipeline)	304
9.1.1 显示圆锥	304
9.1.2 用 ivtk 观察流水线	307
9.2 数据集(Dataset)	313
9.2.1 ImageData	313
9.2.2 RectilinearGrid	318
9.2.3 StructuredGrid	319
9.2.4 PolyData	321
9.3 可视化实例	324
9.3.1 切面	325
9.3.2 等值面	330
9.3.3 流线	333
9.4 TVTK 的改进	337
9.4.1 TVTK 的基本用法	338
9.4.2 Trait 属性	339
9.4.3 序列化(Pickling)	339
9.4.4 集合迭代	340
9.4.5 数组操作	341

第 10 章 Mayavi——更方便的可视化

10.1 用 mlab 快速绘图	343
10.1.1 点和线	343
10.1.2 Mayavi 的流水线	345
10.1.3 二维图像的可视化	348
10.1.4 网格面	352
10.1.5 修改和控制流水线	356
10.1.6 标量场	358
10.1.7 矢量场	361
10.2 Mayavi 和 TVTK 之间的关系	363
10.2.1 显示 TVTK 流水线	363
10.2.2 两条流水线之间的关系	365

10.3	Mayavi 应用程序	367	12.3.1	几何变换	428
10.3.1	操作流水线	368	12.3.2	重映射——remap	430
10.3.2	命令行和对象浏览器	371	12.3.3	直方图统计	433
10.4	将 Mayavi 嵌入到界面中	374	12.3.4	二维离散傅立叶变换	437
第 11 章	VPython——制作 3D 演示		12.4	图像识别	440
	动画	378	12.4.1	用霍夫变换检测直线 和圆	440
11.1	场景、物体和照相机	378	12.4.2	图像分割	444
11.1.1	控制场景窗口	380	12.4.3	用 SURF 进行特征匹配	450
11.1.2	控制照相机	383	第 13 章	数据和文件	453
11.1.3	模型的属性	384	13.1	声音的输入输出	453
11.1.4	三维模型	387	13.1.1	读写 WAV 文件	453
11.2	制作动画演示	390	13.1.2	用 pyAudio 播放和录音	456
11.2.1	简单动画	390	13.2	视频的输入输出	459
11.2.2	盒子中反弹的球	391	13.2.1	读写视频文件	459
11.3	与场景交互	393	13.2.2	安装视频编码	464
11.3.1	响应键盘事件	394	13.3	读写 HDF5 文件	465
11.3.2	响应鼠标事件	394	13.4	读写 Excel 文件	469
11.4	用界面控制场景	397	13.4.1	写 Excel 文件	469
11.5	创建复杂模型	400	13.4.2	读 Excel 文件	471
11.5.1	faces() 的用法	400	第 14 章	数字信号系统	473
11.5.2	读入模型数据	402	14.1	FIR 和 IIR 滤波器	473
第 12 章	OpenCV——图像处理和计算机 视觉	408	14.2	FIR 滤波器设计	477
12.1	存储图像数据的 Mat 对象	409	14.2.1	用 firwin() 设计滤波器	479
12.1.1	Mat 对象和 NumPy 数组	410	14.2.2	用 remez() 设计滤波器	481
12.1.2	像素点类型	414	14.2.3	滤波器的级联	483
12.1.3	其他数据类型	415	14.3	IIR 滤波器设计	485
12.1.4	Vector 类型	417	14.3.1	巴特沃斯低通滤波器	485
12.1.5	在图像上绘图	418	14.3.2	双线性变换	487
12.2	图像处理	421	14.3.3	滤波器的频带转换	490
12.2.1	二维卷积	421	14.4	数字滤波器的频率响应	494
12.2.2	形态学运算	424	14.5	二次均衡滤波器设计工具	497
12.2.3	填充——floodFill	426	14.6	零相位滤波器	500
12.2.4	去瑕疵——inpaint	427	14.7	重取样	501
12.3	图像变换	428			

第 15 章 频域信号处理	505	第 17 章 自适应滤波器	571
15.1 FFT 演示程序	505	17.1 自适应滤波器简介	571
15.1.1 FFT 知识复习	505	17.1.1 系统识别	571
15.1.2 合成时域信号	509	17.1.2 信号预测	572
15.1.3 三角波 FFT 演示程序	511	17.1.3 信号均衡	572
15.2 观察信号的频谱	512	17.2 NLMS 计算公式	573
15.2.1 窗函数	515	17.3 用 NumPy 实现 NLMS 算法	575
15.2.2 频谱平均	517	17.3.1 系统辨识模拟	577
15.2.3 谱图	519	17.3.2 信号均衡模拟	579
15.3 卷积运算	522	17.3.3 卷积逆运算	581
15.3.1 快速卷积	522	17.4 用 C 语言加速 NLMS 运算	583
15.3.2 分段运算	524	17.4.1 用 SWIG 编写扩展模块	583
15.4 信号处理	526	17.4.2 用 Weave 嵌入 C++ 程序	586
15.4.1 基本框架	527	第 18 章 单摆和双摆模拟	588
15.4.2 频域滤波器	528	18.1 单摆模拟	588
15.4.3 频率变调处理	530	18.1.1 小角度时的摆动周期	589
15.4.4 用谱图差减法降噪	531	18.1.2 大角度时的摆动周期	590
15.5 Hilbert 变换	532	18.2 双摆模拟	592
第 16 章 用 C 语言提高计算效率	537	18.2.1 公式推导	592
16.1 用 ctypes 调用 DLL 库	537	18.2.2 微分方程的数值解	595
16.2 用 Weave 嵌入 C++ 程序	541	18.2.3 动画演示	598
16.2.1 Weave 的工作原理	541	第 19 章 分形几何	599
16.2.2 处理 NumPy 数组	543	19.1 Mandelbrot 集合	599
16.2.3 使用 blitz() 提速	546	19.1.1 使用 NumPy 加速计算	601
16.2.4 扩展模块	548	19.1.2 使用 Weave 加速计算	603
16.3 用 Cython 将 Python 编译成 C	549	19.1.3 连续的逃逸时间	604
16.3.1 编译 Cython 程序	549	19.1.4 Mandelbrot 演示程序	605
16.3.2 提高计算效率	550	19.2 迭代函数系统(IFS)	606
16.3.3 快速访问 NumPy 数组	553	19.2.1 二维仿射变换	610
16.4 用 SWIG 创建扩展模块	555	19.2.2 迭代函数系统设计器	610
16.4.1 SWIG 的调用方法 和实例	555	19.3 L-System 分形	613
16.4.2 SWIG 基础	558	19.4 分形山脉	616
16.4.3 SWIG 处理 NumPy 数组	566	19.4.1 一维中点移位法	616
		19.4.2 二维中点移位法	618
		19.4.3 菱形方形算法	619

软件包的安装和介绍

1.1 Python 简介

Python 是一种解释型、面向对象、动态的高级程序设计语言。自从 20 世纪 90 年代初 Python 语言诞生至今，它逐渐被广泛应用于处理系统管理任务和 Web 编程。目前 Python 已经成为最受欢迎的程序设计语言之一。2011 年 1 月，它被 TIOBE 编程语言排行榜评为 2010 年度语言。

由于 Python 语言的简洁、易读以及可扩展性，在国外用 Python 做科学计算的研究机构日益增多，一些知名大学已经采用 Python 教授程序设计课程。例如麻省理工学院的计算机科学及编程导论课程^①就使用 Python 语言讲授。众多开源的科学计算软件包都提供了 Python 的调用接口，例如著名的计算机视觉库 OpenCV、三维可视化库 VTK、医学图像处理库 ITK。而 Python 专用的科学计算扩展库就更多了，例如如下 3 个十分经典的科学计算扩展库：NumPy、SciPy 和 matplotlib，它们分别为 Python 提供了快速数组处理、数值运算以及绘图功能。因此 Python 语言及其众多的扩展库所构成的开发环境十分适合工程技术、科研人员处理实验数据、制作图表，甚至开发科学计算应用程序。

说起科学计算，首先会被提到的可能是 MATLAB。然而除了 MATLAB 的一些专业性很强的工具箱目前还无法替代之外，MATLAB 的大部分常用功能都可以在 Python 世界中找到相应的扩展库。和 MATLAB 相比，用 Python 做科学计算有如下优点：

- 首先，MATLAB 是一款商用软件，并且价格不菲。而 Python 完全免费，众多开源的科学计算库都提供了 Python 的调用接口。用户可以在任何计算机上免费安装 Python 及其绝大多数扩展库。
 - 其次，与 MATLAB 相比，Python 是一门更易学、更严谨的程序设计语言。它能让用户编写出更易读、易维护的代码。
 - 最后，MATLAB 主要专注于工程和科学计算。然而即使在计算领域，也经常会遇到文件管理、界面设计、网络通信等各种需求。而 Python 有着丰富的扩展库，可以轻易完成各种高级任务，开发者可以用 Python 实现完整应用程序所需的各种功能。
- 例如，笔者在一个模拟控制系统的项目中，完全用 Python 实现了系统模拟及算法优化，

^① 此课程的英文全称为 Introduction to Computer Science and Programming，它是麻省理工学院的开放课程之一，读者可以在 MIT 开放课程网(<http://ocw.mit.edu>)上找到此课程的全部视频及课件。

并在此基础上实现了应用程序必需的文档和数据库管理、用户界面设计、与机器设备及其他软件进行通信等功能。最后，整个应用程序可以随意安装到不同的计算机上，而不受任何商用软件的使用条款限制。

1.2 安装软件包

和 MATLAB 等商用软件不同，Python 的众多扩展库由许多社区分别维护和发布，因此要一一将它们收集齐全并安装到计算机中是一件十分耗费时间和精力事情。本书介绍两个科学计算用的 Python 集成软件包。读者只需要下载并执行一个安装程序，就能安装好本书涉及的所有扩展库。

1.2.1 Python(x,y)

Python(x,y)收集了众多的扩展库、文档和教程。在本书所附的光盘中提供了 Python(x,y) 2.6.6 的安装程序，为了保证能正确运行本书的所有实例程序，推荐读者以完全安装模式进行安装。Python(x,y)的版本号与它所使用的 Python 版本号相同。



<http://www.pythonxy.com>

Python(x,y)官方网址

为了确保本书的所有实例程序都能正常运行，请读者参照图 1-1 修改安装选项。将安装模式修改为完全安装，并将 Python(x,y)的安装路径改为“c:\pythonxy”。否则 Python 将可能无法正确调用 MinGW 编译扩展模块。请读者在安装结束之后，确认下列路径，在今后的章节中会经常用到它们：

c:\python26 Python 2.6 的安装路径，所有扩展库都可以在它的子目录“lib\site-packages”下找到

c:\pythonxy\doc 众多扩展库的说明文档和演示程序

c:\pythonxy\mingw MinGW C/C++编译器，在介绍用 C 语言编写扩展模块时会用到它

c:\pythonxy\swig 自动生成扩展模块接口的工具，在介绍用 C 语言编写扩展模块时会用到它

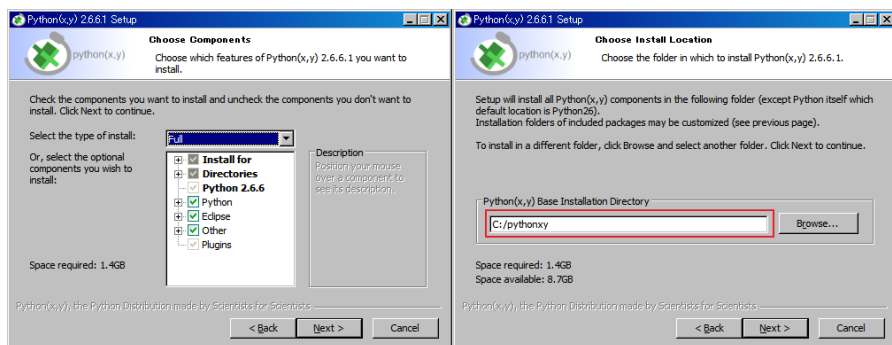


图 1-1 选择“Full”进行完全安装，并将 python(x,y) 的安装路径设置为“C:\pythonxy”

如果使用默认的安装路径,那么“pythonxy”目录会被安装到“c:\Program Files”下。

Python(x,y)提供了如图 1-2 所示的启动程序,从中可以快速启动各种工具,以及打开文档教程所在的文件夹。

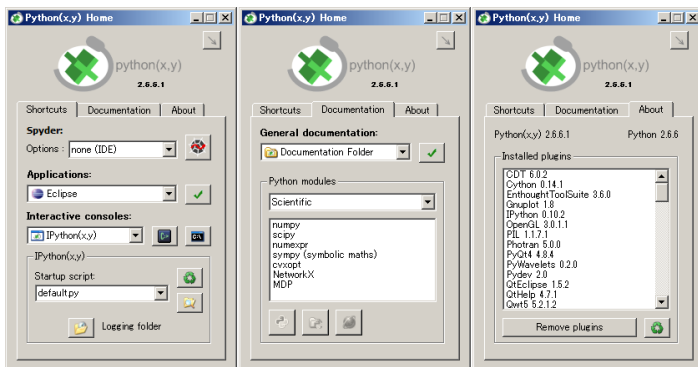


图 1-2 Python(x,y)的启动画面

界面中的 3 个选项卡如下:

- **Shortcuts:** 启动各种应用程序,包括 Eclipse、Mayavi、Spyder 集成开发环境、IPython 交互式命令行等。
- **Documentation:** 打开扩展库的文档,这里列出的文档不是很全面,推荐读者直接打开“c:\pythonxy\doc”文件夹查找文档。
- **About:** 查看所安装的扩展库的版本信息。

1.2.2 Enthought Python Distribution(EPD)

EPD 是一个商用的 Python 发行版本,同样包括了众多的科学软件包,而且作为教学使用是免费的,只需要提供一个教育单位的邮件地址,就可以收到 EPD 的教育版的下载地址。



<http://www.enthought.com/products/getepd.php>

EPD 的官方下载地址

1.3 方便的开发工具

本节介绍几个在开发和调试程序时经常会用到的工具软件,熟练掌握它们的用法能够起到事半功倍的效果。为了展示工具软件的功能,本节以一些扩展库作为演示。读者可以暂时忽略这些扩展库的用法,在后续的章节中会对它们进行详细介绍。

1.3.1 IPython

IPython 是 Python 的一个交互式命令行工具，与 Python 自带的命令行相比，它更容易使用，功能也更强大。它支持语法高亮、自动补全、自动缩进，并且内置了许多有用的功能和函数。

如果读者安装了 Python(x,y)，就可以从它的启动界面中运行 IPython，如图 1-3 所示。

从下拉列表中选择想运行的命令行配置，然后单击右侧的❶或❷按钮运行所选的命令行配置。其中，“Python”选项运行 Python 自带的命令行工具，而“IPython(x,y)”、“IPython(Qt)”、“IPython(wxPython)”、“IPython(mlab)”和“IPython(sh)”等几个选项，分别使用表 1-1 所示的参数来运行 IPython。

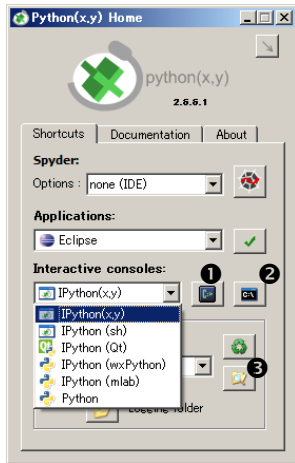


图 1-3 通过 Python(x,y) Home 启动 IPython 的各种选项

表 1-1 运行 IPython 的其他选项及对应参数

选 项	参 数
IPython(x,y)	-pylab -p xy
IPython(Sh)	-q4thread -p sh
IPython(Qt)	-q4thread
IPython(wxPython)	-wthread
IPython(mlab)	-wthread -p mlab
IPython(sh)	-q4thread -p sh

单击❶按钮将用一个名为 Console 的软件启动命令行，此软件使用 Windows 的窗口界面封装命令行界面，并且具有标签页功能。单击❷按钮将用 Windows 自带的命令行界面进行启动。

如果运行“IPython(x,y)”，在启动 IPython 之后将自动运行一个名为“default.py”脚本文件。此脚本默认执行以下函数库的导入操作：

```
import numpy
import scipy
from numpy import *
```

为了与 NumPy、SciPy 社区的推荐导入方式一致，请单击按钮❸，在打开的文件夹中添加一个名为“myimports.py”的文件，并在其中添加如下几行程序代码：

```
import numpy as np
import scipy as sp
import pylab as pl
```

此后运行“IPython(x,y)”的时候请选择“myimports.py”作为启动脚本。

如果要在命令行中和 matplotlib、TraitsUI、Mayavi 等图形界面程序进行交互，就需要给 IPython 传递“-wthread”、“-q4thread”或“-pylab”参数。当使用“-pylab”参数时，在调用 matplotlib 库的绘图函数进行绘图时，将立即显示图表。

下面我们以 matplotlib 绘图为例，实际操作一下。请读者选择“myimports.py”作为启动脚本，并运行 IPython(x,y) 命令行。然后在命令行中输入下面的语句，如果一切顺利，将会立即显示出如图 1-4 所示的正弦波形。

```
>>> x = np.linspace(0, 4*np.pi, 100)
>>> pl.plot(x, np.sin(x))
```

这里的 np 和 pl 是运行“myimports.py”之后的结果，它们分别表示 NumPy 和 pylab 模块。由于 IPython(x,y) 的命令行参数中有“-pylab”，pylab 模块中的所有符号也会被载入到当前的名称空间中，因此也可以用下面的程序绘制正弦波形：

```
>>> x = linspace(0, 4*pi, 100)
>>> plot(x, sin(x))
```

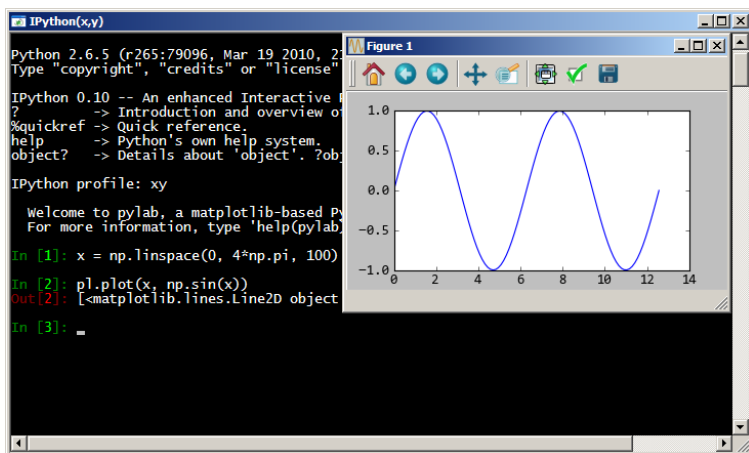


图 1-4 使用 IPython 交互式地绘制正弦波



本书中所有在 IPython 下输入的代码都用“>>>”作为提示符，而不使用 IPython 命令行的默认提示符“In[.]”。

在 IPython 中，可以很方便地使用如下功能：

- 自动补全：输入一部分文字之后按 Tab 键，IPython 将列出所有补全信息。用此功能可以快速输入对象的属性名或者进行文件名补全。
- 查看文档：输入需要查看文档的函数名，然后在后面添加一个或两个问号。“?”表示查看函数的文档，“??”表示查看其 Python 源代码。如果函数不是用 Python 编写的，就看不到其源代码。

- 执行 Python 程序: 用 `run` 命令运行指定的 Python 程序文件。默认是在一个新的环境中运行程序, 当程序退出时将程序运行环境中的对象复制到 IPython 环境中。如果运行 `run` 命令时添加 “-i” 参数, 在 IPython 的当前环境中执行程序, 程序即可直接访问 IPython 环境中的对象。
- 执行剪切板中的程序: 运行 `paste` 命令将在 IPython 环境中运行剪贴板中的程序代码, 它会自动删除代码中的提示符 “>>>”。运行 “`paste foo`” 将把剪切板中的内容复制到变量 `foo` 中。变量 `foo` 是 IPython 提供的 `SList` 列表类型, 它提供了很多对其内容进行操作的方法。
- 执行系统命令: 在要执行的系统命令之前添加一个 “!” 符号。例如, 如果执行 “`!test.py`”, 那么操作系统会运行 “`test.py`” 文件。和 `run` 命令不同, “`test.py`” 将在另外的进程中运行。

在 IPython 中使用 `run` 命令运行 Python 程序能够提高调试程序的速度。因为每次运行用户程序时, IPython 环境没有初始化, 已经载入的模块不需要重新载入, 对于 NumPy、SciPy 和 matplotlib 这样比较庞大的库, 能节省不少载入时间。下面是一个例子:



speedup_test.py

载入 NumPy、SciPy 和 pylab 库并绘制一个简单的频率扫描波

```
import numpy as np
from scipy import signal
import pylab as pl

t = np.linspace(0, 10, 1000)
x = signal.chirp(t, 5, 10, 30)
pl.plot(t, x)
pl.show()
```

此程序计算频率扫描波并使用图表来显示, 如果通过双击或者在命令行中输入文件名来直接运行此程序, 需要等几秒钟才能看到结果。如果先在 “`speedup_test.py`” 所在的文件夹启动 IPython, 然后运行:

```
>>> run speedup_test.py
```

第一次运行时由于同样需要载入所需的模块, 因此所需的时间和直接运行没有差别, 但是再次运行时由于不需要载入模块, 因此结果可以立即显示出来。此后还可以在 IPython 中输入程序, 查看变量的值或者修改界面的各种属性, 例如:

```
>>> x[:5] #查看扫描波的前 5 个值
array([ 1.          ,  0.95076436,  0.80716308,  0.58242516,  0.29822084])
>>> pl.gca().lines[0].set_color("r") #设置曲线颜色为红色
>>> pl.draw() #刷新界面
```

Python 的模块缓存功能虽然可以加速程序运行，但是也会带来一些问题。例如，如果有一个名为“mymodel.py”的模块，在“usemodel.py”中导入 mymodel 模块，那么只有在第一次运行“run usemodel.py”时会载入 mymodel 模块，以后即使“mymodel.py”文件发生改变，Python 也不会重新载入最新的模块。如果要修改调试“mymodel.py”中的程序，就需要在“usemodel.py”中添加如下两行代码：

```
import mymodel
reload(mymodel)
```

这样一来，每次运行“usemodel.py”时都会强制重新载入 mymodel 模块。



mymodel.py, usemodel.py
使用 reload 函数重新载入模块

IPython 还有很强大的调试功能，例如下面的程序使用 $\sin(x) \cos(x)$ 计算一个长度为 10000 的数组，并且调用 imshow() 将此数组显示成一个二维图像。



ipython_debug.py
用 IPython 调试程序中的错误

```
import pylab as pl
import numpy as np

def test_debug():
    x = np.linspace(1, 50, 10000)
    img = np.sin(x*np.cos(x))
    pl.imshow(img)
    pl.show()

test_debug()
```

但是由于程序中有错误，在 IPython 中运行它时，会出现很长一串错误信息。下面给出的是错误信息的最后一部分，我们看到抛出异常的是“image.pyc”中的 set_data() 函数，它是扩展模块中的一个函数，错误信息的意思是——作为图像数据的数组的维数不正确。

```
>>> run ipython_debug.py
[[省略]]
C:\Python26\lib\site-packages\matplotlib\image.pyc in set_data(self, A)
298         if (self._A.ndim not in (2, 3) or
299             (self._A.ndim == 3 and self._A.shape[-1] not in (3, 4))):
--> 300             raise TypeError("Invalid dimensions for image data")
301
302         self._imcache = None
TypeError: Invalid dimensions for image data
WARNING: Failure executing file: <ipython_debug.py>
```

为了找到程序中出错的位置，在 IPython 中输入 `debug` 命令，进入调试状态，并显示出调用堆栈的当前位置：

```
>>> debug
> c:\python26\lib\site-packages\matplotlib\image.py(300)set_data()
   299         (self._A.ndim == 3 and self._A.shape[-1] not in (3, 4)):
--> 300         raise TypeError("Invalid dimensions for image data")
   301
ipdb>
```

调试状态的提示符为“`ipdb`”，输入“`h`”命令可以查看调试状态下能用的所有命令，输入“`h 命令名`”可以查看命令的详细说明。连续执行多次“`u`”命令，沿着调用堆栈往上溯源，直到找到“`ipython_debug.py`”中出错的那一行：

```
ipdb> u
> c:\zhang\pydoc\source\examples\01-intro\ipython_debug.py(7)test_debug()
   6     img = np.sin(x*np.cos(x))
----> 7     pl.imshow(img)
   8     pl.show()
```

由错误信息可知数组 `img` 的维数不对。查看表示数组维数的 `ndim` 属性，发现 `img` 是一维数组，而 `imshow()` 的参数应该是二维数组：

```
ipdb> img.ndim
1
```

输入“`q`”命令结束调试，并编辑“`ipython_debug.py`”，在调用 `imshow()` 之前添加下面一行代码：

```
img.shape = 100, -1
```

然后再重新执行程序，这次就可以看到表示二维数组的图像了。

```
>>> run ipython_debug.py
```

1.3.2 Spyder

Spyder 是 Python(x,y) 的作者为它开发的一个简单的集成开发环境。和其他的 Python 开发环境相比，它最大的优点就是模仿 MATLAB 的“工作空间”的功能，可以很方便地观察和修改数组的值。图 1-5 是 Spyder 的界面截图。



<http://code.google.com/p/spyderlib>

Spyder 项目的地址

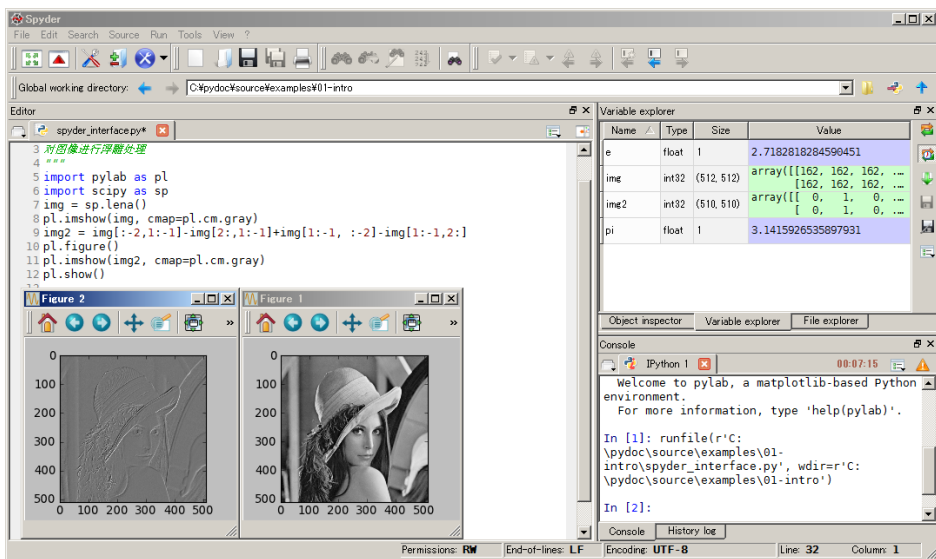


图 1-5 在 Spyder 中执行图像处理的程序

Spyder 的界面由许多窗格构成，用户可以根据自己的喜好调整它们的位置和大小。当多个窗格出现在一个区域时，将使用标签页的形式显示。例如在图 1-5 中，可以看到“Editor”、“Object inspector”、“Variable explorer”、“File explorer”、“Console”、“History log”以及两个显示图像的窗格。在 View 菜单中可以设置是否显示这些窗格。表 1-2 中列出了 Spyder 的主要窗格及其作用：

表 1-2 Spyder 的主要窗格及其作用

窗 格 名 称	作 用
Editor	编辑程序，可用标签页的形式编辑多个程序文件
Console	在别的进程中运行的 Python 控制台
Variable explorer	显示 Python 控制台中的变量列表
Object inspector	查看对象的说明文档和源程序
File explorer	文件浏览器，用于打开程序文件或者切换当前路径

按 F5 键将运行当前编辑器中的程序。第一次运行程序时，将弹出一个如图 1-6 所示的运行配置对话框。在此对话框中可以对程序的运行进行如下配置：

- **Command line options:** 输入程序的运行参数。
- **Working directory:** 输入程序的运行路径。
- **Execute in current Python or IPython interpreter:** 在当前的 Python 控制台中运行程序。程序可以访问此控制台中的所有全局对象，控制台中已经载入的模块不需要重新载入，因此程序的启动速度较快。
- **Execute in a new dedicated Python interpreter:** 新开一个 Python 控制台并在其中运行程序，程序的启动速度较慢，但是由于新控制台中没有多余的全局对象，因此更接近实际的运行情况。当选择此项时，还可以选中“Interact with the Python interpreter after

execution”复选框，这样当程序结束运行时，控制台进程将继续运行，因此可以通过它查看程序运行之后的所有全局对象。此外，还可以在“Command line options”中输入新控制台的启动参数。

运行配置对话框只会在第一次运行程序时出现，如果想修改程序的运行配置，可以按 F6 键打开运行配置对话框。

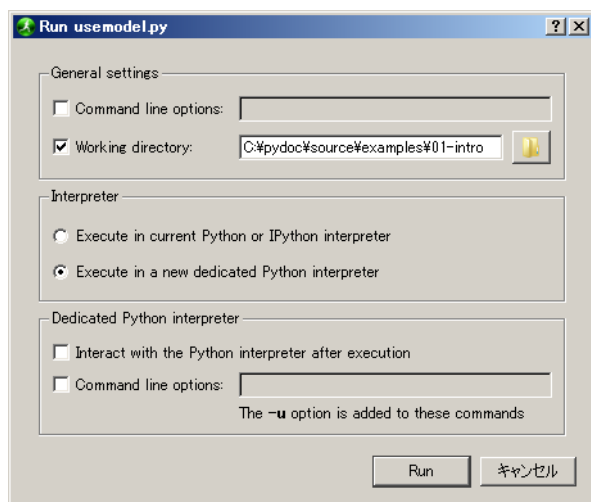


图 1-6 运行配置对话框

控制台中的全局对象可以在“Variable explorer”窗格中找到。此窗格支持数值、字符串、元组、列表、字典以及 NumPy 数组等对象的显示和编辑。图 1-7(左)是“Variable explorer”窗格的截图，其中列出了当前控制台中的变量名、类型、大小以及内容。右击变量名，弹出对此变量进行操作的菜单。在菜单中选择 Edit 选项，弹出图 1-7(右)所示的数组编辑窗口。此编辑窗口中，单元格的背景颜色直观地显示了数值的大小。



当有多个控制台运行时，“Variable explorer”窗格显示当前控制台中的全局对象。

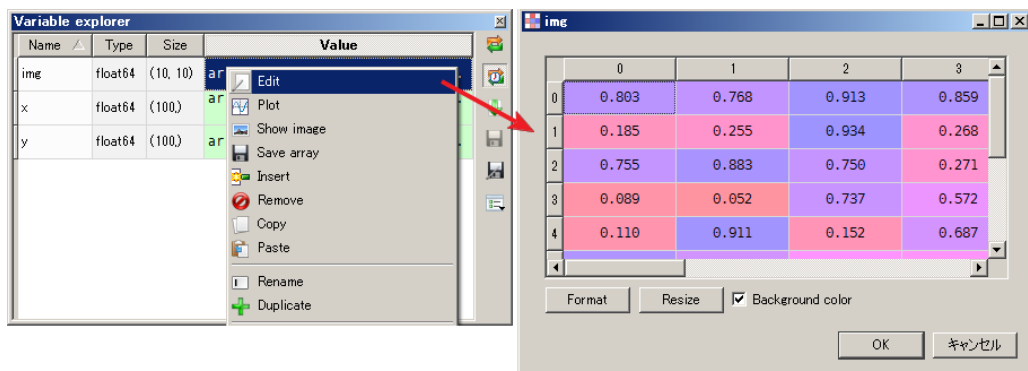


图 1-7 使用“Variable explorer”窗格查看和编辑数组的内容

选择菜单中的 Plot 选项，将弹出如图 1-8 所示的绘图窗口。在绘图窗口的右键菜单中选择“Parameters”，将弹出一个编辑绘图对象的对话框。图 1-8 中使用此对话框修改了曲线的颜色和线宽。

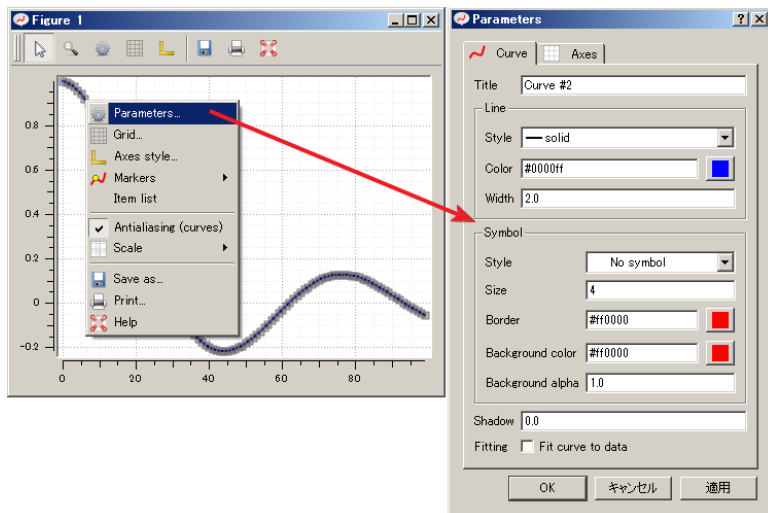


图 1-8 在“Variable explorer”窗格中将数组绘制成曲线

Spyder 的功能比较多，这里仅介绍一些常用的功能和技巧：

- 默认配置下，“Variable explorer”窗格中不显示以大写字母开头的变量，可以单击工具栏中的配置按钮(最后一个按钮)，在菜单中取消“Exclude capitalized references”的选中状态。
- 在控制台中，可以按 Tab 按键进行自动补全。在变量名之后输入“?”，可以在“Object inspector”窗格中查看对象的说明文档。此窗格的 Options 菜单中的“Show source”选项可以开启显示函数的源程序。
- 可以通过“Working directory”工具栏修改工作路径，用户程序运行时，将以此工作路径作为当前路径。例如我们只需要修改工作路径，就可以用同一个程序处理不同文件夹下的数据文件。
- 在程序编辑窗口中按住 Ctrl 键，并单击变量名、函数名、类名或模块名，可以快速跳转到定义位置。如果是在别的程序文件中定义的，将打开此文件。在学习一个新模块的用法时，我们经常需要查看模块中的某个函数或类是如何实现的，使用此功能可以帮助我们快速查看和分析各个模块的源程序。例如下面的程序从不同的扩展库载入了一些模块和类。用 Spyder 打开此文件，按住 Ctrl 键，并单击 signal、pl、HasTraits、Instance、View、Item、lfilter、plot、title 等，将打开定义它们的程序文件，并跳转到相应的行。



gotodefine.py

测试定义跳转功能


```
from scipy import signal
import pylab as pl
from enthought.traits.api import HasTraits, Instance
from enthought.traits.ui.api import View, Item

signal.lfilter
pl.plot
pl.title
```

1.3.3 Wing IDE 101

Wing IDE 是一个功能强大的 Python 集成开发环境，它的专业版是商用软件，但是也提供了一个免费的简装版本 Wing IDE 101。



<http://www.wingware.com/downloads/wingide-101>

Wing IDE 101 的下载地址

和 Spyder 一样，在 Wing IDE 中只需要按住 Ctrl 键并同时单击函数名或类名，就能直接跳转到定义它的位置。此外，Wing IDE 还有不错的调试功能。在程序中设置断点之后，单击 Debug 按钮就可以进入调试运行模式。当运行到断点之后，程序将暂停运行。读者可以用 Wing IDE 打开下面的程序，并将光标移到“self.count += 1”一行，按 F9 键添加断点，然后按 F5 键开始调试程序。



wingide_debug.py

测试 Wing IDE 的断点调试功能

图 1-9 是调试程序时的界面截图。程序执行之后会显示出一个窗口，其中有一个名为“Click Me”的按钮，单击它将调用程序中的_button_fired()，遇到断点从而暂停程序运行。此时可以观察程序的调用堆栈(Call stack)和堆栈数据(Stack Data)。

在主窗口左侧的“Stack Data”窗格中，显示了 locals 和 globals 两个字典，它们分别是当前执行环境下的全局变量和当前堆栈位置中的局部变量。下半部分显示了被选中的名为 self 的局部变量的内容。在主窗口下方的“Call Stack”窗格中显示了执行到断点处的调用堆栈，其中堆栈的顶部，即最下面一行被选中。可以用鼠标选中堆栈中的其他调用点，程序编辑窗格和“Stack Data”窗格中的内容也随之发生变化。通过这种方法可以观察堆栈中的所有局部变量，了解运行到断点处的整个调用过程，并查看与其相关的源程序。

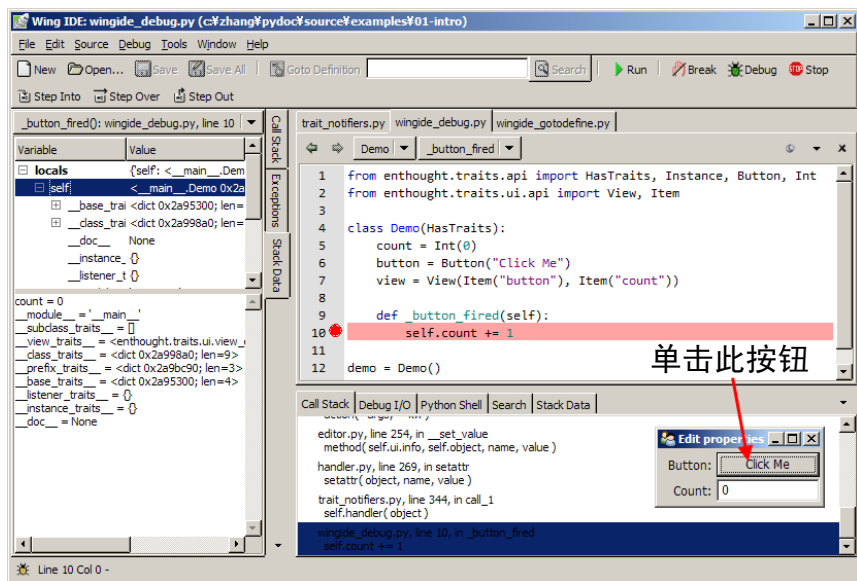


图 1-9 用 Wing IDE 101 调试程序

1.4 函数库介绍

Python 的科学计算功能由众多的扩展库协作完成。在本书的后续章节中，将对下列扩展库进行详细介绍。

1.4.1 数值计算库

NumPy 为 Python 带来了真正的多维数组功能，并且提供了十分丰富的对数组进行处理和运算的函数集。它对常用的数学函数进行数组化，使这些数学函数能直接对数组进行运算，将本来需要在 Python 中进行的循环运算，转移到高效率的库函数中，充分利用这些函数能明显地提高程序的运算速度。

SciPy 则在 NumPy 的基础之上添加了许多科学计算的函数库，其中一些函数是通过对久经考验的 Fortran 数值计算库进行封装实现的，例如：

- 线性代数使用 LAPACK 库
- 快速傅立叶变换使用 FFTPACK 库
- 常微分方程求解使用 ODEPACK 库
- 非线性方程组求解以及最小值求解等使用 MINPACK 库



<http://www.scipy.org>

SciPy 官方网址

有了这两个库, Python 就有几乎和 MATLAB 一样的数据处理能力了。此外, SciPy 中的 Weave 模块能在 Python 程序中直接嵌入 C++ 程序, 进一步提高程序的运算速度。

1.4.2 符号计算库

SymPy 是一套数学符号运算的扩展库, 虽然它目前还没有到达 1.0 版本, 但是已经足够好用, 可以帮助我们进行公式推导, 做一些简单的符号运算工作。



<http://code.google.com/p/sympy>

SymPy 官方网址

1.4.3 界面设计

Python 可以使用多种界面库编写 GUI 程序, 例如以 TK 为基础的 Tkinter、以 wxWidgets 为基础的 wxPython、以 QT 为基础的 PyQt4 等界面库。但是使用这些界面库编写 GUI 程序仍然是一件十分繁杂的工作。为了让读者不在界面设计上耗费大量精力, 从而能把注意力集中到数据处理上, 本书详细介绍了如何使用 Traits 库设计图形界面程序。



<http://code.enthought.com/projects/traits>

Traits 官方网址

Traits 库分为 Traits 和 TraitsUI 两大部分, 使用 Traits 能对 Python 对象的属性进行类型定义, 并为其添加初始化、校验、代理、事件处理等诸多功能。

TraitsUI 库基于 Traits 库, 使用 MVC(模型—视图—控制器)模式快速地定义用户界面, 在最简单的情况下, 甚至不需要写一句界面相关的代码, 就可以通过 Traits 的属性定义获得一个可用的图形界面。使用 TraitsUI 库编写的程序自动支持 wxPython 和 PyQt 界面库。

1.4.4 绘图与可视化

matplotlib 和 Chaco 是两个很优秀的二维绘图库。matplotlib 库能够快速地绘制精美的图表、以多种格式输出, 并且带有简单的三维绘图功能。而 Chaco 则以 Traits 为基础, 能够很方便地编写出交互式图表控件, 并嵌入到用 TraitsUI 编写的界面程序中。



<http://code.enthought.com/projects/chaco>

Chaco 官方网址



<http://matplotlib.sourceforge.net>

matplotlib 官方网址

TVTK 库对标准的 VTK 库用 Traits 进行了封装,如果要在 Python 中使用 VTK,用 TVTK 是最方便的选择。Mayavi 则在 TVTK 的基础上添加了一套面向应用的方便工具,它既可以单独作为三维可视化程序使用,也可以很方便地嵌入到用 TraitsUI 编写的界面程序中。



<http://code.enthought.com/projects/mayavi>

Mayavi 官方网址

VTK(Visualization Toolkit)

视觉化工具库(Visualization Toolkit, VTK)是一个开放源码、跨平台、支援平行处理(VTK 曾用于处理大小近乎 1 个 PB 的资料,其平台为美国 Los Alamos 国家实验室的具 1024 个处理器的大型系统)的图形应用函数库。2005 年曾被美国陆军研究实验室用于即时模拟俄罗斯制反导弹战车 ZSU23-4 受到平面波攻击的情形,其计算节点高达 25 万个之多。

此外,使用 VPython 库能够快速、方便地制作三维动画演示,使数据更有说服力。



<http://vpython.org>

VPython 官方网址

1.4.5 图像处理和计算机视觉

OpenCV 最初是由英特尔公司开发的一套开源的跨平台计算机视觉库,可用于开发实时的图像处理、计算机视觉以及模式识别程序。它有多套 Python 的调用接口,本书将以其中的 pyOpenCV 为例介绍 OpenCV 的一些基础知识。pyOpenCV 库不但很全面地对 OpenCV 的各种函数和类进行了封装,而且能在 OpenCV 的图像对象和 NumPy 数组之间进行互换。这样便同时扩展了 NumPy 的图像处理能力以及 OpenCV 的数组处理能力。



<http://code.google.com/p/pyopencv/>

pyopencv 项目的地址

NumPy——快速处理数据

标准的 Python 中用列表(list)保存一组值，可以当作数组使用。但由于列表的元素可以是任何对象，因此列表中保存的是对象的指针。这样一来，为了保存一个简单的列表[1,2,3]，就需要有三个指针和三个整数对象。对于数值运算来说，这种结构显然比较浪费内存和 CPU 计算时间。

此外 Python 还提供了 array 模块，它所提供的 array 对象和列表不同，能直接保存数值，和 C 语言的一维数组类似。但是由于它不支持多维数组，也没有各种运算函数，因此也不适合做数值运算。

NumPy 的诞生弥补了这些不足，NumPy 提供了两种基本的对象：ndarray^①和 ufunc^②。ndarray(下文统一称之为数组)是存储单一数据类型的多维数组，而 ufunc 则是能够对数组进行处理的函数。

2.1 ndarray 对象

函数库的导入

本书的示例程序假设用以下推荐的方式导入 NumPy 函数库：

```
import numpy as np
```

2.1.1 创建数组



NumPy 的函数和方法都有详细的说明文档和用法示例。在 IPython 中输入函数名并添加一个“?”符号，就可以显示文档内容。例如，输入“np.array?”可以查看 array() 的说明。

首先需要创建数组才能对其进行运算和操作。可以通过给 array()函数传递 Python 的序列对象来创建数组。如果传递的是多层嵌套的序列，将创建多维数组(下例中的变量 c)：



```
numpy_intro.py
```

① 英文全称为 n-dimensional array object。

② 英文全称为 universal function object。

NumPy 的基本使用方法

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([5, 6, 7, 8])
>>> c = np.array([[1, 2, 3, 4], [4, 5, 6, 7], [7, 8, 9, 10]])
>>> b
array([5, 6, 7, 8])
>>> c
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [7, 8, 9, 10]])
```

数组的形状可以通过其 `shape` 属性获得，它是一个描述数组各个轴长度的元组(tuple):

```
>>> a.shape
(4,)
>>> c.shape
(3, 4)
```

数组 `a` 的 `shape` 属性只有一个元素，因此它是一维数组。而数组 `c` 的 `shape` 属性有两个元素，因此它是二维数组，其中第 0 轴的长度为 3，第 1 轴的长度为 4。还可以通过修改数组的 `shape` 属性，在保持数组元素个数不变的情况下，改变数组每个轴的长度。下面的例子将数组 `c` 的 `shape` 属性改为(4,3)，注意：从(3,4)改为(4,3)并不是对数组进行转置，而只是改变每个轴的大小，数组元素在内存中的位置并没有改变。

```
>>> c.shape = 4,3
>>> c
array([[ 1,  2,  3],
       [ 4,  4,  5],
       [ 6,  7,  7],
       [ 8,  9, 10]])
```

当设置某个轴的元素个数为-1 时，将自动计算此轴的长度。由于数组 `c` 中有 12 个元素，因此下面的程序将数组 `c` 的 `shape` 属性改为了(2,6):

```
>>> c.shape = 2,-1
>>> c
array([[ 1,  2,  3,  4,  4,  5],
       [ 6,  7,  7,  8,  9, 10]])
```

使用数组的 `reshape()` 方法，可以创建指定形状的新数组，而原数组的形状保持不变:

```
>>> d = a.reshape((2,2)) # 也可以用 a.reshape(2,2)
```

```
>>> d
array([[1, 2],
       [3, 4]])
>>> a
array([1, 2, 3, 4])
```

数组 `a` 和 `d` 其实共享数据存储空间，因此修改其中任意一个数组的元素都会同时修改另外一个数组的内容：

```
>>> a[1] = 100 # 将数组 a 的第一个元素改为 100
>>> d # 注意数组 d 中的 2 也被改为了 100
array([[ 1, 100],
       [ 3,  4]])
```

数组的元素类型可以通过 `dtype` 属性获得。前面例子中，创建数组所用序列的元素都是整数，因此所创建的数组的元素类型是整型，并且是 32 bit 的长整型：

```
>>> c.dtype
dtype('int32')
```

可以通过 `dtype` 参数在创建数组时指定元素类型，注意 `float` 是 64 bit 的双精度浮点类型，而 `complex` 是 128 bit 的双精度复数类型：

```
>>> np.array([1, 2, 3, 4], dtype=np.float)
array([ 1.,  2.,  3.,  4.])
>>> np.array([1, 2, 3, 4], dtype=np.complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j])
```

NumPy 中的数据类型都有几种字符串表示方式，字符串和类型之间的对应关系都存储在 `typeDict` 字典中，例如 `'d'`、`'double'`、`'float64'` 都表示双精度浮点类型：

```
>>> np.typeDict["d"]
<type 'numpy.float64'>
>>> np.typeDict["double"]
<type 'numpy.float64'>
>>> np.typeDict["float64"]
<type 'numpy.float64'>
```

完整的类型列表可以通过下面的语句得到，它将 `typeDict` 字典中所有的值转换为一个集合，从而去除其中的重复项：

```
>>> set(np.typeDict.values())
set([<type 'numpy.bool_'> , <type 'numpy.int8'> , <type 'numpy.int16'>
     <type 'numpy.float32'> , <type 'numpy.uint8'> , <type 'numpy.complex128'>])
```

```
<type 'numpy.unicode_'> ,<type 'numpy.uint64'> ,<type 'numpy.int64'>
<type 'numpy.complex64'> ,<type 'numpy.string_'> ,<type 'numpy.uint32'>
<type 'numpy.void'> ,<type 'numpy.int32'> ,<type 'numpy.float96'>
<type 'numpy.object_'> ,<type 'numpy.uint32'> ,<type 'numpy.int32'>
<type 'numpy.float64'> ,<type 'numpy.complex192'>,<type 'numpy.uint16'> ])
```

前面的例子都是先创建一个 Python 的序列对象，然后通过 `array()` 将其转换为数组，这样做显然效率不高。因此 NumPy 提供了很多专门用于创建数组的函数。下面的每个函数都有一些关键字参数，具体用法请查看函数说明。

`arange()` 类似于内置函数 `range()`，通过指定开始值、终值和步长创建表示等差数列的一维数组，注意得到的结果数组不包含终值。例如下面的程序创建开始值为 0、终值为 1、步长为 0.1 的等差数组，注意终值 1 不在数组中：

```
>>> np.arange(0,1,0.1)
array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

`linspace()` 通过指定开始值、终值和元素个数创建表示等差数列的一维数组，可以通过 `endpoint` 参数指定是否包含终值，默认值为 `True`，即包含终值。下面两个例子分别演示了 `endpoint` 为 `True` 和 `False` 时的结果，注意 `endpoint` 的值会改变数组的等差步长：

```
>>> np.linspace(0, 1, 10) # 步长为 1/9
array([ 0. , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1. ])
>>> np.linspace(0, 1, 10, endpoint=False) # 步长为 1/10
array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

`logspace()` 和 `linspace()` 类似，不过它所创建的数组是等比数列。下面的例子产生从 10^0 到 10^2 、有 5 个元素的等比数列，注意起始值 0 表示 10^0 ，而终值 2 表示 10^2 ：

```
>>> np.logspace(0, 2, 5)
array([ 1. , 3.16227766, 10. , 31.6227766 , 100. ])
```

基数可以通过 `base` 参数指定，默认值为 10。下面通过将 `base` 参数设置为 2，并设置 `endpoint` 参数为 `False`，创建一个比例为 $2^{1/12}$ 等比数组^③：

```
>>> np.logspace(0, 1, 12, base=2, endpoint=False)
array([ 1. , 1.05946309, 1.12246205, 1.18920712, 1.25992105,
        1.33483985, 1.41421356, 1.49830708, 1.58740105, 1.68179283,
        1.78179744, 1.88774863])
```

`zeros()`、`ones()`、`empty()` 可以创建指定形状和类型的数组。其中：`empty()` 仅仅分配数组所

③ 此等比数组的比值是音乐中相差半音的两个音阶之间的频率比值，因此可以用它计算一个八度中所有半音的频率。

使用的内存，不对数组元素进行初始化操作，因此它的运行速度是最快的。下面的程序创建一个形状为(2,3)、元素类型为整数的数组：

```
>>> np.empty((2,3),np.int)    #只分配内存，不对其进行初始化
array([[ 32571594,   32635312,   505219724],
       [ 45001384,  1852386928,    665972]])
```

而 `zeros()` 则将数组元素初始化为 0，`ones()` 将数组元素初始化为 1。下面创建一个长度为 4、元素类型为浮点数的一维数组，并且元素全部初始化为 0：

```
>>> np.zeros(4, np.float)    #元素类型默认为 np.float，因此这里可以省略
array([ 0.,  0.,  0.,  0.]
```

此外，`zeros_like()`、`ones_like()`、`empty_like()` 等函数可创建与参数数组的形状及类型相同的数组。因此，“`zeros_like(a)`”和“`zeros(a.shape, a.dtype)`”的效果相同。

使用 `frombuffer()`、`fromstring()`、`fromfile()` 等函数可以从字节序列或文件创建数组，下面以 `fromstring()` 为例介绍它们的用法。先创建一个包含 8 个字符的字符串 `s`：

```
>>> s = "abcdefgh"
```

Python 的字符串实际上是一个字节序列，每个字符占一个字节，因此如果从字符串 `s` 创建一个 8 bit 的整数数组，所得到的数组正好就是字符串中每个字符的 ASCII 编码：

```
>>> np.fromstring(s, dtype=np.int8)
array([ 97,  98,  99, 100, 101, 102, 103, 104], dtype=int8)
```

如果从字符串 `s` 创建 16 bit 的整数数组，那么两个相邻的字节就表示一个整数，把字节 98 和字节 97 当作一个 16 位的整数，它的值就是 $98*256+97 = 25185$ 。可以看出，16 bit 的整数是以低位字节在前(little-endian)的方式保存在内存中的。

```
>>> np.fromstring(s, dtype=np.int16)
array([25185, 25699, 26213, 26727], dtype=int16)
>>> 98*256+97
25185
```

如果把整个字符串转换为一个 64 bit 的双精度浮点数数组，那么它的值是：

```
>>> np.fromstring(s, dtype=np.float)
array([ 8.54088322e+194])
```

显然这个结果没有什么意义，但是如果我们用 C 语言的二进制方式写了一组 `double` 类型的数值到某个文件中，那么就可以从此文件中读取相应的数据，并通过 `fromstring()` 将其转换为 `float64` 类型的数组。或者直接使用 `fromfile()` 从二进制文件中读取数据。

还可以先定义一个从下标计算数值的函数，然后用 `fromfunction()` 通过此函数创建数组：

```
>>> def func(i):
...     return i%4+1
...
>>> np.fromfunction(func, (10,))
array([ 1.,  2.,  3.,  4.,  1.,  2.,  3.,  4.,  1.,  2.])
```

`fromfunction()` 的第一个参数为计算每个数组元素的函数，第二个参数指定数组的形状。因为它支持多维数组，所以第二个参数必须是一个序列。上例中第二个参数是长度为 1 的元组 (10,)，因此创建了一个有 10 个元素的一维数组。

下面的例子创建一个表示九九乘法表的二维数组，输出的数组 `a` 中的每个元素 `a[i, j]` 都等于 `func2(i, j)`：

```
>>> def func2(i, j):
...     return (i+1) * (j+1)
...
>>> a = np.fromfunction(func2, (9,9))
>>> a
array([[ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [ 2.,  4.,  6.,  8., 10., 12., 14., 16., 18.],
       [ 3.,  6.,  9., 12., 15., 18., 21., 24., 27.],
       [ 4.,  8., 12., 16., 20., 24., 28., 32., 36.],
       [ 5., 10., 15., 20., 25., 30., 35., 40., 45.],
       [ 6., 12., 18., 24., 30., 36., 42., 48., 54.],
       [ 7., 14., 21., 28., 35., 42., 49., 56., 63.],
       [ 8., 16., 24., 32., 40., 48., 56., 64., 72.],
       [ 9., 18., 27., 36., 45., 54., 63., 72., 81.]])
```

2.1.2 存取元素



`numpy_access1d.py`
一维数组的元素存取

可以使用和列表相同的方式对数组的元素进行存取：

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[5]    # 用整数作为下标可以获取数组中的某个元素
5
>>> a[3:5]  # 用切片作为下标可以获取数组的一部分，包括 a[3]但不包括 a[5]
array([3, 4])
```

```
>>> a[:5] # 切片中省略开始下标, 表示从 a[0] 开始
array([0, 1, 2, 3, 4])
>>> a[:-1] # 下标可以使用负数, 表示从数组最后往前数
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> a[2:4] = 100, 101 # 下标还可以用来修改元素的值
>>> a
array([ 0,  1, 100, 101,  4,  5,  6,  7,  8,  9])
>>> a[1:-1:2] # 切片中的第三个参数表示步长, 2 表示隔一个元素获取一个元素
array([ 1, 101,  5,  7])
>>> a[::-1] # 省略切片的开始下标和结束下标, 步长为-1, 表示整个数组头尾颠倒
array([ 9,  8,  7,  6,  5,  4, 101, 100,  1,  0])
>>> a[5:1:-2] # 步长为负数时, 开始下标必须大于结束下标
array([ 5, 101])
```

和列表不同的是, 通过切片获取的新数组是原始数组的一个视图。它与原始数组共享同一块数据存储空间:

```
>>> b = a[3:7] # 通过切片产生一个新的数组 b, b 和 a 共享同一块数据存储空间
>>> b
array([101,  4,  5,  6])
>>> b[2] = -10 # 将 b 的第 2 个元素修改为-10
>>> b
array([101,  4, -10,  6])
>>> a # a 的第 5 个元素也被修改为-10
array([ 0,  1, 100, 101,  4, -10,  6,  7,  8,  9])
```

除了使用切片下标存取元素之外, NumPy 还提供了整数列表、整数数组和布尔数组等几种高级下标存取方法。

当使用整数列表对数组元素进行存取时, 将使用列表中的每个元素作为下标。使用列表作为下标得到的数组不和原始数组共享数据:

```
>>> x = np.arange(10, 1, -1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[[3, 3, 1, 8]] # 获取数组 x 中下标为 3、3、1、8 的 4 个元素, 组成一个新的数组
array([7, 7, 9, 2])
>>> b = x[[3, 3, -3, 8]] # 下标可以是负数
>>> b[2] = 100
>>> b
array([7, 7, 100, 2])
>>> x # 由于数组 b 和 x 不共享数据空间, 因此数组 x 中的值并没有改变
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[[3, 5, 1]] = -1, -2, -3 # 整数序列下标也可以用来修改元素的值
```

```
>>> x
array([10, -3, 8, -1, 6, -2, 4, 3, 2])
```

当使用整数数组作为数组下标时，将得到一个形状和下标数组相同的新数组，新数组的每个元素都是用下标数组中对应位置的值作为下标从原数组获得的值。当下标数组是一维时，结果和用列表作为下标的结果相同：

```
>>> x = np.arange(10,1,-1)
>>> x[np.array([3,3,1,8])]
array([7, 7, 9, 2])
```

而当下标是多维数组时，则得到的也是多维数组。我们可以将其理解为：先将下标数组展平为一维数组，并作为下标获得一个新的一维数组，然后再将其形状修改为下标数组的形状：

```
>>> x[np.array([[3,3,1,8],[3,3,-3,8]])]
array([[7, 7, 9, 2],
       [7, 7, 4, 2]])
>>> x[[3,3,1,8,3,3,-3,8]].reshape(2,4) # 改变数组形状
array([[7, 7, 9, 2],
       [7, 7, 4, 2]])
```

当使用布尔数组 **b** 作为下标存取数组 **x** 中的元素时，将收集数组 **x** 中所有在数组 **b** 中对应下标为 **True** 的元素。使用布尔数组作为下标获得的数组不和原始数组共享数据内存，注意这种方式只对应于布尔数组，不能使用布尔列表。

```
>>> x = np.arange(5,0,-1)
>>> x
array([5, 4, 3, 2, 1])
>>> # 布尔数组中下标为 0、2 的元素为 True，因此获取数组 x 中下标为 0、2 的元素
>>> x[np.array([True, False, True, False, False])]
array([5, 3])
>>> # 如果是布尔列表，则把 True 当作 1，False 当作 0，按照整数序列方式获取数组 x 中的元素
>>> x[[True, False, True, False, False]]
array([4, 5, 4, 5, 5])
>>> # 布尔数组的长度不够时，不够的部分都当作 False
>>> x[np.array([True, False, True, True])]
array([5, 3, 2])
>>> # 布尔数组下标也可以用来修改元素
>>> x[np.array([True, False, True, True])] = -1, -2, -3
>>> x
array([-1, 4, -2, -3, 1])
```

布尔数组一般不是手工产生，而是使用布尔运算的 **ufunc** 函数产生，关于 **ufunc** 函数请参照 2.2 节，下面我们举一个简单的例子说明布尔数组下标的用法：

```
>>> x = np.random.rand(10) # 产生一个长度为 10、元素值为 0 到 1 的随机数组
>>> x
array([ 0.72223939,  0.921226  ,  0.7770805 ,  0.2055047 ,  0.17567449,
        0.95799412,  0.12015178,  0.7627083 ,  0.43260184,  0.91379859])
>>> x>0.5
>>> # 数组 x 中的每个元素和 0.5 进行大小比较, 得到一个布尔数组, True 表示 x 中对应的值大于 0.5
array([ True,  True,  True, False, False,  True, False,  True, False,  True], dtype=bool)
>>> # 使用 x>0.5 得到的布尔数组收集 x 中的元素, 因此结果就是包含 x 中所有大于 0.5 的元素的数组
>>> x[x>0.5]
array([ 0.72223939,  0.921226  ,  0.7770805 ,  0.95799412,  0.7627083 ,  0.91379859])
```

2.1.3 多维数组

多维数组的存取和一维数组类似, 因为多维数组有多个轴, 因此它的下标需要用多个值表示。NumPy 采用元组(tuple)作为数组的下标, 元组中的每个元素和数组的每个轴对应。图 2-1 显示了一个形状为(6,6)的数组 a, 图中用不同颜色和线型标识出各个下标对应的选择区域。



numpy_access2d.py
多维数组的元素存取

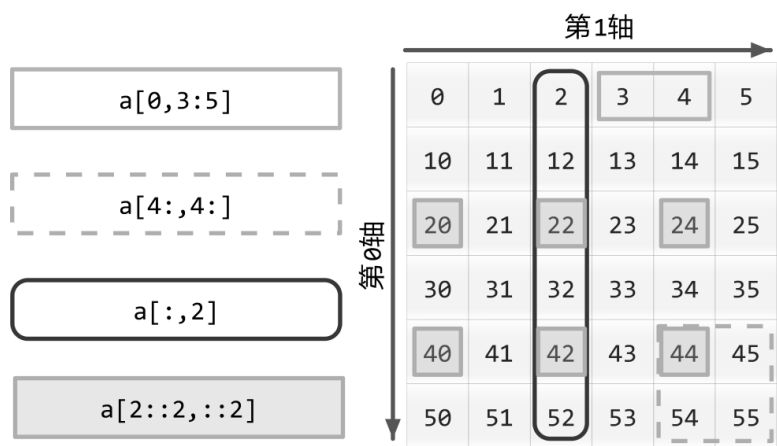


图 2-1 使用数组切片语法访问多维数组中的元素

为什么使用元组作为下标

Python 的下标语法(用[]存取序列中的元素)本身并不支持多维, 但是由于可以使用任何对象作为下标, 因此 NumPy 使用元组作为下标存取数组中的元素, 使用元组可以很方便地表示多个轴的下标。虽然在 Python 程序中, 经常用圆括号将元组的元素括起来, 但其实元组的语法只需要用逗号隔开元素即可, 例如“x,y=y,x”就是用元组交换变量值的一个例子。因此

`a[1,2]`和 `a[(1,2)]`完全相同，都是使用元组(1,2)作为数组 `a` 的下标。

读者也许会对如何创建图 2-1 中的二维数组感到好奇。它实际上是一个加法表，由纵向向量 (0, 10, 20, 30, 40, 50)和横向向量(0, 1, 2, 3, 4, 5)的元素相加而得。可以用下面的语句创建它，至于其原理，将在后面的章节进行介绍。

```
>>> a = np.arange(0, 60, 10).reshape(-1, 1) + np.arange(0, 6)
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

图 2-1 中的下标都是含有两个元素的元组，其中第 0 个元素与数组的第 0 轴(纵轴)对应，而第 1 个元素与数组的第 1 轴(横轴)对应。下面是图中各种多维数组切片的运算结果：

```
>>> a[0,3:5]
array([3, 4])
>>> a[4:,4:]
array([[44, 45],
       [54, 55]])
>>> a[:,2]
array([ 2, 12, 22, 32, 42, 52])
>>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

如果下标元组中只包含整数和切片，那么得到的数组和原始数组共享数据，它是原数组的视图。下面的例子中，数组 `b` 是 `a` 的视图，它们共享数据，因此修改 `b[0]`时，数组 `a` 中对应的元素也被修改：

```
>>> b = a[0,3:5]
>>> b[0] = -b[0]
>>> a[0, 3:5]
array([-3,  4])
```

因为数组的下标是一个元组，所以我们可以将下标元组保存起来，用同一个元组存取多个数组：

```
>>> idx = slice(None, None, 2), slice(2,None)
```

```
>>> a[idx] # 和 a[:,2,:]相同
array([[ 2, -3,  4,  5],
       [22, 23, 24, 25],
       [42, 43, 44, 45]])
>>> a[idx][idx] # 和 a[:,2:][::2,2:]相同
array([[ 4,  5],
       [44, 45]])
```

slice 对象

在[]中可以使用以冒号隔开的两个或三个整数表示切片，但是单独生成切片对象时需要使用 slice()创建。它有三个参数，分别为开始值、结束值和间隔步长，当这些值需要省略时可以使用 None。例如，a[slice(None,None,None),2]和 a[:,2]相同。

用 Python 内置的 slice()函数创建下标比较麻烦，因此 NumPy 提供了一个 s_对象来帮助我们创建数组下标：

```
>>> np.s_[:,2,:]
(slice(None, None, 2), slice(2, None, None))
```

请注意 s_实际上是一个 IndexExpression 类的对象：

```
>>> np.s_
<numpy.lib.index_tricks.IndexExpression object at 0x015093D0>
```

s_为什么不是函数

根据 Python 的语法，只有在方括号“[]”中才能使用以冒号隔开的切片语法，如果 s_是函数，那么这些切片必须使用 slice()创建。类似的对象还有 mgrid 和 ogrid 等，在后面的介绍中我们会学习它们的用法。Python 的下标语法实际上会调用__getitem__()方法，因此我们可以很容易自己实现 s_对象的功能：

```
>>> class S(object):
...     def __getitem__(self, index):
...         return index
>>> S()[:,2,:]
(slice(None, None, 2), slice(2, None, None))
```

在多维数组的下标元组中，也可以使用整数元组或列表、整数数组和布尔数组。当在下标中使用这些对象时，所获得的数据是原始数据的副本，因此修改结果数组不会改变原始数组。图 2-2 显示了如何使用各种序列下标存取多维数组。

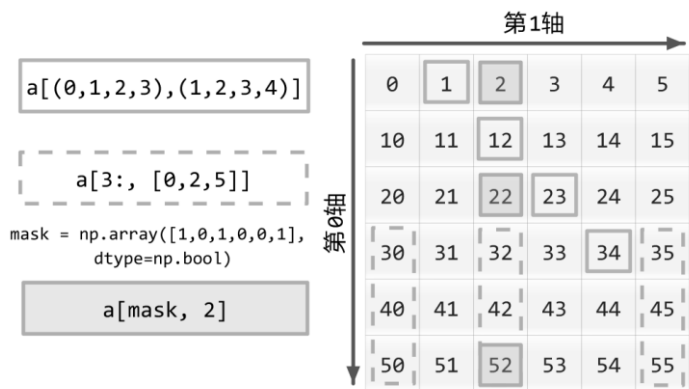


图 2-2 使用整数序列和布尔数组访问多维数组中的元素

```
>>> a[(0,1,2,3),(1,2,3,4)] ❶
array([ 1, 12, 23, 34])
>>> a[3:,[0,2,5]] ❷
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])
>>> mask = np.array([1,0,1,0,0,1], dtype=np.bool)
>>> a[mask, 2] ❸
array([ 2, 22, 52])
```

❶下标仍然是一个含有两个元素的元组，元组中的每个元素都是一个整数元组，分别对应数组的第0轴和第1轴。从两个序列的对应位置取出两个整数组成下标，于是得到的结果是： $a[0,1]$ 、 $a[1,2]$ 、 $a[2,3]$ 、 $a[3,4]$ 。

❷第0轴的下标是一个切片对象，它选取第3行之后的所有行；第1轴的下标是整数列表，它选取第0、2、5列。

❸第0轴的下标是一个布尔数组，它选取第0、2、5行；第1轴的下标是一个整数，选取第2列。注意如果 `mask` 不是布尔数组而是整数数组、列表或元组，就按照❶的方式进行运算：

```
>>> mask = np.array([1,0,1,0,0,1])
>>> a[mask, 2]
array([12, 2, 12, 2, 2, 12])
>>> mask = [True,False,True,False,False,True]
>>> a[mask, 2]
array([12, 2, 12, 2, 2, 12])
```

当下标的长度小于数组的维数时，则剩余的各轴所对应的下标是“:”，即选取它们的所有数据：

```
>>> a[[1,2]] #与 a[[1,2],:]相同
```



```
array([[10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25]])
```

当所有轴都用形状相同的整数数组作为下标时，得到的数组和下标数组的维数相同：

```
>>> x = np.array([[0,1],[2,3]])
>>> y = np.array([[-1,-2],[-3,-4]])
>>> a[x,y]
array([[ 5, 14],
       [23, 32]])
```

它的效果和下面程序的相同：

```
>>> a[(0,1,2,3),(-1,-2,-3,-4)].reshape(2,2)
array([[ 5, 14],
       [23, 32]])
```

当没有指定第 1 轴的下标时，则使用 “:” 作为其下标，因此得到了一个三维数组：

```
>>> a[x]
array([[[ 0, 1, 2, 3, 4, 5],
        [10, 11, 12, 13, 14, 15]],
       [[20, 21, 22, 23, 24, 25],
        [30, 31, 32, 33, 34, 35]]])
```

我们可以使用这种以整数数组为下标的方法来快速替换数组中的每个元素，例如有一个表示灰度图像的数组 `image` 以及一个调色板数组 `palette`，使用 “`palette[image]`” 可以得到通过调色板着色之后的彩色图像：

```
>>> palette = np.array( [ [0,0,0],      #调色板数组
...                       [255,0,0],
...                       [0,255,0],
...                       [0,0,255],
...                       [255,255,255] ] )
>>> image = np.array( [ [ 0, 1, 2, 0 ],   #图像数组
...                    [ 0, 3, 4, 0 ] ] )
>>> palette[image]
array([[[ 0, 0, 0],
        [255, 0, 0],
        [ 0, 255, 0],
        [ 0, 0, 0]],
       [[ 0, 0, 0],
        [ 0, 0, 255],
        [255, 255, 255],
        [ 0, 0, 0]]])
```

2.1.4 结构数组

在 C 语言中可以通过 `struct` 关键字定义结构类型，结构中的字段占据连续的内存空间。类型相同的两个结构体所占用的内存大小相同，因此可以很容易定义结构数组。和 C 语言一样，在 NumPy 中也很容易对这种结构数组进行操作。只要 NumPy 中的结构定义和 C 语言中的结构定义相同，就可以很方便地读取 C 语言的结构数组的二进制数据，将其转换为 NumPy 的结构数组。

假设需要定义一个结构数组，它的每个元素都有 `name`、`age` 和 `weight` 字段。在 NumPy 中可以如下定义：



`numpy_struct_array.py`

用 NumPy 将一个结构数组写入文件中

```
persontype = np.dtype({ ❶
    'names':['name', 'age', 'weight'],
    'formats':['S32','i', 'f']}, align= True )
a = np.array([("Zhang",32,75.5),("Wang",24,65.2)], ❷
    dtype=persontype)
```

❶先创建一个 `dtype` 对象 `persontype`，它的参数是一个描述结构类型的各个字段的字典。字典有两个键：`'names'`和`'formats'`。每个键对应的值都是一个列表。`'names'`定义结构中每个字段的名称，而`'formats'`则定义每个字段的类型。这里使用类型字符串定义字段类型：

- `'S32'`：长度为 32 字节的字符串类型，由于结构中每个元素的大小必须固定，因此需要指定字符串的长度。
- `'i'`：32 bit 的整数类型，相当于 `np.int32`。
- `'f'`：32 bit 的单精度浮点数类型，相当于 `np.float32`。

❷然后调用 `array()`创建数组，通过 `dtype` 参数指定所创建数组的元素类型为 `persontype`。运行上面程序之后，可以在 IPython 中执行如下语句来查看数组 `a` 的元素类型：

```
>>> run numpy_struct_array.py
>>> a.dtype
dtype([('name', '|S32'), ('age', '<i4'), ('weight', '<f4')])
```

这里我们看到了另外一种描述结构类型的方法：一个包含多个元组的列表，其中形如(字段名, 类型描述)的元组描述了结构中的每个字段。类型字符串前面的`|`、`<`、`>`等字符表示字节值的字节顺序：

- `|`：忽视字节顺序
- `<`：低位字节在前，即小端模式(little endian)
- `>`：高位字节在前，即大端模式(big endian)

结构数组的存取方式和一般数组相同，通过下标能够取得其中的元素，注意元素的值看上去像是元组，实际上它是一个结构：

```
>>> a[0]
('Zhang', 32, 75.5)
>>> a[0].dtype
dtype([('name', '<S32'), ('age', '<i4'), ('weight', '<f4')])
```

可以使用字段名作为下标获取对应的字段值：

```
>>> a[0]["name"]
'Zhang'
```

`a[0]`是一个结构元素，它和数组 `a` 共享内存数据，因此可以通过修改它的字段，改变原始数组中对应元素的字段：

```
>>> c = a[1]
>>> c["name"] = "Li"
>>> a[1]["name"]
"Li"
```

不但可以获得结构元素的某个字段，而且可以直接获得结构数组的字段，它返回的是原始数组的视图，因此下面的程序可以通过修改 `b[0]` 改变 `a[0]["age"]`：

```
>>> b=a["age"]
>>> b
array([32, 24])
>>> b[0] = 40
>>> a[0]["age"]
40
```

通过 `a.tostring()` 或 `a.tofile()` 方法，可以将数组 `a` 以二进制的方式转换成字符串或者写入文件中：

```
>>> a.tofile("test.bin")
```

利用下面的 C 语言程序可以将 “test.bin” 文件中的数据读取出来。



read_struct_array.c

用 C 语言读取 NumPy 输出的结构数组文件

```
#include <stdio.h>

struct person
{
```

```

    char name[32];
    int age;
    float weight;
};

struct person p[3];

void main ()
{
    FILE *fp;
    int i;
    fp=fopen("test.bin","rb");
    fread(p, sizeof(struct person), 2, fp);
    fclose(fp);
    for(i=0;i<2;i++)
    {
        printf("%s %d %f\n", p[i].name, p[i].age, p[i].weight);
    }
}

```

内存对齐

为了内存寻址方便，C 语言的结构体会自动添加一些填充用的字节，这叫内存对齐。例如，如果把上面 C 语言所定义结构体中的字段 `name[32]` 改为 `name[30]`，由于内存对齐问题，在 `name` 和 `age` 中间会填补两个字节，最终的结构体的大小不会发生改变。因此如果数组中配置的内存大小不符合 C 语言的对齐规范，将会出现数据错位。为了解决这个问题，在创建 `dtype` 对象时，可以传递参数 `align=True`，这样结构数组的内存对齐就和 C 语言的结构体一致了。

结构类型中可以包括其他的结构类型，下面的语句创建一个含有一个字段 `f1` 的结构，`f1` 的值是另外一个结构，它含有字段 `f2`，其类型为 16 bit 整数。

```

>>> np.dtype([('f1', [( 'f2', np.int16)])])
dtype([('f1', [( 'f2', '<i2')])])

```

当某个字段的类型为数组时，用元组的第三个参数表示其形状。在下面的结构体中，`f1` 字段是一个形状为(2,3)的双精度浮点数组：

```

>>> np.dtype([('f0', 'i4'), ('f1', 'f8', (2, 3))])
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])

```

用下面的字典参数也可以定义结构类型，字典的键为结构中的字段名，值为字段的类型描述，但是由于字典的键是没有顺序的，因此字段的顺序需要在类型描述中给出。类型描述是一个元组，它的第二个值给出字段的以字节为单位的偏移量，例如下例中的 `age` 字段的偏移量为

25 个字节:

```
>>> np.dtype({'surname':('S25',0),'age':(np.uint8,25)})
dtype([('surname', '<S25'), ('age', '<u1')])
```

使用字段的地址偏移量可以定义内存地址不连续的字段,在读取复杂的 C 语言结构体中的部分字段时这种方式十分有效。

2.1.5 内存结构

下面让我们看看数组对象是如何在内存中存储的。如图 2-3 所示,数组的描述信息保存在一个数据结构中,这个结构引用两个对象:用于保存数据的存储区域和用于描述元素类型的 dtype 对象。

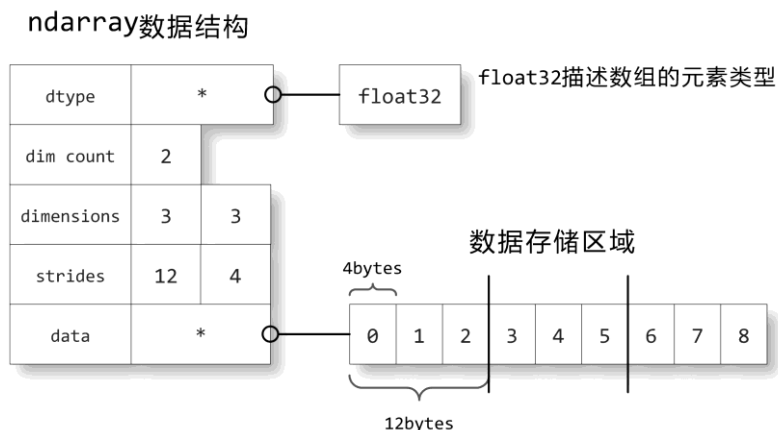


图 2-3 ndarray 数组对象在内存中的存储方式

数据存储区域保存着数组中所有元素的二进制数据, dtype 对象则知道如何将元素的二进制数据转换为可用的值。数组的维数和形状等信息都保存在 ndarray 数组对象的数据结构中。图 2-3 中显示的是下面的数组 a 的内存结构:

```
>>> a = np.array([[0,1,2],[3,4,5],[6,7,8]], dtype=np.float32)
```

数组对象使用其 strides 属性保存每个轴上相邻两个元素的地址差,即当某个轴的下标增加 1 时,数据存储区域指针所增加的字节数。例如图 2-3 中的 strides 为(12,4),即第 0 轴的下标增加 1 时,数据的地址增加 12 个字节。也就是 a[1,0]的地址比 a[0,0]的地址大 12,正好是 3 个单精度浮点数的总字节数。第 1 轴下标增加 1 时,数据的地址增加 4 个字节,正好是一个单精度浮点数的字节数。

如果 strides 属性中的数值正好和对应轴所占据的字节数相同,那么数据在内存中是连续存储的。通过切片下标得到的新数组是原始数组的视图,即它和原始数组共享数据存储区域,但是新数组的 strides 属性和 data 属性会发生变化:

```
>>> b = a[:,2,::2]
>>> b
array([[ 0.,  2.],
       [ 6.,  8.]], dtype=float32)
>>> b.strides
(24, 8)
```

由于数组 **b** 和数组 **a** 共享数据存储区域，而数组 **b** 中的第 0 轴和第 1 轴的元素都是从数组 **a** 中隔一个元素取一个，因此数组 **b** 的 **strides** 变成了(24,8)，正好都是数组 **a** 的两倍。对照前面的图 2-3 很容易看出：数据 0 和 2 的地址相差 8 个字节，而数据 0 和 6 的地址相差 24 个字节。

元素在数据存储区域的排列格式有两种：**C** 语言格式和 **Fortan** 语言格式。在 **C** 语言中，多维数组的第 0 轴是最上位的，即第 0 轴的下标增加 1 时，元素的地址增加的字节数最多。而 **Fortan** 语言中的多维数组的第 0 轴是最下位的，即第 0 轴的下标增加 1 时，地址只增加一个元素的字节数。在 **NumPy** 中，默认以 **C** 语言格式存储数据，如果希望改为 **Fortan** 格式，只需要在创建数组时，设置 **order** 参数为"**F**"即可：

```
>>> c = np.array([[0,1,2],[3,4,5],[6,7,8]], dtype=np.float32, order="F")
>>> c.strides
(4, 12)
```

了解了数组的内存结构后，就可以解释使用数组下标获取数据时的复制和引用问题：

- 当下标使用整数和切片时，获取的数据在数据存储区域中是等间隔分布的。由于只需要修改图 2-3 所示数据结构中的 **dim count**、**dimensions**、**stride** 等属性以及指向数据存储区域的指针 **data**，就能实现整数和切片下标，因此新数组和原始数组能够共享数据存储区域。
- 当使用整数序列、整数数组和布尔数组时，不能保证获取的数据在数据存储区域是等间隔的，因此无法和原始数组共享数据，只能对数据进行复制。

数组的 **flags** 属性描述了数据存储区域的一些属性，直接查看 **flags** 属性将输出各个标识的值：

```
>>> a.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

也可以单独获得其中的某个标识的值：

```
>>> a.flags.c_contiguous
True
```

几个比较重要的标识的含义如下：

- **C_CONTIGUOUS**：数据存储区域是否是 C 语言格式连续区域。
- **F_CONTIGUOUS**：数据存储区域是否是 Fortran 语言格式连续区域。
- **OWNDATA**：数组是否拥有此数据存储区域，当数组是其他数组的视图时，它不拥有数据存储区域。

由于数组 **a** 是通过 `array()` 直接创建的，因此它的数据存储区域是 C 语言格式连续区域，并且它拥有数据存储区域。下面我们看看数组 **a** 的转置数组的标识：

```
>>> a.T.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : False
```

数组的转置可以通过其 **T** 属性获得，转置数组将其数据存储区域看作 Fortran 语言格式连续区域，并且它不拥有数据存储区域。

```
>>> b.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : False
OWNDATA : False
```

由于数组 **b** 是数组 **a** 的一个视图，因此它既不拥有数据存储区域，它的数据也不是连续存储的。通过视图数组的 `base` 属性可以获得保存数据的原始数组：

```
>>> id(b.base)
34064760
>>> id(a)
34064760
```

除了使用切片从同一块数据区域创建不同的 `shape` 和 `strides` 的数组对象之外，我们还可以直接设置这些属性，从而得到用切片实现不了的效果，例如：

```
>>> from numpy.lib.stride_tricks import as_strided
>>> a = np.arange(6)
>>> b = as_strided(a, shape=(4, 3), strides=(4, 4))
>>> b
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
```



使用 `as_strided()` 时 NumPy 不会进行内存越界检查，因此如果 `shape` 和 `strides` 设置不当，就可能会引起意想不到的错误。

在上面这个例子中，我们从 NumPy 的辅助模块中载入了一个 `as_strided()` 函数，并使用它从一个长度为 6 的一维数组 `a`，创建了一个 `shape` 为 (4,3) 的二维数组 `b`。由于通过 `strides` 参数直接指定了数组 `b` 的 `strides` 属性，因此不仅数组 `b` 和数组 `a` 共享数据区域，而且数组 `b` 中前后两行有两个元素是重合的。例如下面修改 `a[2]` 的值，数组 `b` 中前三行中对应的元素也会发生改变：

```
>>> a[2] = 20
>>> b
array([[ 0,  1, 20],
       [ 1, 20,  3],
       [20,  3,  4],
       [ 3,  4,  5]])
```

在对数据进行处理时，可能经常需要对数据进行分块处理，而且为了保持平滑，每块数据之间需要有一定的重叠部分。这时可以使用上面介绍的方法对数据进行带重叠的分块。

2.2 ufunc 运算

`ufunc` 是 `universal function` 的缩写，它是一种能对数组中每个元素进行操作的函数。NumPy 内置的许多 `ufunc` 函数都是在 C 语言级别实现的，因此它们的计算速度非常快。让我们先看一个例子：

```
>>> x = np.linspace(0, 2*np.pi, 10)
>>> y = np.sin(x)
>>> y
array([ 0.00000000e+00,  6.42787610e-01,  9.84807753e-01,
        8.66025404e-01,  3.42020143e-01, -3.42020143e-01,
       -8.66025404e-01, -9.84807753e-01, -6.42787610e-01,
       -2.44921271e-16])
```

先用 `linspace()` 产生一个从 0 到 2π 的等差数组，然后将其传递给 `np.sin()` 函数计算每个元素的正弦值。由于 `np.sin()` 是一个 `ufunc` 函数，因此在其内部对数组 `x` 的每个元素进行循环，分别计算它们的正弦值，并返回一个保存各个计算结果的数组。运算之后数组 `x` 中的值并没有改变，而是新建了一个数组来保存结果。也可以通过 `out` 参数指定计算结果的保存位置。因此如果希望直接在数组 `x` 中保存结果，可以将它传递给 `out` 参数：

```
>>> t = np.sin(x,out=x)
>>> x
array([ 0.00000000e+00,  6.42787610e-01,  9.84807753e-01,
        8.66025404e-01,  3.42020143e-01, -3.42020143e-01,
       -8.66025404e-01, -9.84807753e-01, -6.42787610e-01,
```



```
-2.44921271e-16])
>>> id(t) == id(x)
True
```

ufunc 函数的返回值仍然是计算的结果，只不过它就是数组 x，因此两个数组的 id 是相同的。下面比较 np.sin 和 Python 标准库中 math.sin 的计算速度：



numpy_speed_test.py

NumPy 计算和 Python 标准库的计算速度比较

```
import time
import math
import numpy as np

x = [i * 0.001 for i in xrange(1000000)]
start = time.clock()
for i, t in enumerate(x):
    x[i] = math.sin(t)
print "math.sin:", time.clock() - start

x = [i * 0.001 for i in xrange(1000000)]
x = np.array(x)
start = time.clock()
np.sin(x,x)
print "numpy.sin:", time.clock() - start

x = [i * 0.001 for i in xrange(1000000)]
start = time.clock()
for i, t in enumerate(x):
    x[i] = np.sin(t)
print "numpy.sin loop:", time.clock() - start
```

程序的输出为：

```
math.sin: 0.779217749742
numpy.sin: 0.0772958574344
numpy.sin loop: 5.25540843878
```

可以看出 np.sin 比 math.sin 快 10 倍多，这得益于 np.sin 在 C 语言级别的循环计算。

列表推导式比循环更快

事实上，标准 Python 中有比 for 循环更快的方案——使用列表推导式。但是列表推导式将产生一个新的列表，而不是直接修改原来列表中的元素。下面的语句执行时，将计算出一

个新的列表来保存每个正弦值:

```
>>> x = [math.sin(t) for t in x]
```

`np.sin` 同样也支持计算单个数值的正弦值。不过值得注意的是, 对单个数值的计算 `math.sin` 要比 `np.sin` 快很多。在 Python 级别进行循环时, `np.sin` 的计算速度只有 `math.sin` 的 1/6。这是因为 `np.sin` 为了同时支持数组和单个数值的计算, 其 C 语言的内部实现要比 `math.sin` 复杂得多。此外, 对于单个数值的计算, `np.sin` 的返回值类型和 `math.sin` 的不同, `math.sin` 返回的是 Python 的标准 `float` 类型, 而 `np.sin` 返回的是 `float64` 类型:

```
>>> type(math.sin(0.5))
<type 'float'>
>>> type(np.sin(0.5))
<type 'numpy.float64'>
```

通过下标获取的数组元素的类型为 NumPy 中定义的类型。将其转换为 Python 的标准类型还需要花费额外的时间。为了解决这个问题, 数组提供了 `item()` 方法, 用来获取数组中的单个元素, 并直接返回标准的 Python 数值类型:

```
>>> a = np.arange(6.0).reshape(2,3)
>>> a.item(1,2) # 和 a[1,2]类似
5.0
>>> type(a.item(1,2)) # item()返回的是 Python 的标准 float 类型
<type 'float'>
>>> type(a[1,2]) # 下标方式返回的是 NumPy 的 float64 类型
<type 'numpy.float64'>
```

通过上面的例子我们了解了如何最有效率地使用 `math` 库和 NumPy 中的数学函数。由于它们各有优缺点, 因此在导入时不建议使用 “`import *`” 全部载入, 而是应该使用 “`import numpy as np`” 载入, 这样可以根据需要选择合适的函数。

2.2.1 四则运算

NumPy 提供许多 `ufunc` 函数, 例如计算两个数组之和的 `add()` 函数:

```
>>> a = np.arange(0,4)
>>> a
array([0, 1, 2, 3])
>>> b = np.arange(1,5)
>>> b
array([1, 2, 3, 4])
>>> np.add(a,b)
array([1, 3, 5, 7])
```

```
>>> np.add(a,b,a)
array([1, 3, 5, 7])
>>> a
array([1, 3, 5, 7])
```

`add()` 返回一个数组，数组的每个元素都是两个参数数组中对应元素之和。如果没有指定 `out` 参数，那么将创建一个新的数组来保存计算结果。如果指定了第三个参数 `out`，就不产生新的数组，而是直接将结果保存到指定的数组中。

NumPy 为数组定义了各种数学运算操作符，因此计算两个数组相加可以简单地写为 `a+b`，而 `np.add(a,b,a)` 则可以用 `a+=b` 表示。表 2-1 列出了数组的运算符及其对应的 `ufunc` 函数，注意除号 `/` 的意义根据是否激活 `__future__.division` 会有所不同。

表 2-1 数组的运算符及其对应的 `ufunc` 函数

运 算 符	对应的 <code>ufunc</code> 函数
<code>y = x1 + x2</code>	<code>add(x1, x2 [, y])</code>
<code>y = x1 - x2</code>	<code>subtract(x1, x2 [, y])</code>
<code>y = x1 * x2</code>	<code>multiply(x1, x2 [, y])</code>
<code>y = x1 / x2</code>	<code>divide(x1, x2 [, y])</code> , 如果两个数组的元素为整数，那么用整数除法
<code>y = x1 / x2</code>	<code>true_divide(x1, x2 [, y])</code> , 总是返回精确的商
<code>y = x1 // x2</code>	<code>floor_divide(x1, x2 [, y])</code> , 总是对返回值取整
<code>y = -x</code>	<code>negative(x [, y])</code>
<code>y = x1 ** x2</code>	<code>power(x1, x2 [, y])</code>
<code>y = x1 % x2</code>	<code>remainder(x1, x2 [, y])</code> , 或 <code>mod(x1, x2, [, y])</code>

数组对象支持操作符，极大地简化了表达式的编写。不过要注意：如果表达式很复杂，并且要运算的数组很大，将会因为产生大量的中间结果而降低程序的运算效率。例如，假设对 `a`、`b`、`c` 三个数组采用表达式 “`x=a*b+c`” 进行计算，那么这个表达式相当于：

```
t = a * b
x = t + c
del t
```

也就是说，需要产生一个临时数组 `t` 来保存乘法的运算结果，然后再产生最后的结果数组 `x`。可以将表达式分解为下面的两行语句，以减少一次内存分配：

```
x = a*b
x += c
```

2.2.2 比较和布尔运算

使用“==”、“>”等比较运算符对两个数组进行比较，将返回一个布尔数组，它的每个元素值都是两个数组对应元素的比较结果。例如：

```
>>> np.array([1,2,3]) < np.array([3,2,1])
array([ True, False, False], dtype=bool)
```

每个比较运算符都与一个 ufunc 函数对应，下面是比较运算符与其 ufunc 函数的对照表：

表 2-2 比较运算符及其对应的 ufunc 函数

比较运算符	ufunc 函数
$y = x1 == x2$	<code>equal(x1, x2 [, y])</code>
$y = x1 != x2$	<code>not_equal(x1, x2 [, y])</code>
$y = x1 < x2$	<code>less(x1, x2, [, y])</code>
$y = x1 <= x2$	<code>less_equal(x1, x2, [, y])</code>
$y = x1 > x2$	<code>greater(x1, x2, [, y])</code>
$y = x1 >= x2$	<code>greater_equal(x1, x2, [, y])</code>

由于 Python 中的布尔运算使用 `and`、`or` 和 `not` 等关键字，它们无法被重载，因此数组的布尔运算只能通过相应的 ufunc 函数进行。这些函数名都以“logical_”开头，在 IPython 中使用自动补全功能可以很容易地找到它们：

```
>>> np.logical # 按 Tab 键进行自动补全
np.logical_and np.logical_not np.logical_or np.logical_xor
```

下面是一个使用 `logical_or()` 进行或运算的例子：

```
>>> a = np.arange(5)
>>> b = np.arange(4,-1,-1)
>>> a == b
array([False, False,  True, False, False], dtype=bool)
>>> a > b
array([False, False, False,  True,  True], dtype=bool)
>>> np.logical_or(a==b, a>b) # 和 a>=b 相同
array([False, False,  True,  True,  True], dtype=bool)
```

对两个布尔数组使用 `and`、`or` 和 `not` 等进行布尔运算，将抛出 `ValueError` 异常。因为布尔数组中有 `True` 也有 `False`，所以 NumPy 无法确定用户的运算目的：

```
>>> a==b and a>b
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

错误信息告诉我们可以使用数组的 `any()` 或 `all()` 方法^④。只要数组中有一个值为 `True`，`any()` 就返回 `True`；而只有当数组的全部元素都为 `True` 时，`all()` 才返回 `True`。

```
>>> np.any(a==b)
True
>>> np.any(a==b) and np.any(a>b)
True
```

以 “bitwise_” 开头的函数是比特运算函数，包括 `bitwise_and`、`bitwise_not`、`bitwise_or` 和 `bitwise_xor` 等。也可以使用 “&”、“~”、“|” 和 “^” 等操作符进行计算。

对于布尔数组来说，位运算和布尔运算的结果相同。但在使用时要注意，位运算符的优先级比比较运算符高，因此需要使用括号提高比较运算的运算优先级。例如：

```
>>> (a==b) | (a>b)
array([False, False, True, True, True], dtype=bool)
```

整数数组的位运算和 C 语言的位运算相同，在使用时要注意元素类型的符号，例如下面的 `arange()` 所创建的数组的元素类型为 32 位符号整数，因此对正数按位取反将得到负数。以整数 0 为例，按位取反的结果是 `0xFFFFFFFF`，在 32 位符号整数中，这个值表示 -1。

```
>>> ~np.arange(5)
array([-1, -2, -3, -4, -5])
```

而如果对 8 位无符号整数数组进行位取反运算：

```
>>> ~np.arange(5, dtype=np.uint8)
array([255, 254, 253, 252, 251], dtype=uint8)
```

同样的整数 0，按位取反的结果是 `0xFF`，当它是 8 位无符号整数时，它的值是 255。

2.2.3 自定义 ufunc 函数

通过 NumPy 提供的标准 `ufunc` 函数，可以组合出复杂的表达式，在 C 语言级别对数组的每个元素进行计算。但有时这种表达式不易编写，而对每个元素进行计算的程序却很容易用 Python 实现，这时可以用 `frompyfunc()` 将一个计算单个元素的函数转换成 `ufunc` 函数。这样就可以方便地用所产生的 `ufunc` 函数对数组进行计算了。

例如，我们可以用一个分段函数描述三角波，三角波的样子如图 2-4 所示，它分为三段：上升段、下降段和平坦段。

^④ 在 NumPy 中同时也定义了 `any()` 和 `all()` 函数。

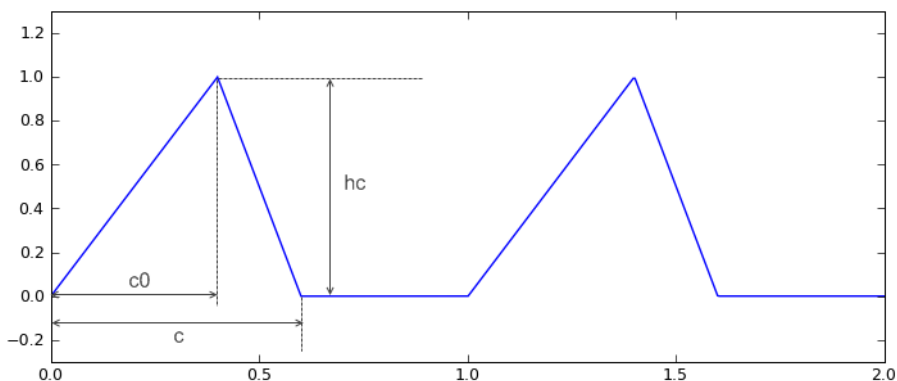


图 2-4 三角波可以用分段函数进行计算



numpy_frompyfunc.py

用 frompyfunc() 计算三角波的波形数组

根据图 2-4，我们很容易写出计算三角波上某点 Y 坐标的函数。显然 `triangle_wave()` 只能计算单个数值，不能对数组直接进行处理。

```
def triangle_wave(x, c, c0, hc):
    x = x - int(x) # 三角波的周期为 1，因此只取 x 坐标的小数部分进行计算
    if x >= c: r = 0.0
    elif x < c0: r = x / c0 * hc
    else: r = (c-x) / (c-c0) * hc
    return r
```

可以用下面的程序先使用列表推导式计算出一个列表，然后用 `array()` 将列表转换为数组。这种做法每次都需要使用列表推导式语法调用函数，对于多维数组是很麻烦的。

```
x = np.linspace(0, 2, 1000)
y1 = np.array([triangle_wave(t, 0.6, 0.4, 1.0) for t in x])
```

通过 `frompyfunc()` 可以将计算单个值的函数转换为一个能对数组中每个元素进行计算的 `ufunc` 函数。`frompyfunc()` 的调用格式为：

```
frompyfunc(func, nin, nout)
```

其中，`func` 是计算单个元素的函数，`nin` 是 `func` 输入参数的个数，`nout` 是 `func` 返回值的个数。下面的程序使用 `frompyfunc()` 将 `triangle_wave()` 转换为一个 `ufunc` 函数对象 `triangle_ufunc1`：

```
triangle_ufunc1 = np.frompyfunc(triangle_wave, 4, 1)
y2 = triangle_ufunc1(x, 0.6, 0.4, 1.0)
```

值得注意的是, `triangle_ufunc1()` 所返回数组的元素类型是 `object`, 因此还需要再调用数组的 `astype()` 方法以将其转换为双精度浮点数组:

```
>>> run numpy_frompyfunc.py # 在 IPython 中运行计算三角波的程序
>>> y2.dtype
dtype('object')
>>> y2 = y2.astype(np.float)
>>> y2.dtype
dtype('float64')
```

使用 `vectorize()` 可以实现和 `frompyfunc()` 类似的功能, 但它可以通过 `otypes` 参数指定返回数组的元素类型。 `otypes` 参数可以是一个表示元素类型的字符串, 也可以是一个类型列表, 使用列表可以描述多个返回数组的元素类型。下面的程序使用 `vectorize()` 计算三角波:

```
triangle_ufunc2 = np.vectorize(triangle_wave, otypes=[np.float])
y3 = triangle_ufunc2(x, 0.6, 0.4, 1.0)
```

最后我们验证一下结果:

```
>>> np.all(y1==y2)
True
>>> np.all(y2==y3)
True
```

2.2.4 广播

当使用 `ufunc` 函数对两个数组进行计算时, `ufunc` 函数会对这两个数组的对应元素进行计算, 因此要求这两个数组的形状相同。如果形状不同, 会进行如下的广播(broadcasting)处理:

(1) 让所有输入数组都向其中维数最多的数组看齐, `shape` 属性中不足的部分都通过在前面加 1 补齐。

(2) 输出数组的 `shape` 属性是输入数组的 `shape` 属性在各个轴上的最大值。

(3) 如果输入数组的某个轴长度为 1 或与输出数组对应轴的长度相同, 这个数组就能够用来计算, 否则出错。

(4) 当输入数组的某个轴长度为 1 时, 沿着此轴运算时都用此轴上的第一组值。

上述 4 条规则理解起来可能比较费劲, 下面让我们看一个实际的例子。

先创建一个二维数组 `a`, 其形状为 (6,1):

```
>>> a = np.arange(0, 60, 10).reshape(-1, 1)
>>> a
array([[ 0], [10], [20], [30], [40], [50]])
>>> a.shape
(6, 1)
```

再创建一维数组 **b**，其形状为(5,):

```
>>> b = np.arange(0, 5)
>>> b
array([0, 1, 2, 3, 4])
>>> b.shape
(5,)
```

计算数组 **a** 和 **b** 的和，得到一个加法表，它相当于计算两个数组中所有元素组的和，得到一个形状为(6,5)的数组：

```
>>> c = a + b
>>> c
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44],
       [50, 51, 52, 53, 54]])
>>> c.shape
(6, 5)
```

由于数组 **a** 和 **b** 的维数不同，根据规则(1)，需要让数组 **b** 的 **shape** 属性向数组 **a** 对齐，于是将数组 **b** 的 **shape** 属性前面加 1，补齐为(1,5)。相当于做了如下计算：

```
>>> b.shape=1,5
>>> b
array([[0, 1, 2, 3, 4]])
```

这样一来，做加法运算的两个输入数组的 **shape** 属性分别为(6,1)和(1,5)，根据规则(2)，输出数组各个轴的长度为输入数组各个轴长度的最大值，可知输出数组的 **shape** 属性为(6,5)。

由于数组 **b** 第 0 轴的长度为 1，而数组 **a** 第 0 轴的长度为 6，因此为了让它们在第 0 轴上能够相加，需要将数组 **b** 第 0 轴的长度扩展为 6，这相当于：

```
>>> b = b.repeat(6,axis=0)
>>> b
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```


由于数组 a 第 1 轴的长度为 1，而数组 b 第 1 轴的长度为 5，因此为了让它们在第 1 轴上能够相加，需要将数组 a 第 1 轴的长度扩展为 5，这相当于：

```
>>> a = a.repeat(5, axis=1)
>>> a
array([[ 0,  0,  0,  0,  0],
       [10, 10, 10, 10, 10],
       [20, 20, 20, 20, 20],
       [30, 30, 30, 30, 30],
       [40, 40, 40, 40, 40],
       [50, 50, 50, 50, 50]])
```

经过上述处理之后，数组 a 和 b 就可以按对应元素进行相加运算了。

当然，在执行“a+b”运算时，NumPy 内部并不会真正将长度为 1 的轴用 repeat() 进行扩展，这样太浪费空间了。

由于这种广播计算很常用，因此 NumPy 提供了快速产生能进行广播运算的数组的 ogrid 对象。

```
>>> x,y = np.ogrid[:5,:5]
>>> x
array([[0],[1],[2],[3],[4]])
>>> y
array([[0, 1, 2, 3, 4]])
```



mgrid 对象的用法和 ogrid 对象类似，但是它所返回的是进行广播之后的数组。请读者运行“np.mgrid[:5,:5]”试试看。

ogrid 是一个很有趣的对象，它和多维数组一样，用切片元组作为下标，返回的是一组可以用来广播计算的数组。其切片下标有两种形式：

- 开始值:结束值:步长，和“np.arange(开始值, 结束值, 步长)”类似。
- 开始值:结束值:长度 j，当第三个参数为虚数时，它表示所返回数组的长度，其和“np.linspace(开始值, 结束值, 长度)”类似。

```
>>> x, y = np.ogrid[:1:4j, :1:3j]
>>> x
array([[ 0.          ],
       [ 0.33333333],
       [ 0.66666667],
       [ 1.          ]])
>>> y
array([[ 0. ,  0.5,  1. ]])
```

利用 `ogrid` 的返回值，可以很容易计算出二元函数在等间距网格上的值。下面是绘制三维曲面 $f(x, y) = xe^{x^2 - y^2}$ 的程序：



`numpy_ogrid_mlab.py`

用 `ogrid` 产生二维坐标网格，计算三维空间的曲面

```
import numpy as np
from enthought.mayavi import mlab

x, y = np.ogrid[-2:2:20j, -2:2:20j]
z = x * np.exp(- x**2 - y**2)

p1 = mlab.surf(x, y, z, warp_scale="auto")
mlab.axes(xlabel='x', ylabel='y', zlabel='z')
mlab.outline(p1)
mlab.show()
```

此程序使用 Mayavi 的 `mlab` 模块快速绘制如图 2-5 所示的 3D 曲面(见封二彩插)，关于 Mayavi 的相关内容将在随后的章节中进行介绍。

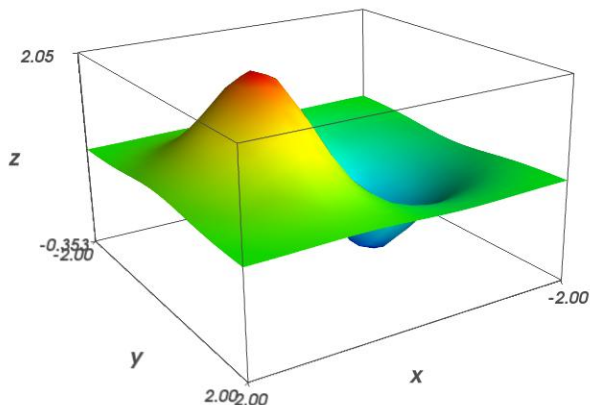


图 2-5 使用 `ogrid` 创建的三维曲面

如果已经有了多个表示在不同轴上取值的一维数组，并且想计算出由它们所构成网格上的每点的函数值，那么可以使用 `ix_()`：

```
>>> x = np.array([0, 1, 4, 10])
>>> y = np.array([2, 3, 8])
>>> gy, gx = np.ix_(y, x)
>>> gx
array([[ 0,  1,  4, 10]])
>>> gy
array([[2], [3], [8]])
```

```
>>> gx+gy # 广播运算
array([[ 2,  3,  6, 12],
       [ 3,  4,  7, 13],
       [ 8,  9, 12, 18]])
```

在上面的例子中，通过 `ix_()` 将数组 `x` 和 `y` 转换成能进行广播运算的二维数组。注意数组 `y` 对应广播运算结果中的第 0 轴，而数组 `x` 与第 1 轴对应。

2.2.5 ufunc 函数的方法

ufunc 函数对象本身还有一些方法，这些方法只对两个输入、一个输出的 ufunc 对象有效，其他的 ufunc 对象调用这些方法时会抛出 `ValueError` 异常。

`reduce()` 方法和 Python 的 `reduce()` 函数类似，它沿着 `axis` 参数指定的轴对数组进行操作，相当于将 `<op>` 运算符插入到沿 `axis` 轴的所有元素之间。

```
<op>.reduce (array, axis=0, dtype=None)
```

例如：

```
>>> np.add.reduce([1,2,3]) # 1 + 2 + 3
6
>>> np.add.reduce([1,2,3],[4,5,6], axis=1) # (1+2+3),(4+5+6)
array([ 6, 15])
```

`accumulate()` 和 `reduce()` 类似，只是它返回的数组和输入数组的形状相同，保存所有的中间计算结果：

```
>>> np.add.accumulate([1,2,3])
array([1, 3, 6])
>>> np.add.accumulate([1,2,3],[4,5,6], axis=1)
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

`reduceat()` 计算多组 `reduce()` 的结果，通过 `indices` 参数指定一系列的起始和终止位置。它的计算有些特别，让我们通过例子详细解释一下：

```
>>> a = np.array([1,2,3,4])
>>> result = np.add.reduceat(a,indices=[0,1,0,2,0,3,0])
>>> result
array([ 1,  2,  3,  3,  6,  4, 10])
```

对于 `indices` 参数中的每个元素都会计算出一个值，因此最终的计算结果和 `indices` 参数的长度相同。结果数组 `result` 中除最后一个元素之外，都按照如下计算得出：

```

if indices[i] < indices[i+1]:
    result[i] = <op>.reduce(a[indices[i]:indices[i+1]])
else:
    result[i] = a[indices[i]]

```

最后一个元素则按照如下计算得出：

```
<op>.reduce(a[indices[-1]:])
```

因此上面例子中，数组 `result` 的每个元素都按照如下计算得出：

```

1 : a[0] -> 1
2 : a[1] -> 2
3 : a[0] + a[1] -> 1 + 2
3 : a[2] -> 3
6 : a[0] + a[1] + a[2] -> 1 + 2 + 3 = 6
4 : a[3] -> 4
10: a[0] + a[1] + a[2] + a[4] -> 1 + 2 + 3 + 4 = 10

```

可以看出：`result[::2]`和 `a` 相等，而 `result[1::2]`和 `np.add.accumulate(a)`相等。使用 `reduceat()` 可以对数组中的多个片段同时进行 `reduce` 运算。

`ufunc` 函数对象的 `outer()`方法对其两个参数数组中每两对元素的组合进行运算。如果数组 `a` 的维数为 `M`，数组 `b` 的维数为 `N`，那么使用 `ufunc` 函数 `op` 的 `outer()`方法对数组 `a`、`b` 计算后，生成的数组 `c` 的维数为 `M+N`。数组 `c` 的形状是数组 `a`、`b` 形状的结合。例如数组 `a` 的形状为 `(2,3)`，数组 `b` 的形状为 `(4,5)`，则数组 `c` 的形状为 `(2,3,4,5)`。

让我们看一个例子：

```

>>> np.multiply.outer([1,2,3,4,5],[2,3,4])
array([[ 2,  3,  4],
       [ 4,  6,  8],
       [ 6,  9, 12],
       [ 8, 12, 16],
       [10, 15, 20]])

```

可以看出，通过 `outer()`计算得到的结果是如下的乘法表：

```

*| 2  3  4
-----
1| 2  3  4
2| 4  6  8
3| 6  9 12
4| 8 12 16
5|10 15 20

```

如果将这两个数组按照等同程序一步一步地进行计算，就会发现乘法表最终是通过广播的方式计算出来的。

2.3 多维数组的下标存取

在前面的介绍中，我们通过一些实例介绍了如何对多维数组进行下标访问。实际上，NumPy 提供的下标功能十分强大，在掌握了“广播”相关的知识之后，让我们再回过头来系统地学习数组的下标规则。

2.3.1 下标对象

首先，多维数组的下标应该是一个长度上与数组的维数相同的元组。如果下标元组的长度比数组的维数大，就会出错。如果小，就需要在下标元组的后面补“:”，使得它的长度与数组维数相同。

如果下标对象不是元组，NumPy 会首先把它转换为元组。这种转换可能会和用户所希望的不一致，因此为了避免出现问题，请显式地使用元组作为下标。例如，数组 `a` 是一个三维数组，下面分别使用一个二维列表 `lidx` 和一个二维数组 `aidx` 作为下标，得到的结果是不一样的。

```
>>> a = np.arange(3*4*5).reshape(3,4,5)
>>> lidx = [[0],[1],[2]]
>>> aidx = np.array(lidx)
>>> a[lidx]
array([7])
>>> a[aidx]
array([[[[ 0, 1, 2, 3, 4],
          [ 5, 6, 7, 8, 9],
          [[省略]]
```

这是因为 NumPy 将列表 `lidx` 转换成了 `([0],[1],[2])`，而将数组 `aidx` 转换成了 `[aidx, :, :]`：

```
>>> a[tuple(lidx)]
array([7])
>>> a[aidx, :, :]
array([[[[ 0, 1, 2, 3, 4],
          [ 5, 6, 7, 8, 9],
          [[省略]]
```

经过各种转换和添加“:”之后，得到了一个标准的下标元组。它的各个元素有如下几种类型：切片、整数、整数数组和布尔数组。如果元素不是这些类型，如列表或元组，就将其转换成整数数组。

如果下标元组的所有元素都是切片和整数，那么用它作为下标得到的是原始数组的一个视图，即它和原始数组共享数据存储空间。

2.3.2 整数数组作为下标

下面让我们看看下标元组中的元素由切片和整数数组构成的情况。假设整数数组有 N_c 个，切片有 N_s 个。 $N_c + N_s$ 为数组的维数 D 。

首先这 N_c 个整数数组必须满足广播条件，假设它们进行广播之后的维数为 M ，形状为 $(d_0, d_1, \dots, d_{m-1})$ 。

如果 N_s 为 0，即没有切片元素，那么下标得到的结果数组 **result** 的形状和整数数组广播之后的形状相同。它的每个元素值可按照下面的公式得出：

$$\text{result}[i_0, i_1, \dots, i_{m-1}] = X[\text{ind}_0[i_0, i_1, \dots, i_{m-1}], \dots, \text{ind}_{N_c-1}[i_0, i_1, \dots, i_{m-1}]]$$

其中 ind_0 到 ind_{N_c-1} 为进行广播之后的整数数组。让我们通过一个例子加深对此公式的理解：

```
>>> i0 = np.array([[1,2,1],[0,1,0]])
>>> i1 = np.array([[0]],[1]])
>>> i2 = np.array([[2,3,2]])
>>> b = a[i0, i1, i2]
>>> b
array([[22, 43, 22],
       [ 2, 23,  2]],
       [[27, 48, 27],
       [ 7, 28,  7]])
```

首先，i0、i1、i2 三个整数数组的 **shape** 属性分别为(2,3)、(2,1,1)、(1,1,3)，根据广播规则，先在长度不足 3 的 **shape** 属性前面补 1，使它们的维数相同，广播之后的 **shape** 属性为各个轴的最大值：

```
(1, 2, 3)
(2, 1, 1)
(1, 1, 3)
-----
2  2  3
```

即三个整数数组广播之后的 **shape** 属性为(2,2,3)，这也就是下标运算所得到的结果数组的维数：

```
>>> b.shape
(2, 2, 3)
```

可以使用 **broadcast_arrays()** 查看广播之后的数组：

```
>>> ind0, ind1, ind2 = np.broadcast_arrays(i0, i1, i2)
>>> ind0
array([[[1, 2, 1],
        [0, 1, 0]],
       [[1, 2, 1],
        [0, 1, 0]]])
>>> ind1
array([[[0, 0, 0],
        [0, 0, 0]],
       [[1, 1, 1],
        [1, 1, 1]]])
>>> ind2
array([[[2, 3, 2],
        [2, 3, 2]],
       [[2, 3, 2],
        [2, 3, 2]]])
```

对于数组 b 中的任意一个元素 $b[i,j,k]$ ，它是数组 a 中经过 $ind0$ 、 $ind1$ 和 $ind2$ 进行下标转换之后的值：

```
>>> i,j,k = 0,1,2
>>> b[i,j,k]
2
>>> a[ind0[i,j,k],ind1[i,j,k],ind2[i,j,k]]
2
>>> i,j,k = 1,1,1
>>> b[i,j,k]
28
>>> a[ind0[i,j,k],ind1[i,j,k],ind2[i,j,k]]
28
```

下面考虑 N_s 不为 0 的情况。当存在切片下标时，情况就变得更加复杂了。可以细分为两种情况：下标元组中的整数数组之间没有切片，即整数数组只有一个或者是连续的。这时结果数组的 `shape` 属性为：将原始数组的 `shape` 属性中整数数组所占据的部分替换为它们广播之后的 `shape` 属性。例如，假设原始数组 a 的 `shape` 属性为 $(3,4,5)$ ， $i0$ 和 $i1$ 广播之后的形状为 $(2,2,3)$ ，则 $a[1:3,i0,i1]$ 的形状为 $(2,2,2,3)$ ：

```
>>> c=a[1:3, i0, i1]
>>> c.shape
(2, 2, 2, 3)
```

其中，数组 c 的 `shape` 属性中的第一个 2 是切片 “1:3” 的长度，后面的 $(2,2,3)$ 则是 $i0$ 和 $i1$ 广播之后数组的形状：

```
>>> ind0, ind1 = np.broadcast_arrays(i0, i1)
>>> ind0.shape
(2, 2, 3)
>>> i,j,k = 1,1,2
>>> c[:,i,j,k]
array([21, 41])
>>> a[1:3,ind0[i,j,k],ind1[i,j,k]] # 和 c[:,i,j,k]的值相同
array([21, 41])
```

如果下标元组中的整数数组不是连续的，那么结果数组的 `shape` 属性为整数数组广播之后的形状后面再加上切片元素对应的形状。例如，`a[i0,:,i1]` 的 `shape` 属性为 `(2,2,3,4)`。其中，`(2,2,3)` 是 `i0` 和 `i1` 广播之后的形状，而 4 是数组 `a` 第 1 轴的长度：

```
>>> d = a[i0, :, i1]
>>> d.shape
(2, 2, 3, 4)
>>> i,j,k = 1,1,2
>>> d[i,j,k,:]
array([ 1,  6, 11, 16])
>>> a[ind0[i,j,k],:,ind1[i,j,k]]
array([ 1,  6, 11, 16])
```

2.3.3 一个复杂的例子

下面让我们用所学的下标存取的知识，解决在 NumPy 邮件列表中提出的一个比较经典的问题，此问题的原文链接地址为：



<http://mail.scipy.org/pipermail/numpy-discussion/2008-July/035764.html>

NumPy 邮件列表中的原文链接

我们对问题进行一些简化，提问者想要实现的下标运算是：有一个形状为 `(I, J, K)` 的三维数组 `v` 和一个形状为 `(I, J)` 的二维数组 `idx`，`idx` 的每个值都是 0 到 `K-L` 的整数。他想通过下标运算得到一个数组 `r`，对于第 0 轴和第 1 轴的每个下标 `i` 和 `j` 都满足下面条件：

```
r[i,j,:] = v[i,j,idx[i,j]:idx[i,j]+L]
```

如图 2-6 所示^⑤，左图中不透明的方块是我们希望获取的部分，通过下标运算之后将得到右图所示的数组。

^⑤ 绘制此图的源程序为 “`numpy_array_index_demo.py`”。

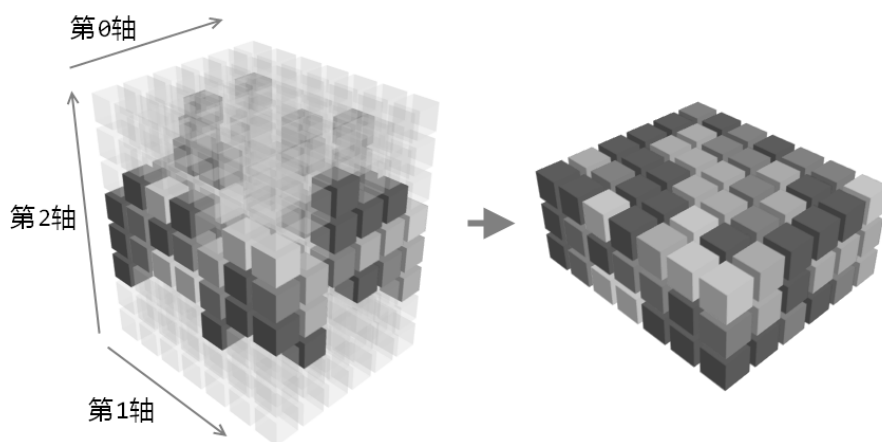


图 2-6 三维数组下标运算问题的示意图

首先创建一个方便调试的数组 `v`，它在第 2 轴上每一层的值就是该层的高度，即 `v[:, :, i]` 的所有元素的值都为 `i`。然后随机产生数组 `idx`，它的每个元素的取值都在 0 到 `K-L` 之间：



`numpy_array_index.py`
三维数组下标运算问题

```
>>> I, J, K, L = 6, 7, 8, 3
>>> _, _, v = np.mgrid[:I, :J, :K]
>>> idx = np.random.randint(0, K-L, size=(I,J))
```

然后用数组 `idx` 创建第 2 轴的下标数组 `idx_k`，它是一个形状为 `(I,J,L)` 的三维数组。它的第 2 轴上的每一层的值都等于 `idx` 数组加上层的高度，即 “`idx_k[:, :, i] = idx[:, :] + i`”：

```
>>> idx_k = idx.reshape(I,J,1) + np.arange(L)
>>> idx_k.shape
(6, 7, 3)
```

然后分别创建第 0 轴和第 1 轴的下标数组，它们的 `shape` 属性分别为 `(I,1,1)` 和 `(1,J,1)`：

```
>>> idx_i, idx_j, _ = np.ogrid[:I, :J, :K]
```

使用 `idx_i`、`idx_j`、`idx_k` 对数组 `v` 进行下标运算即可得到结果：

```
>>> r = v[idx_i, idx_j, idx_k]
>>> i, j = 2, 3 # 验证结果，读者可以修改为使用循环测试
>>> r[i,j,:]
array([4, 5, 6])
>>> v[i,j,idx[i,j]:idx[i,j]+L]
array([4, 5, 6])
```

2.3.4 布尔数组作为下标

当使用布尔数组直接作为下标对象或者元组下标对象中有布尔数组时，都相当于用 `nonzero()` 将布尔数组转换成一组整数数组，然后使用整数数组进行下标运算。

`nonzeros(a)` 返回数组 `a` 中值不为零的元素的下标，它的返回值是一个长度为 `a.ndim` (数组 `a` 的轴数) 的元组，元组的每个元素都是一个整数数组，其值为非零元素的下标在对应轴上的值。例如，对于一维布尔数组 `b1`，`nonzero(b1)` 得到的是一个长度为 1 的元组，它表示 `b1[0]` 和 `b1[2]` 的值不为 0 (False)。

```
>>> b1 = np.array([True, False, True, False])
>>> np.nonzero(b1)
(array([0, 2]),)
```

对于二维数组 `b2`，`nonzero(b2)` 得到的是一个长度为 2 的元组。它的第 0 个元素是数组 `a` 中值不为 0 的元素的第 0 轴的下标，第 1 个元素则是第 1 轴的下标，因此从下面的结果可知 `b2[0,0]`、`b[0,2]` 和 `b2[1,0]` 的值不为 0：

```
>>> b2 = np.array([[True, False, True], [True, False, False]])
>>> np.nonzero(b2)
(array([0, 0, 1]), array([0, 2, 0]))
```

当布尔数组直接作为下标时，相当于使用由 `nonzero()` 转换之后的元组作为下标对象：

```
>>> a = np.arange(3*4*5).reshape(3,4,5)
>>> a[b2]
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])
>>> a[np.nonzero(b2)]
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])
```

当下标对象是元组，并且其中有布尔数组时，相当于将布尔数组展开为由 `nonzeros()` 转换之后的各个整数数组：

```
>>> a[1:3, b2]
array([[20, 22, 25],
       [40, 42, 45]])
>>> a[1:3, np.nonzero(b2)[0], np.nonzero(b2)[1]]
array([[20, 22, 25],
       [40, 42, 45]])
```

TraitsUI——轻松制作用户界面

Python 有着丰富的界面开发库，除了默认安装的 Tkinter 以外，wxPython、pyQt4 等都是非常优秀的界面开发库。但是它们有一个共同的问题：需要开发者掌握众多的 API 函数，许多细节需要开发者自己进行配置，例如控件的属性、位置以及事件响应，等等。

在开发科学计算程序时，我们希望快速实现一个够用的界面，让用户能够交互式地处理数据，而又不希望在界面制作上花费过多的精力。以 Traits 库为基础、以 MVC 模式为设计思想的 TraitsUI 库就是实现这一理想的最佳方案。

MVC 模式

MVC 的英文全称为 Model-View-Controller，它的目的是实现一种动态的程序设计，简化程序的修改和扩展工作，并且使程序的各个部分能够充分地重复利用。

Model(模型)：程序中存储数据以及对数据进行处理的部分。

View(视图)：程序的界面部分，实现数据的显示。

Controller(控制器)：起到视图和模型之间的组织作用，控制程序的流程，例如将界面的操作转换为对模型的处理。

7.1 默认界面

TraitsUI 库是一套建立在 Traits 库基础之上的用户界面库。它和 Traits 库紧密相连，如果读者已经设计好了一个从 HasTraits 继承而来的类，那么直接调用其 `configure_traits()` 方法，系统将会使用 TraitsUI 库自动生成对话框界面，以供用户交互式地修改对象的 Trait 属性。下面是一个简单的例子：



traitsUI_default_view.py

编辑 HasTraits 对象的对话框

```
from enthought.traits.api import HasTraits, Str, Int

class Employee(HasTraits):
    name = Str
```

```

department = Str
salary = Int
bonus = Int

Employee().configure_traits()

```

此程序创建一个 `Employee` 类，然后调用 `configure_traits()` 显示出如图 7-1(左)所示的默认界面。

在此自动生成的界面中，所有的属性都采用文本框进行编辑，并且每个文本框前面都有一个文字标签，上面的文字根据 `Trait` 属性名自动生成：第一个字母变为大写，所有的下划线变为空格。对话框的最下面提供了 `OK` 和 `Cancel` 按钮以便确定或取消对 `Trait` 属性的修改。

由于 `salary` 属性定义为 `Int` 类型，因此当输入不能转换为整数的字符串时，输入框将以红色背景提醒输入错误，并且 `OK` 按钮变成无效，如图 7-1(右)所示。

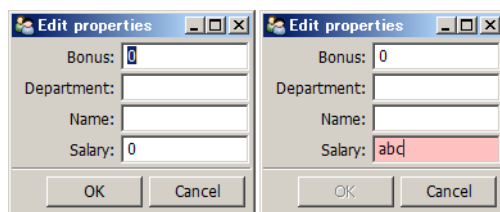


图 7-1 自动生成的 `Employee` 类的对话框(左)、提醒非法的输入数据并且使 `OK` 按钮无效(右)

没有写一行与界面相关的代码，就能得到一个已经够实用的界面，应该还是很令人满意的。如果想手工控制界面的设计和布局，就需要添加自己的代码了。

7.2 用 View 定义界面

`HasTraits` 的派生类用 `Trait` 属性保存数据，它相当于 MVC 模式中的模型(Model)。当没有指定界面的显示方式时，`Traits` 库会自动创建一个默认的界面。我们可以通过视图(View)对象为模型设计更加实用的界面。

7.2.1 外部视图和内部视图

下面是用视图对象定义界面的完整程序，图 7-2 显示的是界面截图。



traitsUI_simple_view.py
使用视图对象描述界面

```

from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item ❶

class Employee(HasTraits):

```

```

name = Str
department = Str
salary = Int
bonus = Int

view = View( ❷
    Item('department', label=u"部门", tooltip=u"在哪个部门干活"), ❸
    Item('name', label=u"姓名"),
    Item('salary', label=u"工资"),
    Item('bonus', label=u"奖金"),
    title = u"员工资料", width=250, height=150, resizable=True ❹
)

if __name__ == "__main__":
    p = Employee()
    p.configure_traits()

```

此程序在模型类 `Employee` 的基础之上，添加了与界面显示相关的代码。❶界面相关的内容都位于 `TraitsUI` 库中，这里从中载入 `View` 和 `Item`。`View` 是描述界面的视图类，`Item` 是描述界面中的控件和模型对象的 `Trait` 属性之间关系的类。

❷在 `Employee` 类中创建了一个 `View` 对象，为 `Employee` 类的每个 `Trait` 属性创建对应的 `Item` 对象。❸创建多个 `Item` 对象，并作为参数传递给 `View()`。`Item` 对象是视图的基本组成单位，每个 `Item` 对象描述界面中的一个编辑器，这些编辑器用于编辑模型对象中对应的 `Trait` 属性的值。界面中的编辑器按照 `Item` 对象传递给 `View()` 的先后顺序显示，而不再按照 `Traits` 属性名排序。`Item` 对象有许多参数，它们对 `Item` 对象的内容、表现以及行为进行描述。第一个参数指定与编辑器对应的 `Trait` 属性名，`label` 和 `tooltip` 参数设置编辑器的标签和提示文本。`Item` 对象还有很多其他属性，请读者参考 `TraitsUI` 的用户手册，或者在 `IPython` 中输入“`Item??`”直接查看其源代码。下面是 `Item` 类的部分源程序，`Item` 类从 `HasTraits` 继承，因此它的属性都是 `Trait` 属性：

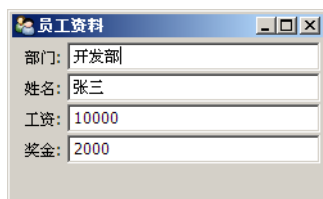


图 7-2 使用视图对象描述界面

```

class Item ( ViewSubElement ):
    """ An element in a Traits-based user interface.
    """

    # Trait definitions:

    # A unique identifier for the item. If not set, it defaults to the value
    # of **name**.
    id = Str

    # User interface label for the item in the GUI. If this attribute is not

```

```
# set, the label is the value of **name** with slight modifications:
# underscores are replaced by spaces, and the first letter is capitalized.
# If an item's **name** is not specified, its label is displayed as
# static text, without any editor widget.
label = Str

# Name of the trait the item is editing:
name = Str
```

除了 Item 之外，TraitsUI 模块还定义了 Item 的几个派生类：Label、Heading 和 Spring。它们只用于辅助界面布局，因此不需要和模型对象的 Trait 属性关联。

④View 类也从 HasTraits 继承，可以直接在创建 View 对象时，通过关键字参数设置其 Trait 属性。title 属性为窗口标题栏中的文字，width 和 height 属性为窗口的大小，resizable 属性为 True 表示窗口的大小可变。

同一个模型对象可以通过多个视图对象用不同的界面显示，下面看一个例子：



traitsUI_views.py
使用多个视图对象

```
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Group, Item ❶

g1 = [Item('department', label=u"部门", tooltip=u"在哪个部门干活"), ❷
      Item('name', label=u"姓名")]
g2 = [Item('salary', label=u"工资"),
      Item('bonus', label=u"奖金")]

class Employee(HasTraits):
    name = Str
    department = Str
    salary = Int
    bonus = Int

    traits_view = View( ❸
        Group(*g1, label = u'个人信息', show_border = True),
        Group(*g2, label = u'收入', show_border = True),
        title = u"缺省内部视图")

    another_view = View( ❹
        Group(*g1, label = u'个人信息', show_border = True),
        Group(*g2, label = u'收入', show_border = True),
        title = u"另一个内部视图")

global_view = View( ❺
```

```

Group(*g1, label = u'个人信息', show_border = True),
Group(*g2, label = u'收入', show_border = True),
title = u"外部视图")

p = Employee()

# 使用内部视图 traits_view
p.edit_traits() ❹

# 使用内部视图 another_view
p.edit_traits(view="another_view") ❺

# 使用外部视图 view1
p.configure_traits(view=global_view) ❻

```

❶从 TraitsUI 库载入 Group 类，用 Group 对象可以对界面中的编辑器进行分组。为了使后续定义视图对象的程序更加简洁，❷程序中定义了两个全局列表 g1 和 g2，它们的元素都是 Item 对象。❸❹在 Employee 类的内部用 View() 定义了两个视图对象：traits_view 和 another_view。而❺定义了一个全局的视图对象：global_view。在定义视图对象时，用 Group 对用于定义界面上编辑器的 Item 对象进行分组。

值得注意的是，Employee 类中定义的两个视图对象既不是类的属性，也不是实例的属性。这些内部视图对象会放到 Employee 类的 __view_traits__ 属性中。__view_traits__ 属性是一个 ViewElements 对象，它的 content 属性是保存所有内部视图的字典：

```

>>> run traitsUI_views.py
>>> Employee.__view_traits__.content.keys()
['another_view', 'traits_view']

```

❻当调用 edit_traits() 显示界面时，默认使用在模型类内部定义的默认视图对象 traits_view 生成界面。❼使用 view 参数可以指定显示界面时所使用的内部视图对象的名称。❽也可以直接将视图对象传递给 view 参数，这样可以使用在模型类外定义的视图对象生成界面。图 7-3 显示了用三种视图对象生成的界面。

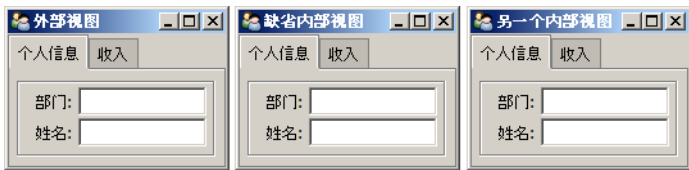


图 7-3 使用外部视图和内部视图定义界面的显示

edit_traits() 和 configure_traits() 一样，也被用于生成界面，它们的区别在于：edit_traits() 显示界面之后不进入后台界面库的消息循环，因此如果直接运行只调用 edit_traits() 的程序，界面将在显示之后立即关闭，程序的运行也随之结束。而对于 configure_traits()，将进入消息循环，

直到用户关闭所有窗口。因此通常情况下，主界面窗口或模态对话框^①使用 `configure_traits()` 显示，而无模态窗口或对话框则使用 `edit_traits()` 显示。

选择后台界面库

用 TraitsUI 库创建的界面可以选择后台界面库，目前支持的有 `qt4` 和 `wx` 两种。在启动程序时添加 “-toolkit qt4” 或 “-toolkit wx” 可选择使用何种界面库生成界面。本书全部使用 `wx` 作为后台界面库。

在本小节的例子中，`Employee` 类用于保存数据，因此它属于 MVC 模式中的模型(Model)，而 `View` 对象定义了 `Employee` 的界面显示部分，它属于视图(View)。通过将视图对象传递给模型对象的 `configure_traits()` 方法，将模型对象和视图对象联系起来。在用来定义编辑器的 `Item` 对象中，不直接引用模型对象的属性，而是通过属性名和模型对象进行联系。这样一来，模型和视图之间的耦合性很弱，只需要属性名匹配，同一个视图对象可以运用到不同的模型对象之上。

有时候我们希望模型类知道如何显示它自己，这时可以在模型类的内部定义视图。在模型类中定义的视图可以被其派生类继承，因此派生类能使用父类的视图。在调用 `configure_traits()` 时如果不设置 `view` 参数，将使用模型对象内部的默认视图对象生成界面。如果在模型类中定义了多个视图对象，默认使用名为 `traits_view` 的视图对象。

7.2.2 多模型视图

在上一小节的例子中，一个模型可以对应多个视图。同样，使用一个视图可以将多个模型对象的数据显示在一个界面窗口中。下面是用一个视图对象同时显示多个模型对象的例子。程序的运行界面如图 7-4 所示。



traitsUI_multi_models.py

用一个视图对象同时显示多个模型对象

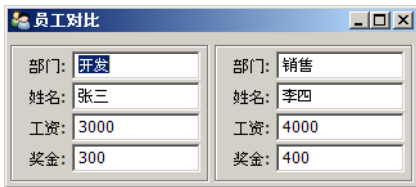


图 7-4 用一个视图对象同时显示多个模型对象

对于前面的模型类 `Employee`，我们可以设计如下的复合视图对象 `comp_view`，它能同时显示两个 `Employee` 对象：

```
comp_view = View(
```

^① 在模态对话框显示时，程序的其它其他窗口均无法响应用户操作。


```

Group(
    Group(
        Item('p1.department', label=u"部门"),
        Item('p1.name', label=u"姓名"),
        Item('p1.salary', label=u"工资"),
        Item('p1.bonus', label=u"奖金"),
        show_border=True
    ),
    Group(
        Item('p2.department', label=u"部门"),
        Item('p2.name', label=u"姓名"),
        Item('p2.salary', label=u"工资"),
        Item('p2.bonus', label=u"奖金"),
        show_border=True
    ),
    orientation = 'horizontal'
),
title = u"员工对比"
)

```

注意: `Item` 对象的第一个参数不是简单的模型对象的属性名, 它同时设置了 `Item` 对象的两个属性——`object` 和 `name`。例如, 参数 `"p1.department"` 将设置 `Item` 对象的 `object` 属性为 `"p1"`, `name` 属性为 `"department"`。`object` 属性告诉 `Item` 对象如何找到模型对象, 而 `name` 属性则告诉 `Item` 对象如何找到模型对象中与其对应的属性。

接下来, 下面的程序生成组成模型对象的两个 `Employee` 对象——`employee1` 和 `employee2`:

```

employee1 = Employee(department = u"开发", name = u"张三", salary = 3000, bonus = 300)
employee2 = Employee(department = u"销售", name = u"李四", salary = 4000, bonus = 400)

```

在显示界面时, 使用 `context` 参数将包含两个模型对象的字典传递给 `configure_traits()`:

```

HasTraits().configure_traits(view=comp_view, context={"p1":employee1, "p2":employee2})

```

通过 `context` 参数传递的实际上是视图对应的模型。这里的模型对象是一个字典, 它的键和 `Item` 对象的 `object` 属性的值相同。由于已经通过 `context` 参数传递了模型对象, 因此 `configure_traits()` 方法原本所属的对象将不会被用作界面的模型对象。这里直接创建一个临时的 `HasTraits` 对象, 然后调用其 `configure_traits()` 方法。

如果读者认为这种写法有些取巧, 可以直接调用视图对象的 `ui()` 方法显示界面, 它的参数就是界面所要显示的模型对象。由于 `ui()` 和 `edit_traits()` 一样^② 不会开始界面库的消息循环, 因此需要在运行 `ui()` 之后添加开始消息循环的代码。下面的消息循环代码支持所有的后台界面库:

② 实际上, `edit_traits()` 会调用视图对象的 `ui()` 来显示界面。

```
comp_view.ui({"p1":employee1, "p2":employee2})

from enthought.pyface.api import GUI
GUI().start_event_loop() # 开始后台界面库的消息循环
```

如果读者只需要用 wx 库显示界面，也可以用下面的代码来运行 wx 库的消息循环：

```
import wx
wx.PySimpleApp().MainLoop() # 开始 wx 库的消息循环
```

7.2.3 Group 对象

在前面例子的视图定义中，我们通过 Group 对象将一组相关的 Item 对象组织在一起。本节详细介绍如何使用 Group 对象组织界面。

在程序“traitsUI_views.py”中，View 对象中直接放置了多个 Group 对象，效果如图 7-3 所示，它使用标签页的形式显示在 View 下定义的多个 Group 对象。

如果希望能同时看到两个 Group 对象，可以像程序“traitsUI_multi_models.py”一样，再创建一个 Group 对象，将那两个 Group 对象包括起来，效果如图 7-4 所示。由于外层 Group 对象的 orientation 属性为'horizontal'，因此它内部的两个 Group 对象将水平方向排列。下面的代码显示了 View 对象中 Group 对象的嵌套关系：

```
View(
    Group(
        Group(...),
        Group(...),
        orientation = 'horizontal'
    )
)
```

在创建 Group 对象时，可以通过设置其 orientation 和 layout 等属性，改变 Group 对象的内容呈现方式。由于某些设置会经常用到，因此 TraitsUI 还提供了几个 Group 的派生类，以覆盖这些属性的默认值。例如，下面是从 Group 类继承的 HSplit 类的代码：

```
class HSplit ( Group ):
    # ... ...
    layout      = 'split'
    orientation = 'horizontal'
```

HSplit 对象将其包括的内容按照水平方向排列，并且在两块区域之间添加一个可调整位置的分隔条。使用 HSplit 和如下的代码等价：

```
Group( ... , layout = 'split', orientation = 'horizontal')
```

为了正确显示分隔条，内容中需要有一个 Group 对象具有 scrollable 属性，如下面的省略代码所示：

```
view1 = View(  
    HSplit(  
        VGroup(  
            ... ...,  
            scrollable = True  
        ),  
        VGroup(  
            ... ...  
        ),  
    ),  
    resizable = True,  
    width = 400,  
    height = 150  
)
```

下面的程序演示了 4 种不同的界面分组方式，效果如图 7-5 所示。



traitsUI_group.py

用 Group 对 View 中的编辑器进行分组

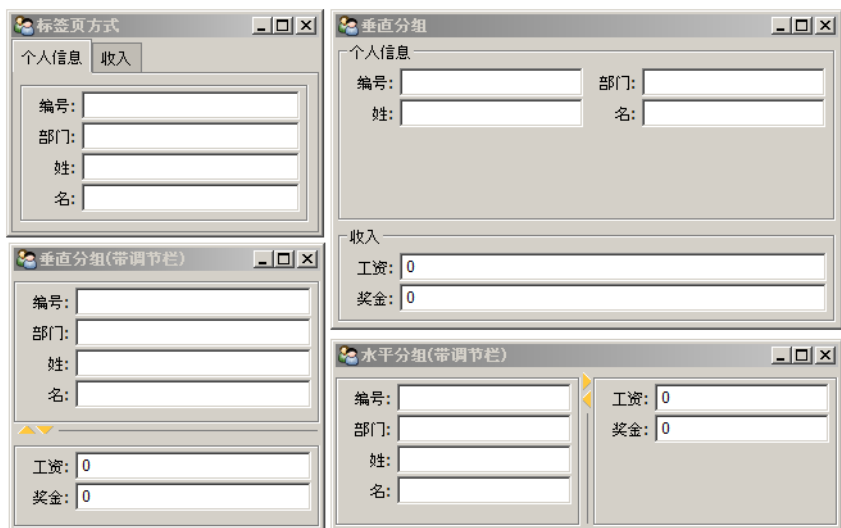


图 7-5 在界面中用 Group 对象进行分组

下面是 Group 的各种派生类及其对应的属性的默认设置：

- HGroup: 内容水平排列。

```
Group(orientation= 'horizontal')
```

- HFlow: 内容水平排列, 当超过水平宽度时, 将自动换行。show_labels 属性为 False, 表示隐藏其中的所有编辑器的标签文字。

```
Group(orientation= 'horizontal', layout='flow', show_labels=False)
```

- HSplit: 内容水平分隔, 中间插入分隔条。

```
Group(orientation= 'horizontal', layout='split')
```

- Tabbed: 内容分标签页显示。

```
Group(orientation= 'horizontal', layout='tabbed')
```

- VGroup: 内容垂直排列。

```
Group(orientation= 'vertical')
```

- VFlow: 内容垂直排列, 当超过垂直高度时, 将自动换列。

```
Group(orientation= 'vertical', layout='flow', show_labels=False)
```

- VFold: 内容垂直排列, 可折叠。

```
Group(orientation= 'vertical', layout='fold', show_labels=False)
```

- VGrid: 按照多列网格进行垂直排列, columns 属性决定网格的列数。

```
Group(orientation= 'vertical', columns=2)
```

- VSplit: 内容垂直排列, 中间插入分隔条。

```
Group(orientation= 'vertical', layout='split')
```

除了上面的 orientation、layout、show_labels、columns 等属性之外, Group 对象还提供了许多其他的属性以控制其显示效果。和 Item 一样, 读者可以在 Group 类的源程序中找到每个属性的详细说明。

下面我们通过一个例子说明 visible_when 和 enabled_when 属性的用法。这两个属性控制 Group 对象在界面中是否显示以及是否有效。



Item 也提供了 visible_when 和 enabled_when 属性, 用法和 Group 的完全相同。



traitsUI_group_condition.py
控制 Group 的显示以及是否有效

```
class Shape(HasTraits):
    shape_type = Enum("rectangle", "circle")
    editable = Bool
    x, y, w, h, r = [Int]*5

    view = View(
        VGroup(
            HGroup(Item("shape_type"), Item("editable")),
            VGroup(Item("x"), Item("y"), Item("w"), Item("h"),
                visible_when="shape_type=='rectangle'", enabled_when="editable"),
            VGroup(Item("x"), Item("y"), Item("r"),
                visible_when="shape_type=='circle'", enabled_when="editable"),
        ), resizable = True)
```

程序中, Shape 是一个表示矩形或圆形的类, 其具体形状由 shape_type 属性决定, 而图形的参数则由 x、y、w、h、r 等属性决定。editable 属性决定是否能通过用户界面修改图形参数。在视图定义中, 使用 VGroup 对象定义了两个编辑器组, 分别编辑矩形参数和圆形参数。通过设置 VGroup 的 visible_when 和 enabled_when 属性, 将模型对象的 shape_type 属性和 editable 属性与编辑器的界面显示联系起来。

visible_when 和 enabled_when 属性都是表示布尔表达式的字符串。当布尔表达式中涉及的模型对象的属性发生变化时, 将会对字符串进行求值, 并根据求值结果更新界面的显示。图 7-6 是程序的显示效果, 其中左图对应的 shape_type 属性为 "rectangle"、editable 属性为 True。当 shape_type 属性为 "rectangle" 时, 将显示矩形参数的编辑器而隐藏圆形参数的编辑器。当 editable 属性为 False 时, 所有编辑器都变成无效。

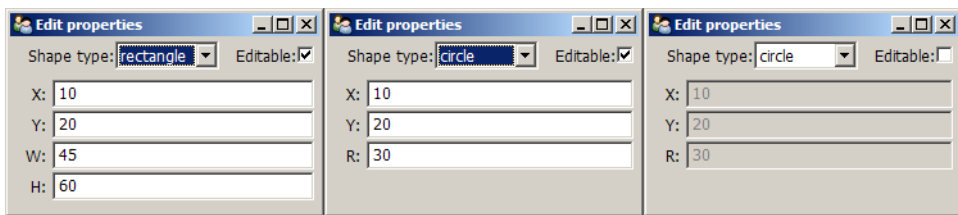


图 7-6 演示 visible_when 和 enabled_when 属性的用法

7.2.4 配置视图

前面介绍了如何使用 Item 和 Group 等对象组织窗口界面中的内容, 本节介绍如何配置窗口本身的属性。

通过 kind 属性可以修改 View 对象的显示类型:

- 'modal': 模态窗口, 非即时更新。
- 'live': 非模态窗口, 即时更新。
- 'livemodal': 模态窗口, 即时更新。
- 'nonmodal': 非模态窗口, 非即时更新。
- 'wizard': 向导类型。
- 'panel'、'subpanel': 嵌入到其他窗口中的面板, 即时更新, 非模态窗口。

其中, 'modal'、'live'、'livemodal'、'nonmodal'这 4 种类型的 View 对象都采用窗口界面来显示内容。所谓模态窗口, 是指在窗口关闭之前, 程序中的其他窗口都不能被激活。而即时更新则是指, 当窗口中编辑器的内容改变时, 会立即反映到编辑器所对应的模型对象的属性值。非即时更新的窗口则会复制模型对象, 所有的改变在副本上进行, 只有当用户单击 OK 或 Apply 按钮确定修改时, 才会修改原始模型对象的属性。

'wizard'由一系列特定的向导窗口组成, 属于模态窗口, 并且即时更新数据。

'panel'和'subpanel' 则是嵌入到窗口中的面板, 'panel'可以拥有自己的命令按钮, 而'subpanel'则没有命令按钮。

在对话框中经常可以看到 OK、Cancel、Apply 之类的按钮, 我们称之为命令按钮, 它们完成所有对话框都需要的一些共同操作。在 TraitsUI 中, 这些按钮可以通过 View 对象的 buttons 属性进行设置, 其值为要显示的按钮列表。

TraitsUI 中定义了 UndoButton、ApplyButton、RevertButton、OKButton、CancelButton 和 HelpButton 共 6 个标准的命令按钮, 每个按钮对应一个名字。在指定 buttons 属性时, 可以使用按钮的类名或者它对应的名字。与按钮类对应的名字就是类名除去 Button, 例如 UndoButton 对应为"Undo"。

enthought.traits.ui.menu 中还预定义了一些命令按钮, 以方便用户直接使用整套按钮:

```
OKCancelButtons = [OKButton, CancelButton ]
ModalButtons = [ ApplyButton, RevertButton, OKButton, CancelButton, HelpButton ]
LiveButtons = [ UndoButton, RevertButton, OKButton, CancelButton, HelpButton ]
```

7.3 用 Handler 控制界面和模型

虽然 TraitsUI 库的界面设计使用 MVC 模式, 但是在前面的介绍中, 没有出现过任何有关控制器(Controller)的代码。这是因为 TraitsUI 库提供了默认的控制器。我们可以通过继承 Handler 类, 创建自己的控制器类, 从而对视图和模型进行更自由的控制。

控制器的最主要功能是界面的事件处理, 这些事件要么对模型对象进行修改, 要么对界面进行修改。模型、视图以及控制器都是单独的实体。一个视图对象可以为多个模型对象生成界面, 同样, 一个控制器也可以用来处理不同视图界面中所产生的事件。因此控制器和视图以及模型之间不存在静态的联系。但是为了让控制器能够真正起作用, 它必须知道自己所要处理的

视图和模型。在 TraitsUI 中，控制器使用 UIInfo 对象获得它所对应的视图和模型。

当 TraitsUI 从视图创建界面时，将创建一个 UIInfo 对象，其中包括界面和模型对象的引用。当界面事件引起控制器的方法被调用时，这个 UIInfo 对象会作为参数传递给被调用的方法。

TraitsUI 提供了三种指定控制器的方法：

- 将控制器作为视图的属性：使用视图对象的 handler 属性指定控制器对象，此视图所产生的界面都使用它进行事件处理。
- 显示界面时设置：调用 edit_traits()、configure_traits() 或 ui() 等方法显示界面时，将控制器对象传递给 handler 参数。它比视图的 handler 属性有更高的优先级，将覆盖 handler 属性的设置。
- 将视图作为控制器的一部分进行定义。

7.3.1 用 Handler 处理事件

当显示某个视图时，将按照下面的顺序执行控制器中的方法：

- (1) 创建一个 UIInfo 对象。
- (2) 执行控制器的 init_info() 方法。
- (3) 创建一个 UI 对象以表示实际的窗口。
- (4) 执行控制器的 init() 方法。
- (5) 执行控制器的 position() 方法。
- (6) 显示实际的窗口。

除了上面的 init_info()、init()、position() 之外，当用户对界面进行操作时，还会执行如下的方法：

- apply(): 用户单击窗口中的 Apply 按钮，在模型对象的数据更新之后。
- close(): 用户关闭窗口，在窗口关闭之前。
- closed(): 在窗口关闭之后。
- revert(): 用户单击了 Revert 或 Cancel 按钮。
- setattr(): 用户通过界面修改了模型对象的某个 Trait 属性。
- show_help(): 用户单击了窗口中的 Help 按钮。

下面通过一个实例演示上述各个方法的使用法：



traitsUI_handler.py

用 Handler 的方法处理各种事件

```
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item, Group, Handler
from enthought.traits.ui.menu import ModalButtons

g1 = [Item('department', label=u"部门"),
      Item('name', label=u"姓名")]
```

```

g2 = [Item('salary', label=u"工资"),
      Item('bonus', label=u"奖金")]

class Employee(HasTraits):
    name = Str
    department = Str
    salary = Int
    bonus = Int

    def _department_changed(self): ❶
        print self, "department changed to ", self.department

    def __str__(self): ❷
        return "<Employee at 0x%x>" % id(self)

view1 = View(
    Group(*g1, label = u'个人信息', show_border = True),
    Group(*g2, label = u'收入', show_border = True),
    title = u"外部视图",
    kind = "modal", ❸
    buttons = ModalButtons
)

class EmployeeHandler(Handler): ❹
    def init(self, info):
        super(EmployeeHandler, self).init(info)
        print "init called"

    def init_info(self, info):
        super(EmployeeHandler, self).init_info(info)
        print "init info called"

    def position(self, info):
        super(EmployeeHandler, self).position(info)
        print "position called"

    def setattr(self, info, obj, name, value):
        super(EmployeeHandler, self).setattr(info, obj, name, value)
        print "setattr called:%s.%s=%s" % (obj, name, value)

    def apply(self, info):
        super(EmployeeHandler, self).apply(info)
        print "apply called"

    def close(self, info, is_ok):
        super(EmployeeHandler, self).close(info, is_ok)
        print "close called: %s" % is_ok

```



```

        return True

    def closed(self, info, is_ok):
        super(EmployeeHandler, self).closed(info, is_ok)
        print "closed called: %s" % is_ok

    def revert(self, info):
        super(EmployeeHandler, self).revert(info)
        print "revert called"

if __name__ == "__main__":
    zhang = Employee(name="Zhang")
    print "zhang is ", zhang
    zhang.configure_traits(view=view1, handler=EmployeeHandler()) ❸

```

- ❶在 Employee 模型类中，定义了 department 属性的事件处理方法_department_changed()。
- ❷覆盖标准的字符串转换方法__str__()，用以显示模型对象占据的地址^③。
- ❸为了显示对话框的标准按钮，在创建视图对象时设置 View 的 kind 和 button 参数分别为 'modal' 和 ModalButtons。这样一来，界面所显示的对话框便是“模态窗口、非即时更新”，并且有 Apply、Revert、OK、Cancel、Help 等按钮，如图 7-7 所示。

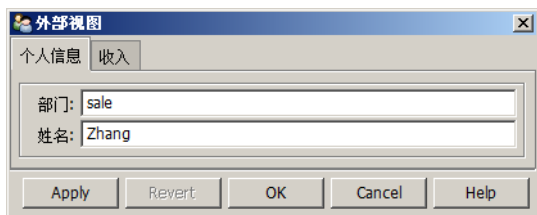


图 7-7 带标准按钮的模式对话框

❹EmployeeHandler 从 Handler 继承，它覆盖了 Handler 中的 init、init_info、position、setattr、apply、close、closed、revert 等方法。如果 close() 返回 True，窗口会被关闭；如果返回 False，就不会关闭窗口。在这些方法中，将首先调用父类中被覆盖的方法，从而实现默认的控制器的功能。实际上，父类 Handler 中的大部分方法不执行任何任务，因此也可以不运行它们，请读者阅读 Handler 类的源代码以了解每个方法的默认功能。

❺最后，创建一个控制器对象，并将它传递给 configure_traits() 的 handler 参数。

在 IPython 或命令行中运行此程序，在对话框的“部门”文本框中输入“sale”，然后单击 Apply 按钮，最后单击 OK 按钮关闭对话框。程序会在命令行窗口中输出控制器的各个方法的调用情况。

首先，对象 zhang 所表示的对象地址为“0x21794b0”：

```
zhang is <Employee at 0x21794b0>
```

③ 使用 Python 的内置函数 id() 可以获取对象的地址，地址相同的两个对象实际上是同一个对象。

调用 `zhang` 的 `configure_traits()` 之后，在窗口显示之前，调用了 `init_info()`、`init()`、`position()` 三个方法：

```
init info called
init called
position called
```

接下来输入 “sale”，每次输入一个字符，都会修改模型对象的 `department` 属性，从而调用模型对象的 `_department_changed()`，接着会调用控制器的 `setattr()`。因此控制器的 `setattr()` 是在模型数据更新之后被调用的。

```
<Employee at 0x29d60f0> department changed to s
setattr called:<Employee at 0x29d60f0>.department=s
<Employee at 0x29d60f0> department changed to sa
setattr called:<Employee at 0x29d60f0>.department=sa
<Employee at 0x29d60f0> department changed to sal
setattr called:<Employee at 0x29d60f0>.department=sal
<Employee at 0x29d60f0> department changed to sale
setattr called:<Employee at 0x29d60f0>.department=sale
```

`setattr()` 的调用参数如下：

```
setattr(self, info, obj, name, value)
```

其中，`info` 是 `UIInfo` 对象，`obj` 是被修改属性的模型对象，`name` 是被修改的属性名，而 `value` 是被修改之后的值。仔细比较前面输出的对象地址就会发现：被修改了属性的模型对象的地址并不是对象 `zhang` 的地址。这是因为“非即时更新”的对话框会对一个副本对象进行修改，在单击 `Apply` 或 `OK` 按钮时，才会将副本的内容写回原对象；而对副本对象的修改是“即时更新”的。

当单击 `Apply` 按钮时，程序输出了以下内容，我们看到：原始对象的 `department` 属性也被更新为 “sale” 了。

```
setattr called:<Employee at 0x29d60f0>.department=sale
<Employee at 0x21794b0> changed to sale
apply called
```

而 `setattr()` 再次被调用的原因是：单击 `Apply` 按钮时，“部门”文本框会失去焦点，从而引起 `setattr()` 被调用。读者可以交替单击界面中的两个文本框，观察失去焦点时的 `setattr()` 调用。由于此时模型对象的 `department` 属性没有变化，因此 `_department_changed()` 不会被调用。

最后单击 `OK` 按钮，程序输出下面两行，其中的 `True` 是 `is_ok` 参数的值，`True` 表示用户单击的是 `OK` 按钮，如果单击 `Cancel` 按钮，输出的将为 `False`：

```
close called: True
closed called: True
```

7.3.2 Controller 和 UIInfo 对象

Handler 类中每个事件处理方法的第一个参数都是 UIInfo 对象，通过它可以获得控制器对应的模型对象和视图对象所产生的界面。但是有时我们希望通过控制器的属性访问它们。TraitsUI 提供了从 Handler 继承的 Controller 类，它有两个 Trait 属性：model 和 info，分别保存模型对象和 UIInfo 对象。下面通过实例介绍 Controller 的用法以及 UIInfo 对象的内容。



traitsUI_UIInfo.py

使用 Controller 观察 UIInfo 对象

程序中的模型和视图与上一节的 “traitsUI_handler.py” 相同。创建模型对象、控制器以及显示界面的代码如下：

```
zhang = Employee(name="Zhang")
c = Controller(zhang)
c.configure_traits(view=view1)
```

在创建 Controller 控制器时使用模型对象进行初始化，之后可以通过 c.model 访问此模型对象。调用 Controller 对象的 configure_traits()，不会显示控制器本身的 Trait 属性编辑窗口，而是显示模型对象的属性编辑窗口。

在开启 “-wthread” 选项的 IPython 命令行中，执行下面的语句之后，就可以交互式地查看控制器 c 的内容了：

```
>>> run traitsUI_UIInfo.py
>>> c
>>> <enthought.traits.ui.handler.Controller object at 0x032F58D0>
```

由于无论是控制器类、视图类还是模型类，最终都是从 HasTraits 类继承，因此可以调用 get() 来快速查看其内容：

```
>>> c.get()
{'info': <enthought.traits.ui.ui_info.UIInfo object at 0x036097E0>,
 'model': <__main__.Employee object at 0x035BE9C0>}
>>> c.info.get()
{'initialized': True,
 'ui': <enthought.traits.ui.ui.UI object at 0x035BEC30>}
>>> c.info.ui.get()
[[省略]]
```

我们看到，c.info 是一个 UIInfo 对象，而 UIInfo 对象中最重要的内容就是 UI 对象 c.info.ui。

UI 对象中保存有用户界面的各种信息。对 UI 对象的详细介绍已超出本书的范围，感兴趣的读者可自行查看其源代码。下面简要地查看 UI 对象的几个属性：

```
>>> ui = c.info.ui
>>> ui.title = "hello" # 修改窗口的标题
>>> ui.context # 存储和界面相关的模型、控制器对象的字典
{'controller': <enthought.traits.ui.handler.Controller object at 0x035BECF0>,
 'handler': <enthought.traits.ui.handler.Controller object at 0x035BECF0>,
 'object': <__main__.Employee object at 0x035BE9C0>} # 模型对象
>>> ui.control # ui 对象所表示的实际界面控件，它是 wx 库的一个 Frame 对象，即窗口
<wx._windows.Frame; proxy of <Swig Object of type 'wxFrame *' at 0x33b3668> >
>>> ui.view # 创建 ui 对象的视图对象
( Group(
    Item( 'department'
        label = u'\u90e8\u95e8'
    [[省略]]
>>> ui._editors # 界面中编辑 Trait 属性用的编辑器列表
[<enthought.traits.ui.wx.text_editor.SimpleEditor object at 0x035F3A50>,
 [[省略]]
```

7.3.3 响应 Trait 属性的事件

前面曾介绍过，在从 HasTraits 继承的模型类中，可以通过定义 `_traitname_changed()` 来响应 `traitname` 属性值改变的事件。这是在模型类中响应事件，如果要在控制器类中响应，可以通过定义 `setattr()`，响应用户通过界面对模型对象的 Trait 属性进行修改的事件。

如果希望仅响应模型对象中某个特定属性的事件，可以在控制器类中定义如下格式的事件响应方法：

```
extended_traitname_changed(self, info)
```



`setattr()` 只有在通过界面修改模型对象的属性时才会被调用。但是，只要模型对象的 `traitname` 属性被修改，`extended_traitname_changed()` 就会被调用，而不管是否是通过界面进行修改。

其中的 `extended` 是视图所产生的 UI 对象的 `context` 属性中与模型对象相对应的键，通常都为 `'object'`。如果 UI 对象的 `context` 属性是由 `context` 参数指定，那么 `extended` 可以是其中的任何一个模型对象所对应的键，在实例 `“traits_multi_models.py”` 中，它可以是 `'p1'` 或 `'p2'`。

这样的事件响应方法在界面窗口初始化时，以及在对应的属性改变时都会被调用。为了区分二者，可以对 `info` 参数的 `initialized` 属性进行判断。下面是一个例子：



traitsUI_handler_event.py

在控制器中响应模型对象特定属性的事件

```
from enthought.traits.api import HasTraits, Bool
from enthought.traits.ui.api import View, Handler

class MyHandler(Handler):
    def setattr(self, info, object, name, value): ❶
        Handler.setattr(self, info, object, name, value)
        info.object.updated = True ❷
        print "setattr", name

    def object_updated_changed(self, info): ❸
        print "updated changed", "initialized=%s" % info.initialized
        if info.initialized:
            info.ui.title += "*"

class TestClass(HasTraits):
    b1 = Bool
    b2 = Bool
    b3 = Bool
    updated = Bool(False)

view1 = View('b1', 'b2', 'b3',
             handler=MyHandler(),
             title = "Test",
             buttons = ['OK', 'Cancel'])

tc = TestClass()
tc.configure_traits(view=view1)
```

❶MyHandler 类中定义了 setattr()方法，在通过界面修改了模型对象的任何一个 Trait 属性之后，它将被调用。❷在 setattr()中修改模型对象的 updated 属性为 True。

❸当模型对象的 updated 属性被修改时，它在控制器对象中对应的 object_updated_changed() 将被调用。由于 updated 属性不是通过界面修改的，因此这时 setattr()不会被再次调用。

在 IPython 下执行此程序之后，直接修改模型对象 tc 的 b1 属性：

```
>>> run traitsUI_handler_event.py
updated changed initialized=False # 初始化界面时，将调用一次
>>> tc.b1 = True # 直接修改模型对象的属性，不会调用控制器的 setattr 方法
```

在界面中选中“B2”复选框，IPython 中将输出如下内容：

```
updated changed initialized=True
setattr b2
```

程序的执行过程如图 7-8 所示。左图为界面显示之后的初始状态。如中图所示，直接修改模型对象的属性不会触发控制器的 `setattr()` 运行，但是“B1”复选框会改为被选中；当通过界面选中“B2”复选框之后，将调用控制器的 `setattr()`，从而设置模型对象的 `update` 属性为 `True`，这会再次触发控制器的 `object_updated_changed()` 运行，从而在窗口标题中添加“*”。

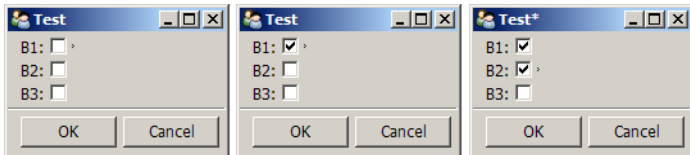


图 7-8 界面初始状态(左)，直接修改 `tc.b1` 之后(中)，选中“B2”复选框之后(右)

7.4 属性编辑器

每个 `Trait` 类型都有一种默认的界面编辑器(控件)与之对应，如果在视图对象中不指定编辑器，将使用默认的编辑器生成界面。每种编辑器都可以有 4 种样式：“simple”、“custom”、“text”和“readonly”：

- “simple”：默认值，使用一个比较简单的编辑器，尽量少占用界面空间。
- “custom”：使用较复杂的编辑器，尽量呈现更多的内容。
- “text”：使用一个文本编辑器。
- “readonly”：使用只读控件显示。

TraitsUI 提供了丰富的编辑器库，以至于我们很少有自己设计编辑器的需求。由于 TraitsUI 的编辑器种类繁多，本书不能一一对其进行详细介绍，感兴趣的读者可运行 `Python(x,y)` 的文档目录下的演示程序，运行界面如图 7-9 所示。



c:\pythonxy\doc\Enthought Tool Suite\Traits\examples\demo\demo.py

TraitsUI 演示程序

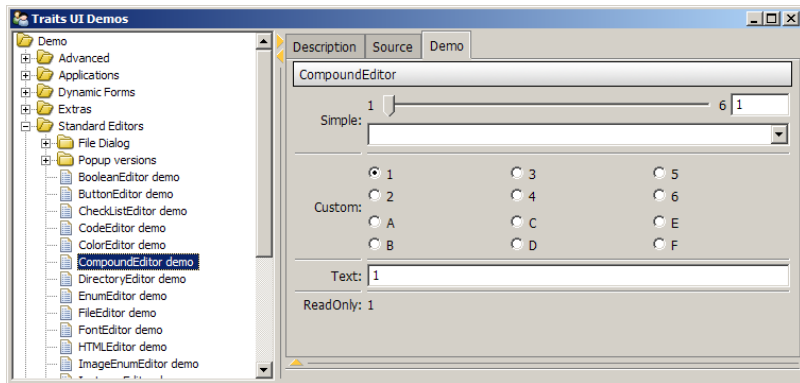


图 7-9 TraitsUI 演示程序的运行界面

下面将以几个实例简单地介绍如何使用 TraitsUI 提供的编辑器。

7.4.1 编辑器演示程序

由于 TraitsUI 提供了众多的编辑器，并且每个编辑器又有许多属性配置，对它们一一进行介绍将是一件枯燥无味的事情。因此本节介绍一个能显示各种编辑器效果的演示程序，图 7-10 是它的界面截图。界面的左半部分是用来创建各种 Trait 属性的源程序列表，对于选中的某个 Trait 属性，在界面的右半部分将使用"simple"、"custom"、"text"和"readonly"4 种样式创建属性编辑器。

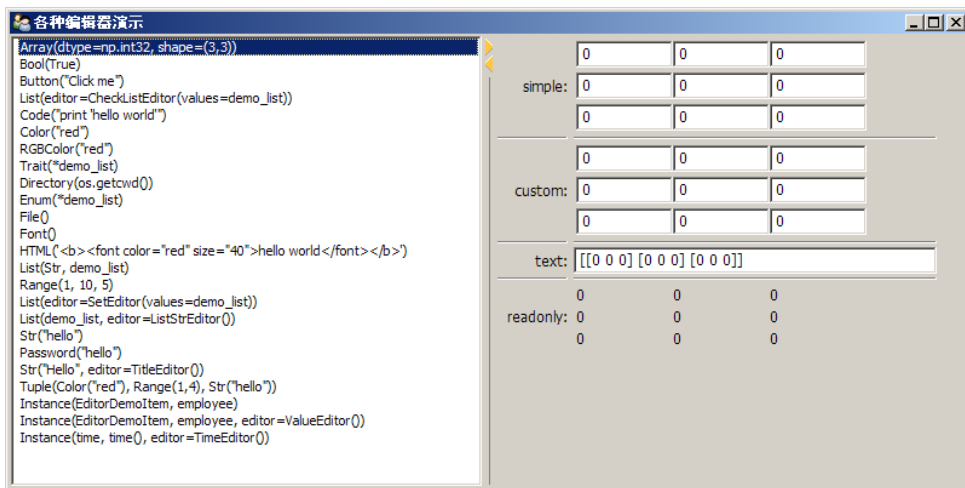


图 7-10 各种编辑器的效果演示



traitsUI_editors.py

各种编辑器的效果演示

```
class EditorDemo(HasTraits):
    codes = List(Str)
    selected_item = Instance(EditorDemoItem)
    selected_code = Str ❶
    view = View(
        HSplit(
            Item("codes", style="custom", show_label=False,
                editor=ListStrEditor(editable=False, selected="selected_code")), ❶
            Item("selected_item", style="custom", show_label=False),
        ),
        resizable=True,
        width = 800,
        height = 400,
        title=u"各种编辑器演示"
```

```

)

def _selected_code_changed(self):
    item = EditorDemoItem(code=self.selected_code)
    item.add_trait("item", eval(self.selected_code)) ❷
    self.selected_item = item

```

上面是创建主界面的程序。在 `EditorDemo` 类中，`codes` 属性保存一组用来创建各种 `Trait` 属性的字符串，`selected_item` 属性是 `EditorDemoItem` 的对象。在 `EditorDemo` 类的视图定义中，使用 `HSplit` 将 `codes` 和 `selected_items` 所对应的编辑器进行水平分割。❶用 `editor` 参数设置 `codes` 属性的编辑器为 `ListStrEditor`，它是一个显示一组字符串的列表框控件。其 `editable` 属性为 `False`，表示列表框中的字符串都是只读的。`selected` 属性是保存被选中的字符串的 `Trait` 属性名，在 `EditorDemo` 类中用 `selected_code` 属性保存列表框中被选中的字符串。可以通过 `editor` 参数设置 `Item` 对象的编辑器，这样界面中将使用指定的编辑器显示 `Trait` 属性。

❷当用户通过列表框选中了某个字符串时，`selected_code` 属性将发生变化，因此 `_selected_code_changed()` 将被调用。这里创建一个 `EditorDemoItem` 对象，并调用其 `add_trait()` 方法，动态地为其创建一个名为 `item` 的 `Trait` 属性，其类型则通过 `eval()` 对 `selected_code` 字符串进行求值得到。这段代码通过字符串动态地创建 `Trait` 类型，并使用此类型动态地创建 `Trait` 属性。

```

class EditorDemoItem(HasTraits):
    code = Code()
    view = View(
        Group(
            Item("item", style="simple", label="simple", width=-300), ❶
            "_", ❷
            Item("item", style="custom", label="custom"),
            "_",
            Item("item", style="text", label="text"),
            "_",
            Item("item", style="readonly", label="readonly"),
        ),
    )

```

在 `EditorDemoItem` 类的视图定义中，我们使用 4 种样式为其 `item` 属性定义编辑器。请注意在 `EditorDemoItem` 类的定义中并没有 `item` 属性，但是由于视图使用属性名字符串定义编辑器，因此只有在真正使用视图创建界面时，才会去访问 `item` 属性，这时已经通过 `add_trait()` 为其添加了 `item` 属性。❶ `Item` 对象的 `width` 属性可以指定编辑器的宽度，以像素点为单位的长度用整数表示，负数表示强制设置其宽度。`width` 属性还有多种设置宽度的用法，请读者查看 `Item` 类源代码中的注释。❷使用下划线字符串在界面中创建分割线。

```
employee = Employee()
```



```

demo_list = [u"低通", u"高通", u"带通", u"带阻"]

trait_defines = """
    Array(dtype="int32", shape=(3,3))
    Bool(True)
    Button("Click me")
    List(editor=CheckListEditor(values=demo_list))
    Code("print 'hello world'")
    Color("red")
    RGBColor("red")
    Trait(*demo_list)
    Directory(os.getcwd())
    Enum(*demo_list)
    File()
    Font()
    HTML('<b><font color="red" size="40">hello world</font></b>')
    List(Str, demo_list)
    Range(1, 10, 5)
    List(editor=SetEditor(values=demo_list))
    List(demo_list, editor=ListStrEditor())
    Str("hello")
    Password("hello")
    Str("Hello", editor=TitleEditor())
    Tuple(Color("red"), Range(1,4), Str("hello"))
    Instance(EditorDemoItem, employee)
    Instance(EditorDemoItem, employee, editor=ValueEditor())
    Instance(time, time(), editor=TimeEditor())
"""

demo = EditorDemo()
demo.codes = [s.split("#")[0].strip() for s in trait_defines.split("\n") if s.strip()!=""]
demo.configure_traits()

```

最后是定义各种 Trait 类型的程序。我们可以在定义 Trait 类型时，通过 editor 参数设置其对应的编辑器，这样就不需要在视图的 Item 对象中定义了。

请读者自行研究每个 Trait 类型的定义以及它们所创建的界面控件，这里就不再进行详细说明了。

7.4.2 对象编辑器

随着程序开发的进行，界面中的控件数目会逐渐增多，功能会越来越复杂，这意味着与界面对应的模型类也会变得复杂起来。为了便于代码的理解、管理以及重用，我们需要对模型类及其对应的界面视图对象进行重构。将程序中重复使用、相对独立的部分作为组件分离出来，单独为其设计模型类和视图对象，最终的应用程序将由一系列这样的组件构成。这些组件可以

在程序的不同地方重复使用，从而起到功能分离、代码重用等多方面的作用。TraitsUI 的 MVC 模式非常适合这种组件开发方式，下面让我们通过一些实例深入理解 MVC 模式所带来的便利。



traitsUI_component.py

组件演示程序

此实例程序将创建一个如图 7-11 所示的界面，用户可以通过上方的下拉列表框选择一种形状，下方的控件会自动根据所选的形状发生变化，当通过这些控件输入形状数据时，界面下方的信息栏会自动进行更新。由于程序较长，下面我们将它分为几个部分进行分析。

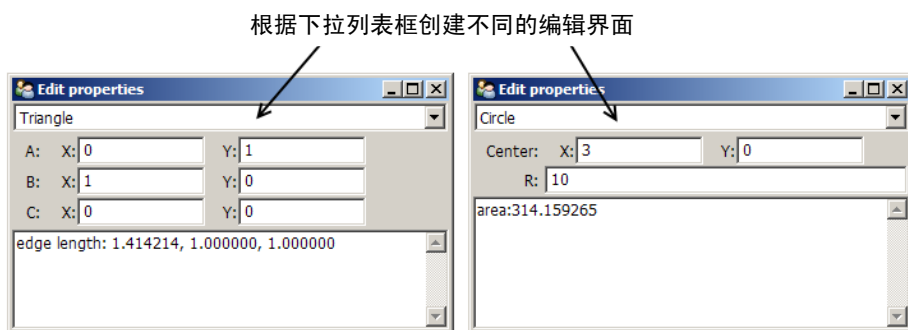


图 7-11 组件演示，根据下拉列表框创建不同的编辑界面

```
class Point(HasTraits):
    x = Int
    y = Int
    view = View(HGroup(Item("x"), Item("y")))
```

上面的程序定义了用于保存平面上点的坐标的 `Point` 类。我们还为它指定了一个视图对象，视图中 `X` 和 `Y` 轴的坐标值输入框是横向排列的。在 IPython 中运行下面的语句即可看到 `Point` 对象所创建的界面效果：

```
>>> Point().configure_traits()
```

我们可以将 `Point` 类当做组件使用，将它嵌入到更复杂的界面中去。在下面的程序中定义了一个基类 `Shape`，以及两个派生类 `Triangle` 和 `Circle`，并使用 `Point` 类定义所有表示二维坐标点的属性：

```
class Shape(HasTraits):
    info = Str ❶

    def __init__(self, **traits):
        super(Shape, self).__init__(**traits)
        self.set_info() ❷
```

```

class Triangle(Shape):
    a = Instance(Point, ()) ❸
    b = Instance(Point, ())
    c = Instance(Point, ())

    view = View(
        VGroup(
            Item("a", style="custom"), ❹
            Item("b", style="custom"),
            Item("c", style="custom"),
        )
    )

    @on_trait_change("a.[x,y],b.[x,y],c.[x,y]")
    def set_info(self):
        a,b,c = self.a, self.b, self.c
        l1 = ((a.x-b.x)**2+(a.y-b.y)**2)**0.5
        l2 = ((c.x-b.x)**2+(c.y-b.y)**2)**0.5
        l3 = ((a.x-c.x)**2+(a.y-c.y)**2)**0.5
        self.info = "edge length: %f, %f, %f" % (l1,l2,l3)

class Circle(Shape):
    center = Instance(Point, ())
    r = Int

    view = View(
        VGroup(
            Item("center", style="custom"),
            Item("r"),
        )
    )

    @on_trait_change("r")
    def set_info(self):
        self.info = "area:%f" % (pi*self.r**2)

```

❶在 Shape 类中定义了一个 info 属性, ❷在初始化方法中调用派生类的 set_info() 以修改 info 属性。

❸在 Triangle 类中使用 “Instance(Point, ())” 定义了表示三角形三个顶点坐标的属性: a、b 和 c。在 Circle 类中使用同样的方式定义了表示圆心坐标的 center 属性。这些属性的值都是 Point 对象。Instance 的第二个参数指定创建默认对象时所用的参数, 当没有第二个参数时, 它所定义的属性的默认值为 None, 这里用一个空元组表示与其对应的属性的默认值是通过调用 “Point()” 得到的, 即默认为 Point() 所创建的 Point 对象。

❹如果 Trait 属性是 Instance 类型, 并且它在视图中对应的编辑器为 “custom” 样式, 那么属

性对象的视图将直接嵌入到当前的视图中。因此，在 `Triangle` 和 `Circle` 对象的编辑界面中将嵌入多个 `Point` 对象的编辑器。在 `IPython` 中运行下面的程序可以看到所创建的界面效果：

```
>>> Triangle().configure_traits()
>>> Circle().configure_traits()
```

接下来，使用上面的形状类创建最终的形状选择类 `ShapeSelector`：

```
class ShapeSelector(HasTraits):
    select = Enum(*[cls.__name__ for cls in Shape.__subclasses__()]) ❶
    shape = Instance(Shape) ❷

    view = View(
        VGroup(
            Item("select"),
            Item("shape", style="custom"), ❸
            Item("object.shape.info", style="custom"), ❹
            show_labels = False
        ),
        width = 350, height = 300, resizable = True
    )

    def __init__(self, **traits):
        super(ShapeSelector, self).__init__(**traits)
        self._select_changed()

    def _select_changed(self):
        klass = [c for c in Shape.__subclasses__() if c.__name__ == self.select][0]
        self.shape = klass() ❺
```

❶ 下拉列表框所对应的 `select` 属性的类型为 `Enum`(枚举)，为了让程序显得更自动化一些，我们不直接指定枚举类型的候选值，而是通过 `Shape` 的派生类名创建候选值列表。这样一来，当我们添加其他的 `Shape` 派生类时，便不需要对这段代码进行任何修改。

❷ `shape` 属性的类型是 `Shape`，由于这里不需要创建默认的 `Shape` 对象，因此不用指定 `Instance` 的第二个参数。❸ 当 `select` 属性发生变化时，在其事件处理方法 `_select_changed()` 中，创建 `select` 属性所对应的类的对象并赋值给 `shape` 属性。❹ `shape` 属性所对应的编辑器是 "custom" 样式，因此将它的编辑界面作为组件嵌入到 `ShapeSelector` 的界面中。并且它能根据当前的 `shape` 属性值，动态更新界面上的编辑器。也就是说，当 `shape` 属性是 `Triangle` 对象时，将使用 `Triangle` 类的视图创建编辑器；而当 `shape` 属性是 `Circle` 对象时，将使用 `Circle` 类的视图创建编辑器。

❺ 通过 "object.shape.info" 可以为 `shape` 属性的 `info` 属性在界面中创建编辑器。当为某个 `Trait` 属性的属性创建编辑器时，需要在属性名的前面添加 "object."。

读者也许会认为这种为属性的属性创建编辑器的做法有些混乱，一个比较简单的解决方法

就是从 ShapeSelector 的视图中删除"object.shape.info"的 Item 对象，并分别给 Triangle 和 Circle 的视图添加显示 info 属性的编辑器：Item("info", style="custom")。这种做法的缺点是，需要给每个从 Shape 派生的类的视图添加 info 属性的编辑器，而当我们不想显示 info 属性时，代码的修改量也会随着 Shape 的派生类的增加而增加。

还有一种使用多个视图对象的方法，它充分体现了 MVC 模式将模型和视图完全分离的优点。



traitsUI_component_multi_view.py
使用多个视图显示组件

由于程序的改动不大，我们只介绍它和“traitsUI_component.py”的不同之处。

```
class Shape(HasTraits):
    info = Str
    view_info = View(Item("info", style="custom", show_label=False))
```

首先，为 Shape 类添加一个 view_info 视图，专门用于显示其 info 属性。这样一来，Shape 的派生类 Triangle 和 Circle 便都具有两个视图：view、view_info。如果模型类有多个视图，那么当将其嵌入到其他视图中时，需要指定使用哪个视图创建编辑器。因此，ShapeSelector 类的视图需要进行如下修改：

```
view = View(
    VGroup(
        Item("select", show_label=False),
        VSplit( ❶
            Item("shape", style="custom", editor=InstanceEditor(view="view")), ❷
            Item("shape", style="custom", editor=InstanceEditor(view="view_info")),
            show_labels = False
        )
    ),
    width = 350, height = 300, resizable = True
)
```

❶为了和前面的例子有所区别，这里用一个垂直分隔容器将形状数据输入界面和显示形状信息的控件分隔开。❷shape 属性的编辑器样式仍然为"custom"，但是为了指定编辑器所使用的视图，我们需要通过 editor 参数传递一个 InstanceEditor 对象。而通过 InstanceEditor 对象的 view 参数可以指定创建界面时所使用的视图名。实际上，Instance 类型的 Trait 属性默认就是使用 InstanceEditor 作为"custom"样式的编辑器，因此前面的程序中都没有通过 editor 参数指定。当需要修改 InstanceEditor 对象的一些默认值时，就需要我们手工创建它了。

下面总结一下前面的内容：

- 通过将 Instance 类型的 Trait 属性的编辑器样式指定为"custom", 可以实现界面的层层嵌套, 即组件功能。
- 当模型类有多个视图对象时, 通过 InstanceEditor 的 view 参数可以选择其中的某个视图来创建编辑此模型对象的控件。

TraitsUI 的组件并不局限于界面中的某一块区域, 我们可以在界面中的不同位置用不同的视图, 为同一个模型对象创建多个不同的编辑器, 因此使用 TraitsUI 创建的界面是非常灵活的。

7.4.3 字符串列表编辑器

在 Traits 库中, List 是表示列表的 Trait 类型。根据列表元素的类型, 可以使用不同的编辑器显示其内容。让我们首先从最简单的字符串列表开始。下面的程序使用了三种不同的编辑器来显示字符串列表, 界面如图 7-12 所示。

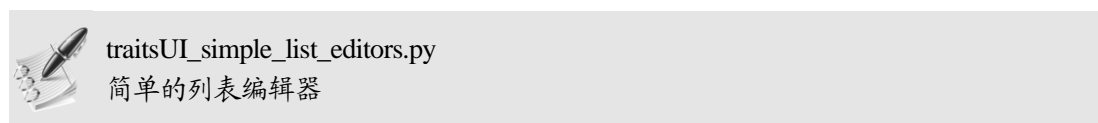


图 7-12 使用三种编辑器显示简单列表属性

```
filter_types = [u"低通", u"高通", u"带通", u"带阻"] ❶
class ListDemo(HasTraits):

    items = List(Str) ❷
    view = View(
        HSplit(
            Item("items", style="custom", show_label=False), ❸
            Item("items", style="custom",
                editor=CheckListEditor(values=filter_types, cols=4)), ❹
            Item("items", editor=SetEditor(values=filter_types)), ❺
            show_labels=False
        ),
        resizable=True,
        width = 600,
        title=u"简单列表编辑器演示"
    )
```

❶首先创建一个列表 `filter_types`，它表示列表属性中元素的可能取值。❷定义 `items` 属性为字符串列表类型，这里的 `Str` 表示列表的元素类型必须是字符串。

❸使用默认的列表编辑器显示 `items` 属性，效果如图 7-12 左侧的编辑器所示。列表的每个元素使用一个文本编辑器进行编辑，在编辑器的左侧有一个蓝色的图标。单击此图标会弹出一个添加、删除和改变元素顺序的菜单。使用此编辑器可以为 `items` 属性添加任意字符串，不受候选值列表 `filter_types` 的限制。此外，指定 `Item` 对象的 `style` 参数为 `"custom"`，让列表编辑器能够完全显示其内容。

❹通过 `editor` 参数指定编辑器为 `CheckListEditor` 对象。在创建 `CheckListEditor` 编辑器对象时，通过 `values` 和 `cols` 参数分别指定候选值列表和显示的列数。

`CheckListEditor` 编辑器会根据对应的 `Item` 对象的 `style` 属性来决定所使用的控件。这里选用 `"custom"` 样式，将使用复选框控件作为编辑器。如果样式为默认的 `"simple"`，将会用下拉列表框作为编辑器。下拉列表框只能进行单选，因此它所对应的列表属性只能有一个元素，请读者修改演示程序以自行验证。显示效果如图 7-12 中间的编辑器所示。

❺使用 `SetEditor` 作为编辑器，由于 `SetEditor` 的 `"simple"` 和 `"custom"` 样式效果相同，因此不需要设置 `style` 属性。效果如图 7-12 右侧的编辑器所示。

使用其中任意一个编辑器修改属性值，其他的编辑器也会同时更新。由于 `CheckListEditor` 和 `SetEditor` 编辑器将 `items` 属性的内容锁定为有限的候选项，因此无法通过最左边的默认编辑器添加其他的字符串。

在上面的例子中，`items` 属性的候选值是固定的，在定义 `ListDemo` 类时，这些候选值就已经被确定，以后也无法改变。但是有时我们希望候选列表本身也可以动态地改变。为了实现这个目的，可以使用另外一个 `Trait` 属性保存候选列表：

```
class ListDemo2(HasTraits):
    filter_types = List(Str, value=[u"低通", u"高通", u"带通", u"带阻"]) ❶
    items = List(Str)
    view = View(
        HGroup(
            Item("filter_types", label=u"候选"), ❷
            Item("items", style="custom",
                editor=CheckListEditor(name="filter_types")), ❸
            show_labels=False
        ),
        resizable=True,
        width = 300,
        height = 180,
        title=u"动态修改候选值"
    )
```

❶定义一个 `filter_types` 属性来保存候选值列表，通过 `value` 参数指定初始值。❷在视图中使用默认的列表编辑器显示候选值属性，用户通过它可以动态地修改候选值列表的内容。❸在

创建 CheckListEditor 编辑器对象时, 使用 name 参数指定候选值列表的属性名。图 7-13 是程序的运行效果。在此界面中, 我们通过左边的列表编辑器添加了一个候选值: “均衡器”, 右边的 CheckListEditor 编辑器的内容也同时发生了变化。

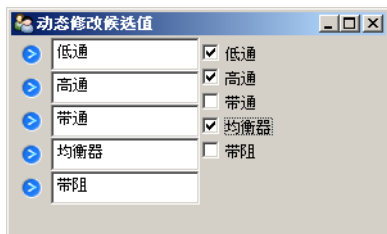


图 7-13 动态更改候选值列表

7.4.4 对象列表编辑器

当列表属性的元素是 HasTraits 对象时, 还可以使用表格或标签页作为它的编辑器, 效果如图 7-14 所示。

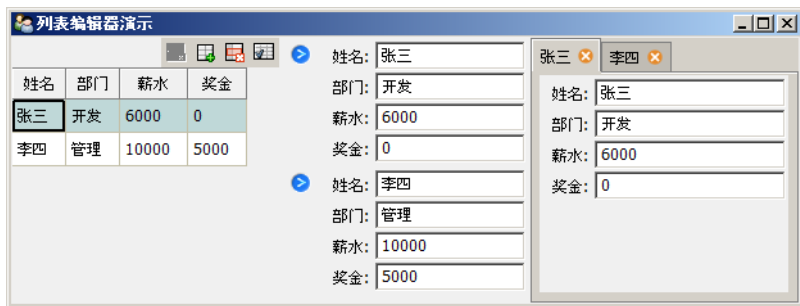


图 7-14 使用三种编辑器显示对象列表属性: 表格(左)、列表(中)、标签页(右)



TraitsUI_list_editors.py

使用三种编辑器显示对象列表属性

```
from enthought.traits.api import HasTraits, Unicode, Int, List, Instance
from enthought.traits.ui.api import View, Item, TableEditor, ListEditor, HGroup
from enthought.traits.ui.table_column import ObjectColumn

class Employee(HasTraits):
    name = Unicode(label=u"姓名") ❶
    department = Unicode(label=u"部门")
    salary = Int(label=u"薪水")
    bonus = Int(label=u"奖金")
    view = View("name", "department", "salary", "bonus") ❷

table_editor = TableEditor( ❸
```



```

columns = [
    ObjectColumn(name="name", label=u"姓名"),
    ObjectColumn(name="department", label=u"部门"),
    ObjectColumn(name="salary", label=u"薪水"),
    ObjectColumn(name="bonus", label=u"奖金")
],
row_factory = Employee,
deletable = True,
show_toolbar = True)

list_editor = ListEditor(style="custom") ❹
tab_editor = ListEditor(use_notebook=True, deletable=True, page_name=".name") ❺

class EmployeeList(HasTraits):
    employees = List(Instance(Employee, factory=Employee)) ❸
    view = View(
        HGroup(
            Item("employees", editor=table_editor), ❻
            Item("employees", editor=list_editor),
            Item("employees", style="custom", editor=tab_editor),
            show_labels=False
        ),
        resizable=True,
        width = 600,
        title=u"列表编辑器演示"
    )

```

首先，Employee 是列表中对象的类型。在前面的例子中，界面中的标签在 Item 对象中定义。❶这里演示了另外一种定义标签的方法，可以在定义各个 Trait 属性时，使用 label 参数指定它们在视图中的标签。❷由于标签已经在 Trait 属性中定义，因此视图中可以不使用 Item，而是直接使用各个属性名的字符串。

在 EmployeeList 类中，employees 属性是一个 Employee 对象的列表。❸我们用 Instance(Employee)指定列表的元素类型为 Employee 对象。通过 Instance 的 factory 参数可以指定创建列表的默认对象的工厂函数，这里直接使用 Employee 类，也可以通过传递一个空元组来表示使用对应的类创建默认对象。此工厂函数将在图 7-14(中)的列表编辑器中用来给 employees 属性添加新的 Employee 对象。❹在视图中，我们使用三种编辑器显示 employees 属性。

❺用 TableEditor 创建表格编辑器，它的 columns 属性是一个 ObjectColumn 对象的列表，每个 ObjectColumn 对象描述模型对象中与其对应的 Trait 属性。row_factory 属性用于在表格中创建新的对象，这里也直接使用 Employee 类。deletable 属性为 True，表示可以删除表格中的行。show_toolbar 属性为 True，表示显示表格的工具栏，通过工具栏可以删除或添加行。

❻当 ListEditor 编辑器的 style 属性为"custom"时，列表中的每个模型对象都将使用与其对

应的视图来创建编辑器。⑦当 ListEditor 编辑器的 use_notebook 属性为 True 时，将使用标签页显示列表中的多个模型对象。deletable 属性为 True，表示可以通过关闭标签页来从列表中删除模型对象。page_name 属性指定标签页标题所对应的 Trait 属性名，注意属性名的前面有一个“.”。



标签页的标题如果输入中文可能会出现错误，请按照下面的说明修改 TraitsUI 库的源代码。

本书写作时采用的 TraitsUI 库的版本为 3.6，如果在标签页标题中输入中文，会出现错误，这是因为 TraitsUI 中还有些代码对 Unicode 的支持不够，希望日后会有所改善。目前可以通过分析错误提示信息，修改 TraitsUI 库中相应的源程序。在下面的文件中搜索“??? ”：

site-packages\traitsbackendwx-3.6.0-py2.6.egg\enthought\traits\ui\wx\list_editor.py

将相应行的“str”改为“unicode”，有两处需要修改：

```
# 第一处
if name is None:
    name = str( xgetattr( view_object, # str 改为 unicode
                        self.factory.page_name[1:], '???' ) )

# 第二处
name = str( name ) or '???' # str 改为 unicode
```

7.5 菜单、工具条和状态栏

菜单、工具条和状态栏是窗口应用程序的标准组件，它们可以通过 View 对象的 menubar、toolbar 和 statusbar 属性指定。下面的程序演示了这一过程，效果如图 7-15 所示。



traitsUI_menu_toolbar.py

为界面添加菜单、工具条和状态栏

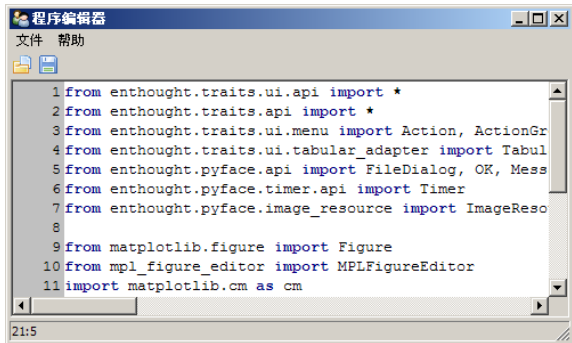


图 7-15 为界面添加菜单、工具条和状态栏

```

from enthought.traits.api import HasTraits, Code, Str, Int, on_trait_change
from enthought.traits.ui.api import View, Item, Handler, CodeEditor
from enthought.traits.ui.menu import Action, ActionGroup, Menu, MenuBar, ToolBar
from enthought.pyface.image_resource import ImageResource

class MenuDemoHandler(Handler):
    def exit_app(self, info):
        info.ui.control.Close()

class MenuDemo(HasTraits):
    status_info = Str
    current_line = Int
    text = Code
    def traits_view(self):
        file_menu = Menu( ❷
            ActionGroup(
                Action(id="open", name=u"打开", action="open_file"),
                Action(id="save", name=u"保存", action="save_file"),
            ),
            ActionGroup(
                Action(id="exit_app", name=u"退出", action="exit_app"),
            ),
            name = u"文件"
        )

        about_menu = Menu(
            Action(id="about", name=u"关于", action="about_dialog"),
            name = u"帮助"
        )

        tool_bar = ToolBar( ❸
            Action(
                image = ImageResource("folder_page.png", search_path = ["img"]),
                tooltip = u"打开文档",
                action = "open_file"
            ),
            Action(
                image = ImageResource("disk.png", search_path = ["img"]),
                tooltip = u"保存文档",
                action = "save_file"
            ),
        )

        return View(
            Item("text", style="custom", show_label=False,

```

```

        editor=CodeEditor(line="current_line")),
        menubar = MenuBar(file_menu, about_menu), ❶
        toolbar = tool_bar,
        statusbar = ["status_info"], ❷
        resizable = True,
        width = 500, height = 300,
        title = u"程序编辑器",
        handler = MenuDemoHandler()
    )

    @on_trait_change("text,current_line")
    def update_status(self):
        self.status_info = "%d:%d" % (self.text.count("\n")+1, self.current_line)

    def open_file(self):
        print "open_file"

    def save_file(self):
        print "save_file"

    def about_dialog(self):
        print "about_dialog"

demo = MenuDemo()
demo.configure_traits()

```



当视图的定义比较复杂时，可以定义名为 `traits_view` 的方法来返回视图对象。

❶ 菜单栏是一个 `MenuBar` 对象，它由多个 `Menu` 对象组成。❷ 每个 `Menu` 对象定义菜单栏中的一个菜单。在 `Menu` 对象中，使用 `Action` 对象定义菜单中的每个选项，并且可以使用 `ActionGroup` 对象对菜单进行分组。`Action` 对象的 `name` 属性是菜单上的文字，`action` 属性是对应的事件处理方法名，`id` 属性为其唯一标识。`Action` 类还有许多其他的属性，读者可以在 IPython 中输入如下命令来查看 `Action` 及其父类的源代码，以了解更多的属性用法：

```

>>> Action??
>>> Action.__base__??

```

❸ 工具条是一个 `ToolBar` 对象，它由多个 `Action` 对象组成，每个 `Action` 对象对应工具条中的一个按钮。通过 `Action` 对象的 `image` 属性可以指定工具按钮上显示的图标。为了方便指定图标文件的路径，我们使用 `ImageResource` 管理图标资源。

当用户使用菜单栏和工具条时，将运行 `Action` 对象的 `action` 属性所指定的事件处理方法。事件处理方法可以在模型类中定义，也可以在控制器类中定义。在上面的例子中，`open_file()`、`save_file()`、`about_dialog()` 等在模型类中定义，而 `exit_app()` 在控制器类中定义。因为在 `exit_app()`

中，需要调用后台界面库中窗口对象的 `Close()` 方法来关闭应用程序。

④状态栏则可以直接用一个字符串列表来指定，此列表中的每个字符串都是一个 `Trait` 属性名。在状态栏中将实时地显示对应的 `Trait` 属性值。

7.6 设计自己的编辑器

除了使用 `TraitsUI` 提供的标准编辑器之外，我们还可以使用自己编写的编辑器。这样可以 根据应用程序的需求，制作出更加专业的界面。本节简要介绍 `Trait` 编辑器的工作原理，并且制作一个封装 `matplotlib` 绘图控件的编辑器，最后用它制作一个对 `CSV` 数据文件进行绘图的小工具。

Traits 库的路径

如果读者想深入了解 `TraitsUI` 的工作原理，需要查看它的源程序，因此首先需要知道它们在哪里。下面所有的路径都在 `site-packages` 目录下。

Traits: `Traits-x.x.x-py2.6-win32.egg\enthought\traits`，以下简称“%traits%”。

TraitsUI: `Traits-x.x.x-py2.6-win32.egg\enthought\traits\ui`，以下简称“%ui%”。

wx 后台界面库: `TraitsBackendWX-x.x.x-py2.6.egg\enthought\traitsui\wx`，以下简称“%wx%”。

7.6.1 Trait 编辑器的工作原理

先看下面的小程序，它定义了一个 `TestStrEditor` 类，其中有一个名为 `test` 的 `Trait` 属性，类型为 `Str`。在视图 `view` 中定义一个 `Item` 对象以显示 `test` 属性，但是没有通过 `editor` 参数指定它所使用的编辑器。当显示界面时，`Traits` 库将自动挑选文本框控件作为 `test` 属性的编辑器。



traitsUI_texteditor.py

字符串属性的默认编辑器

```
class TestStrEditor(HasTraits):
    test = Str
    view = View(Item("test"))
```

通过 `Trait` 类型对象的 `create_editor()` 方法可以获得它的默认编辑器，例如：

```
>>> t = TestStrEditor()
>>> ed = t.trait("test").trait_type.create_editor()
>>> type(ed)
<class 'enthought.traits.ui.editors.text_editor.ToolkitEditorFactory'>
```

这个编辑器对象 `ed` 的类也是从 `HasTraits` 继承，因此可以调用 `get()` 来显示它所有的 `Trait` 属性名和对应值：

```
>>> ed.get()
{'auto_set': True,
 'custom_editor_class': <class 'enthought.traits.ui.wx.text_editor.CustomEditor'>,
 'enabled': True,
 'enter_set': False,
 'evaluate': <enthought.traits.ui.editors.text_editor._Identity object at 0x0427F1B0>,
 'evaluate_name': '',
 'format_func': None,
 'format_str': '',
 'invalid': '',
 'is_grid_cell': False,
 'mapping': {},
 'multi_line': True,
 'password': False,
 'readonly_editor_class': <class 'enthought.traits.ui.wx.text_editor.ReadOnlyEditor'>,
 'simple_editor_class': <class 'enthought.traits.ui.wx.text_editor.SimpleEditor'>,
 'text_editor_class': <class 'enthought.traits.ui.wx.text_editor.SimpleEditor'>,
 'view': None}
```

`create_editor()` 的源代码可以在 “%traits%\trait_types.py” 中的 `BaseStr` 类的定义中找到。`create_editor()` 得到的是一个 `text_editor.ToolkitEditorFactory` 类，它的完整模块路径为：

```
enthought.traits.ui.editors.text_editor.ToolkitEditorFactory
```

在 “%ui%\editors\text_editor.py” 中可以找到它的定义，它从 `EditorFactory` 继承，而 `EditorFactory` 类的代码在 “%ui%\editor_factory.py” 中。`EditorFactory` 类是 `Trait` 编辑器的核心，所有的编辑器类都通过它和后台界面库关联起来。让我们详细看看 `EditorFactory` 类中关于控件生成方面的代码：

```
class EditorFactory ( HasPrivateTraits ):
    # 下面 4 个属性描述 4 个类型的编辑器类
    simple_editor_class = Property ❶
    custom_editor_class = Property
    text_editor_class  = Property
    readonly_editor_class = Property

    # 用 simple_editor_class 创建实际的控件
    def simple_editor ( self, ui, object, name, description, parent ):
        return self.simple_editor_class( parent,
                                         factory      = self,
```

```

        ui          = ui,
        object       = object,
        name         = name,
        description  = description )

# 这是类的方法，它通过搜索当前类和父类，自动找到与其匹配的后台界面库中的控件类
@classmethod
def _get_toolkit_editor(cls, class_name): ❷
    editor_factory_classes = [factory_class for factory_class in cls.mro()
                              if issubclass(factory_class, EditorFactory)]
    for index in range(len( editor_factory_classes )):
        try:
            factory_class = editor_factory_classes[index]
            editor_file_name = os.path.basename(
                sys.modules[factory_class.__module__].__file__)
            return toolkit_object('.'.join([editor_file_name.split('.')[0], ❸
                                             class_name])), True)

        except Exception, e:
            if index == len(editor_factory_classes)-1:
                raise e
    return None

# simple_editor_class 属性的 get 方法，获取属性值
def _get_simple_editor_class(self):
    try:
        SimpleEditor = self._get_toolkit_editor('SimpleEditor')
    except:
        SimpleEditor = toolkit_object('editor_factory:SimpleEditor')
    return SimpleEditor

```

❶EditorFactory 对象有 4 个属性，分别用来保存后台界面库中的 4 种样式的编辑器类：simple_editor_class、custom_editor_class、text_editor_class 和 readonly_editor_class。在前面的例子中，ed 对象的 simple_editor_class 属性为：

```

>>> ed.simple_editor_class
<class 'enthought.traits.ui.wx.text_editor.SimpleEditor'>

```

我们看到：它用的是 wx 后台界面库的 text_editor 模块中的 SimpleEditor 类，稍后将详细查看它的代码。

❷EditorFactory 类通过类方法_get_toolkit_editor()获得后台界面库中的类。由于是类方法，它的第一个参数 cls 就是 EditorFactory 类本身。当调用它的派生类的_get_toolkit_editor()时，第一个参数 cls 就是派生类本身。通过调用 cls.mro()可以获得 cls 及其所有父类，然后一个一个地

查找，从后台界面库中找到与之匹配的类，这个工作由 `toolkit_object()` 完成，其源代码在“%ui%\toolkit.py”中。

在后台界面库中，类的组织结构和 TraitsUI 库中的完全一样，因此不需要额外的配置文件，只需要几个字符串替代操作就可以将 TraitsUI 库中 `EditorFactory` 的派生类和后台界面库中实际的编辑器类关联起来。图 7-16 显示了 TraitsUI 库中 `EditorFactory` 和后台界面库的关系。



图 7-16 TraitsUI 库中 `EditorFactory` 和后台界面库的关系

后台界面库中定义了所有编辑器的控件，例如在“%wx%\text_editor.py”中可以找到产生文本框控件的类 `text_editor.SimpleEditor`。类名表示控件的 4 种样式——“simple”、“custom”、“text”和“readonly”，文件名(模块名)则表示控件的类型。下面是 `text_editor.SimpleEditor` 的部分代码：

```
class SimpleEditor ( Editor ):

    # Flag for window styles:
    base_style = 0

    # Background color when input is OK:
    ok_color = OKColor

    # Function used to evaluate textual user input:
    evaluate = evaluate_trait

    def init ( self, parent ):
        """ Finishes initializing the editor by creating the underlying toolkit
            widget.
        """
        factory      = self.factory
        style        = self.base_style
        self.evaluate = factory.evaluate
        self.sync_value( factory.evaluate_name, 'evaluate', 'from' )

        if (not factory.multi_line) or factory.password:
            style &= ~wx.TE_MULTILINE

        if factory.password:
            style |= wx.TE_PASSWORD
```



```

multi_line = ((style & wx.TE_MULTILINE) != 0)
if multi_line:
    self.scrollable = True

if factory.enter_set and (not multi_line):
    control = wx.TextCtrl( parent, -1, self.str_value,
                           style = style | wx.TE_PROCESS_ENTER )
    wx.EVT_TEXT_ENTER( parent, control.GetId(), self.update_object )
else:
    control = wx.TextCtrl( parent, -1, self.str_value, style = style )

wx.EVT_KILL_FOCUS( control, self.update_object )

if factory.auto_set:
    wx.EVT_TEXT( parent, control.GetId(), self.update_object )

self.control = control
self.set_tooltip()

```

真正产生控件的程序在 `init()` 方法中，此方法在产生界面时自动被调用，注意不要和对象的初始化方法 `__init__()` 搞混淆，`init()` 在生成界面时被调用。下面查看 `SimpleEditor` 的父类：

```

>>> ed.simple_editor_class.mro()
[<class 'enthought.traits.ui.wx.text_editor.SimpleEditor'>,
 <class 'enthought.traits.ui.wx.editor.Editor'>,
 <class 'enthought.traits.ui.editor.Editor'>,
 <class 'enthought.traits.has_traits.HasPrivateTraits'>,
 <class 'enthought.traits.has_traits.HasTraits'>,
 <type 'CHasTraits'>,
 <type 'object'>]

```

`SimpleEditor` 从后台界面库的 `Editor` 类继承，`Editor` 类中定义了界面库编辑器中一些通用的属性和方法。而界面库中的 `Editor` 类又从 `TraitsUI` 库的 `Editor` 类继承。它定义了所有 `Trait` 编辑器的基本属性和功能，源代码可以在“%ui%\editor.py”中找到。

7.6.2 制作 matplotlib 的编辑器

Enthought 的官方绘图库采用的是 Chaco，不过如果读者对 matplotlib 更为熟悉，就可以在界面中使用 matplotlib 的绘图控件进行绘图。为了实现这个目的，我们需要自己编写一个 `Trait` 编辑器，用它封装 matplotlib 的绘图控件。下面是完整的源代码，程序的运行结果如图 7-17 所示。



`mpl_figure_editor.py`

嵌入 TraitsUI 界面中的 matplotlib 控件

```

import wx
import matplotlib
# matplotlib 采用 WXAgg 为后台，这样才能将绘图控件嵌入以 wx 为后台界面库的 traitsUI 窗口中
matplotlib.use("WXAgg")
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.backends.backend_wx import NavigationToolbar2Wx
from enthought.traits.ui.wx.editor import Editor
from enthought.traits.ui.basic_editor_factory import BasicEditorFactory

class _MPLFigureEditor(Editor): ❶
    """
    相当于 wx 后台界面库中的编辑器，它负责创建真正的控件
    """
    scrollable = True

    def init(self, parent):
        self.control = self._create_canvas(parent)
        self.set_tooltip()

    def update_editor(self): ❷
        pass

    def _create_canvas(self, parent): ❸
        """
        创建一个 Panel，布局则采用垂直排列的 BoxSizer，在 panel 中添加
        FigureCanvas、NavigationToolbar2Wx 和 StaticText 三个控件
        FigureCanvas 的鼠标移动事件调用 mousemoved 函数，在 StaticText 中
        显示鼠标所在的数据坐标
        """
        panel = wx.Panel(parent, -1, style=wx.CLIP_CHILDREN)
        def mousemoved(event):
            panel.info.SetLabel("%s, %s" % (event.xdata, event.ydata))
        panel.mousemoved = mousemoved
        sizer = wx.BoxSizer(wx.VERTICAL)
        panel.SetSizer(sizer)
        mpl_control = FigureCanvas(panel, -1, self.value) ❹
        mpl_control.mpl_connect("motion_notify_event", mousemoved)
        toolbar = NavigationToolbar2Wx(mpl_control)
        sizer.Add(mpl_control, 1, wx.LEFT | wx.TOP | wx.GROW)
        sizer.Add(toolbar, 0, wx.EXPAND|wx.RIGHT)
        panel.info = wx.StaticText(panel, -1)
        sizer.Add(panel.info)

        self.value.canvas.SetMinSize((10,10))
        return panel

```

```

class MPLFigureEditor(BasicEditorFactory): ❸
    """
    相当于 traits.ui 中的 EditorFactory，它返回真正创建控件的类
    """

    klass = _MPLFigureEditor

if __name__ == "__main__":
    from matplotlib.figure import Figure
    from enthought.traits.api import HasTraits, Instance
    from enthought.traits.ui.api import View, Item
    from numpy import sin, linspace, pi

    class Test(HasTraits):
        figure = Instance(Figure, ()) ❹
        view = View(
            Item("figure", editor=MPLFigureEditor(), show_label=False),
            width = 400,
            height = 300,
            resizable = True)
        def __init__(self):
            super(Test, self).__init__()
            axes = self.figure.add_subplot(111)
            t = linspace(0, 2*pi, 200)
            axes.plot(sin(t))

    Test().configure_traits()

```

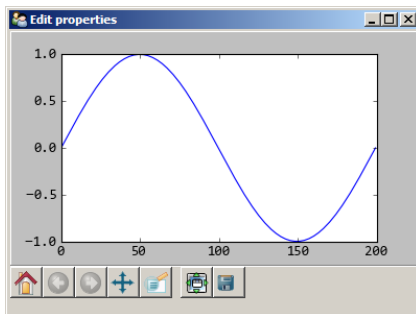


图 7-17 在 TraitsUI 界面中嵌入的 matplotlib 绘图控件

由于这个 matplotlib 绘图编辑器没有'simple'等 4 种样式，也不会放到 wx 后台界面库的模块中，因此不需要采用前面介绍的自动查找编辑器类的办法。对于这种情况，TraitsUI 提供了 BasicEditorFactory 类以方便我们实现编辑器和后台界面库的连接。它的源程序可以在“%ui%\basic_editor_factory.py”中找到。下面是其中的一部分：

```

class BasicEditorFactory ( EditorFactory ):
    klass = Any

```

```
def _get_simple_editor_class ( self ):
    return self.klass
...
```

BasicEditorFactory 类通过覆盖父类的 `_get_simple_editor_class()` 方法，直接返回创建控件用的类 `klass`。⑤ `MPLFigureEditors` 从 `BasicEditorFactory` 继承，并指定创建控件的类为 `_MPLFigureEditor`。

① `_MPLFigureEditor` 和 `text_editor.SimpleEditor` 一样，也从 `Editor` 类继承，在它的 `init()` 方法中，调用 ③ `_create_canvas()` 创建实际的控件。

② `Editor` 类中有一个 `update_editor()` 方法，它在对应的 `Trait` 属性改变时会被调用，因为绘图控件不需要这个功能，所以覆盖 `update_editor()`，让它不做什么事情。

④ 在 `matplotlib` 中，创建 `FigureCanvas` 对象时需要指定与其对应的 `Figure` 对象，这里 `self.value` 就是 `Figure` 对象，⑥ 它在模型类 `Test` 中用一个名为 `figure` 的 `Trait` 属性表示。控件可以通过其 `value` 属性获得模型类中它所编辑的对象。因此，`_MPLFigureEditor` 中的 `value` 属性和 `Test` 类中的 `figure` 属性是同一个 `Figure` 对象。

最后，`_create_canvas()` 中的程序编写和在标准的 `wx` 窗口中添加控件是一样的，与界面库相关的细节不是本书的重点，因此这里不再详细解释，读者可以参考 `matplotlib` 和 `wxPython` 的相应文档。

7.6.3 CSV 数据绘图工具

用前面介绍的 `matplotlib` 绘图控件可以快速制作一个 CSV 数据绘图工具。用此工具打开一个 CSV 数据文档之后，可以用其中的任意两列数据为 X、Y 轴数据绘制图表。用户还可以自由地添加新的图表，修改图表的标题，选择图表的 X 轴和 Y 轴的数据。程序运行时需要从“`mpl_figure_editor.py`”模块载入刚才介绍的 `matplotlib` 绘图控件模块。程序的界面如图 7-18 所示(见文前彩插)。



traitsUI_csv_viewer.py

使用 matplotlib 绘图控件制作 CSV 数据绘图工具

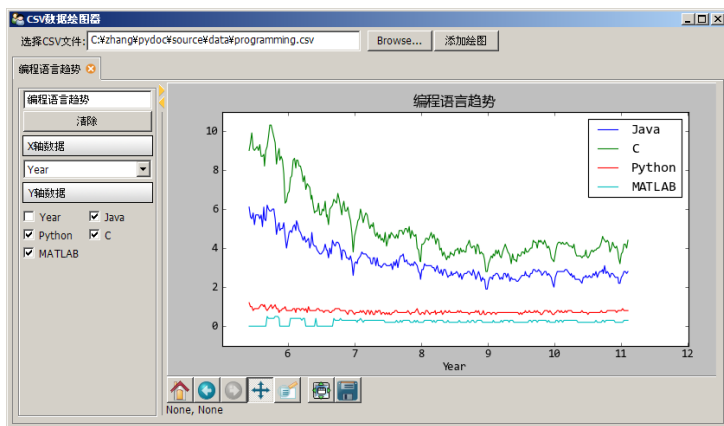


图 7-18 CSV 数据绘图工具的界面

程序以标签页的形式显示多个绘图，用户可以从左侧的数据选择栏中选择 X 轴和 Y 轴的数据。标签页可以自由地拖动，构成上下左右分栏，并且可以隐藏左侧的数据选择栏，如图 7-19 所示(见文前彩插)。

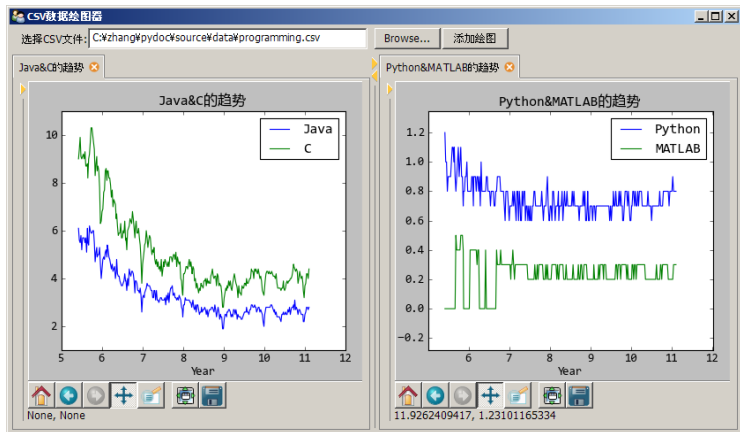


图 7-19 使用可调整 DOCK 的多标签页界面，方便用户对比数据

由于绘图控件是 matplotlib 提供的，因此平移、缩放、保存文件等功能也一应俱全。由于所有的界面都是采用 TraitsUI 设计的，因此主窗口既可以用来单独显示，也可以嵌入到一个更大的界面中，运用十分灵活。

程序中已经有比较详细的注释，这里不再重复。如果读者对 Traits 库的某项用法还不太了解，可以直接查看其源代码，代码中都有详细的注释。整个程序的界面处理都只是组装 View 对象，看不到任何关于控件操作的代码，因此大大节省了程序的开发时间。



标签页的标题如果输入中文可能会出现错误，请按照 7.4.4 节的说明修改 TraitsUI 库的源代码。