第3章 Wolfram语言列表

在 Wolfram 语言中，各种类型的数据均由原子类型数据构成，而用花括号“{ }”包围起来的各种类型数据的结构称为列表（List）。例如，“{a，b，1，2，3，{5.1}}”为长度为 5 的列表，由符号 a、b 和整数 1、2、3 以及一个子列表{5.1}构成。Wolfram 语言为函数式的语言，花括号“{ }”对应的函数为 List，上述的列表“{a，b，1，2，3，{5.1}}”在 Wolfram 语言内部的形式为“List[a，b，1，2，3，List[5.1]]”（借助于函数 FullForm[{a，b，1，2，3，{5.1}}]查看）。使用花括号或 List 函数是创建列表的基本方法，此外，Wolfram 语言提供了创建列表的大量内置函数。本章将详细介绍列表的创建函数和基于列表的常用操作函数等。列表是 Wolfram 语言中最常用的基础数据结构，熟悉列表的创建和操作方法是全面掌握 Wolfram 语言的关键。

3.1 列表构造

Wolfram 语言中，数据的存储和处理主要基于列表实现。最直接的创建列表方法为使用 List 函数或花括号，并手动输入列表中的数据，如图 3-1 所示。

在图 3-1 中，“In[2]”使用 List 函数创建了一个列表 t1，“In[3]”使用花括号方式创建了另一个列表 t2，“In[4]”创建了一个空列表 t3。在“In[5]”中使用 Length 函数统计了列表 t1、t2 和 t3 的长度，如“Out[5]”所示，依次为 4、6 和 0。“In[6]”调用 FullForm 函数显示了列表 t2 的完全格式，即 Wolfram 语言表示列表 t2 的格式，可见列表是用 List 函数施加于一组数据而构成的结构。

借助于花括号“{ }”和 List 函数手动创建列表，考虑时间因素，仅限于创建列表元素个数比较少的小型列表。对于大型列表而言，一般借助于各种数据传感器采集感兴趣的数据信息，这些数据以数据文件的形式保存在硬盘或云盘上，然后，借助于 Import 函数将这些数据导入 Mathematica 软件中，以列表的形式存储；对于一些有规律的列表，可以借助于 Wolfram 语言的内置列表构造函数构造。下面将介绍 Import 函数和常用的列表构造函数。

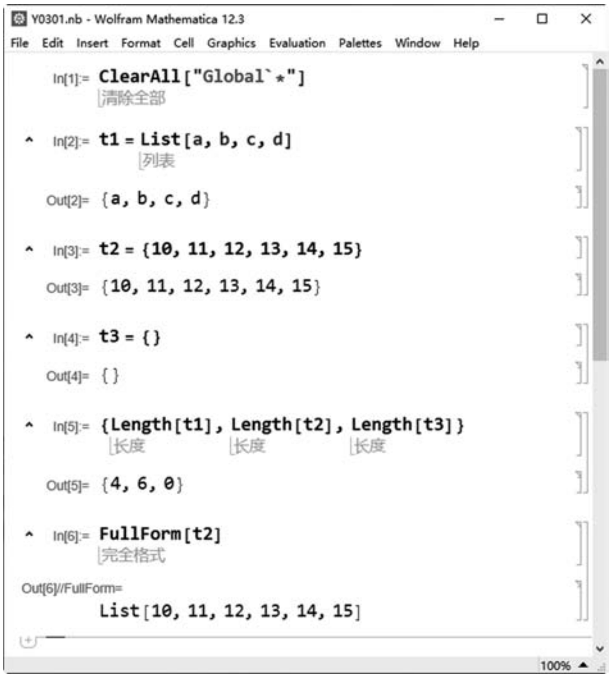


图 3-1　直接创建列表的方法

3.1.1　Import 函数

Import 函数可以向笔记本中导入各种类型的文件数据，也可以像“爬虫”一样从网站上导入网页数据等。Import 函数功能极其强大，而且随着 Mathematica 软件版本的升级，Import 函数正在支持越来越多的数据文件类型。目前已经支持 234 种数据文件类型格式，可以通过全局参数“$ImportFormats”查看，其中，几乎包括全部图像数据文件格式。

下面以 XML 格式文档和 Excel 格式文档为例介绍 Import 函数导入文件数据的用法。这里，打开 Word 软件编辑一个文件，如图 3-2 所示，将该文件存储为 XML 格式文件，文件名取为 D0301.xml，保存在目录“E:\ZYMaths\YongZhang”下。

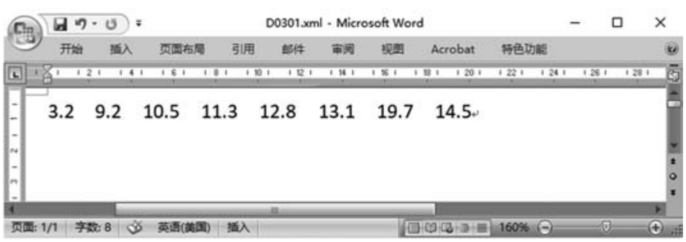
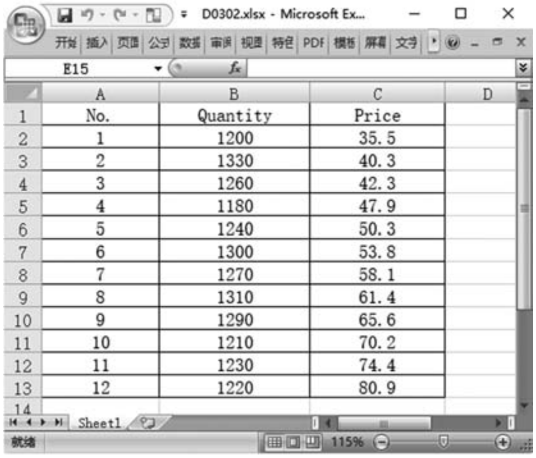


图 3-2　XML 文档 D0301.xml

由图 3-2 可知，文档 D0301.xml 的内容为一个浮点数序列。

现在，打开 Excel 软件编辑一个电子表格，如图 3-3 所示，将文件命名为 D0302.xlsx，保存在目录“E:\ZYMaths\YongZhang”下。



	A	B	C	D
1	No.	Quantity	Price	
2	1	1200	35.5	
3	2	1330	40.3	
4	3	1260	42.3	
5	4	1180	47.9	
6	5	1240	50.3	
7	6	1300	53.8	
8	7	1270	58.1	
9	8	1310	61.4	
10	9	1290	65.6	
11	10	1210	70.2	
12	11	1230	74.4	
13	12	1220	80.9	

图 3-3　Excel 格式文件 D0302. xlsx

在图 3-3 中，Excel 文件 D0302. xlsx 具有一个表单 Sheet1，为一个 13 行 3 列的电子表格。

Import 函数可以读取上述两个文件中的内容，如图 3-4 所示。

在图 3-4 中，“In[1]”调用函数“ClearAll["Global' * "]”清除笔记本中已定义的变量的值，然后，调用“SetDirectory[NotebookDirectory[]]”将工作目录设置为“E:\ZYMaths\YongZhang”，如“Out[1]”所示。

在“In[2]”中，使用 Import 函数从“D0301. xml”文件中读取数据，使用了选项“CDATA”，表示读取数据项，此时，将 D0301. xml 中的数据以字符串列表的形式读出，而这个字符串列表的第一个元素为图 3-2 中显示的数据。所以，使用 First 函数获取读到的字符串列表的第一个元素，并将其赋给 t1，t1 的值如“Out[2]”所示。这里，全局变量 t1 的值为一个字符串，在“In[3]”中使用函数“InputForm[t1]”可以显示 t1 的输入形式，如“Out[3]”所示，即"3.2 9.2 10.5 11.3 12.8 13.1 19.7 14.5"，这一个由双引号包围的字符串。现在，将字符串形式的 t1 转化为数值列表。在“In[4]”中调用函数“StringSplit[t1]”，从 t1 中的空格处分隔字符串 t1，得到一个字符串列表，保存在 t2 中，如“Out[4]”所示。然后，在“In[5]”中调用函数“ToExpression[t2]”将字符串列表 t2 转化为数值列表，保存在 t3 中，如“Out[5]”所示。

上述操作表明，函数 Import 以字符串列表的形式读取 XML 文件中的内容，读出的数据常常需要进行后续的处理，这与 XML 文件存储格式有关。Excel 格式文件中的数据有明确的类型标识，Import 函数读取 Excel 格式文件的数据可以识别数据的类型。在“In[6]”中调用“Import["D0302. xlsx"]”读取图 3-3 所示 Excel 文件 D0302. xlsx 中的内容，读出的结果保存在 t4 中，如“Out[6]”所示，这是一个三层的列表，这个列表的每个子列表对应着一个 Excel 文件中的表单。由图 3-3 可知，这里的 D0302. xlsx 只有一个表单“Sheet1”，所以，t4 仅有一个子列表。这个子列表为一个二维的列表，它的每个子列表对应着图 3-3 中的一行数据。

在图 3-4 的“In[7]”中使用 TableForm 函数将 t4 中的数据以表格的形式呈现，这里的 TableForm 函数的第一个参数为“Rest[First[t4]]”，其中，“First[t4]”返回列表 t4 的第一个元素，这里是返回 t4 的第一个子列表，即“{{No. ,Quantity,Price},{1. ,1200. ,35.5},{2. ,1330. ,40.3},{3. ,1260. ,42.3},{4. ,1180. ,47.9},{5. ,1240. ,50.3},{6. ,1300. ,

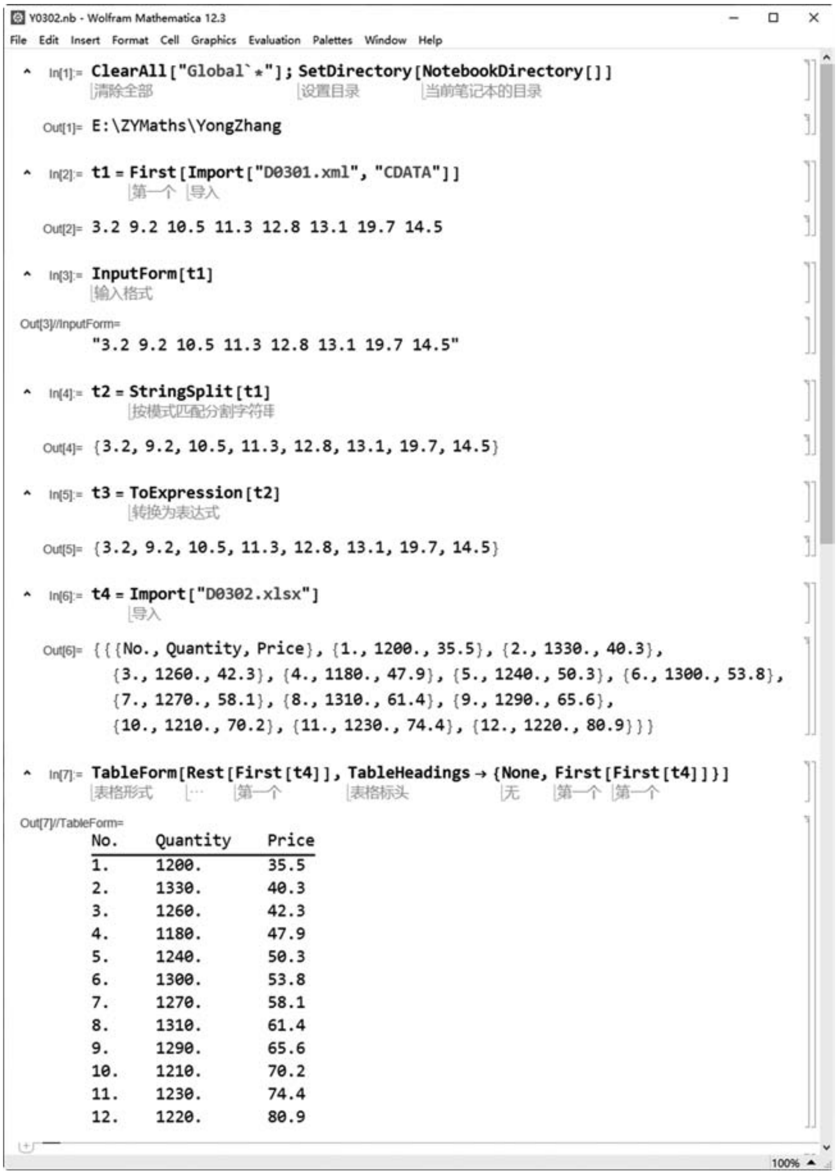


图 3-4　Import 函数读取 XML 格式文件和 Excel 文件

53.8},{7.,1270.,58.1},{8.,1310.,61.4},{9.,1290.,65.6},{10.,1210.,70.2},{11.,1230.,74.4},{12.,1220.,80.9}}”,然后,“Rest[First[t4]]”返回“First[t4]”的除第一个元素外的元素构成的列表,即“{1.,1200.,35.5},{2.,1330.,40.3},{3.,1260.,42.3},{4.,1180.,47.9},{5.,1240.,50.3},{6.,1300.,53.8},{7.,1270.,58.1},{8.,1310.,61.4},{9.,1290.,65.6},{10.,1210.,70.2},{11.,1230.,74.4},{12.,1220.,80.9}}”。其中函数“Rest[列表]”得到丢掉“列表”的第一个元素的新列表。TableForm 函数的第二个参数为选项“TableHeadings−>{None,First[First[t4]]}”,用于设定表格的行标题和列标题,其中 None 表示无行标题,而列标题为“First[First[t4]]”,表示 t4 的第一个元素(这里为二层子列表)的第一个元素(这个元素仍为列表),即“{No.,Quantity,Price}”。TableForm 函数的

显示结果如图 3-4 的“Out[7]”所示。

除了读取数据文件外，Import 函数还常用于读取图像文件。类似于上述将创建的 XML 文档和 Excel 文件先保存到工作目录再调用 Import 函数读取其文件内容相似，需要先将拟读取的图像文件保存到工作目录中，然后使用 Import[“图像文件全名”]读取图像文件内容，并以图像的形式显示在笔记本中。在 Import 函数中必须输入“图像文件全名”，即包含图像文件的文件名和扩展名。如果图像文件不在工作目录中，Import 函数需要使用完整的路径名加上图像文件全名读取图像文件内容。

在笔记本中，输入“$Path”可以查看 Mathematica 软件的默认搜索路径，其中，包括目录“C:\Program Files\Wolfram Research\Mathematica\12.3\Documentation\English\System\”。在这个目录下有一个子目录“ExampleData”，在这个子目录中有一些实例数据文件，其中包括“rose.gif”。可借助 Import 函数读取这个图像文件，其语句为“Import["C:\\Program Files\\Wolfram Research\\Mathematica\\12.3\\Documentation\\English\\System\\ExampleData\\rose.gif"]”。在使用“Import”函数读取文件时，默认的搜索路径可以省略，上述语句中只需要输入“Import["ExampleData\\rose.gif"]”或“Import["ExampleData/rose.gif"]”，这里子目录的分隔符使用双斜线“\\”或反斜线“/”。注意：子目录“ExampleData/”不能省略，因为它不在默认的搜索路径中。图 3-5 为 Import 函数读取图像文件的实例。

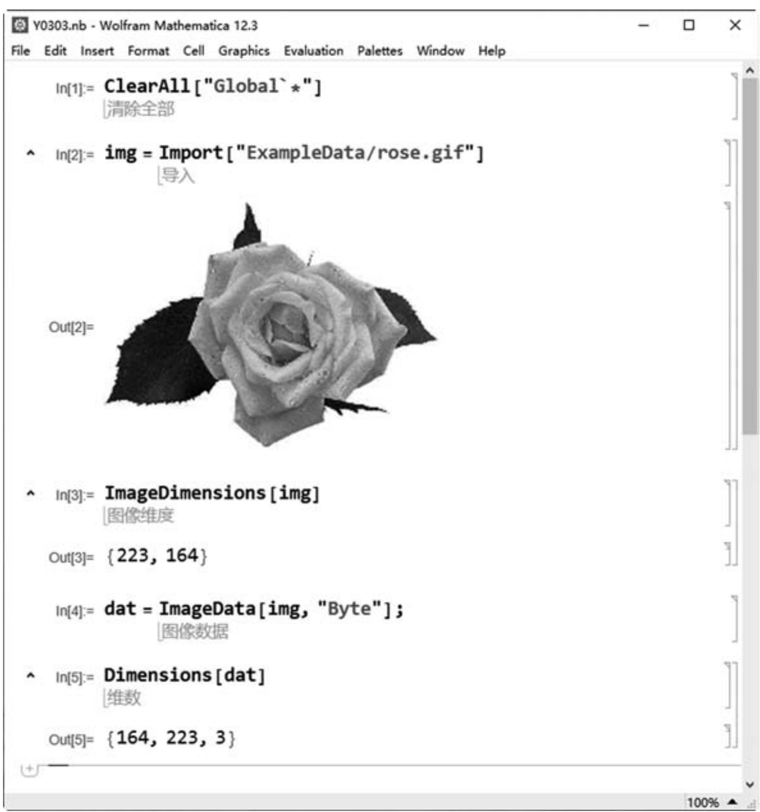


图 3-5 Import 函数读取图像文件的实例

在图3-5中,“In[2]”执行语句“img=Import["ExampleData/rose.gif"]”读取图像文件rose.gif,赋给全局变量img,并在“Out[2]”中显示了该图像(“玫瑰花”)。在“In[3]”中使用语句“ImageDimensions[img]”获取图像img的大小,如“Out[3]”所示为“{223,164}”,表示图像的宽为223,高为164,即这是一幅164×223像素点的彩色图像。在“In[4]”中,执行语句“dat=ImageData[img,"Byte"];”,使用函数“ImageData”获取图像中的数据(即每个像素点的值),参数“Byte”表示像素点的数据以字节形式读取,读取的数据保存在全局变量dat中。由于图像img为一幅彩色图像,每个像素点由R(红色)、G(绿色)和B(蓝色)三色表示,因此,dat的大小为164×223×3。在“In[5]”中使用语句“Dimensions[dat]”得到dat的维数,如“Out[5]”所示,为“{164,223,3}”。

函数Import还可以像“网络爬虫”一样从互联网上爬取数据,例如,语句“Import["http://www.wolfram.com"]”将读取网页“http://www.wolfram.com”上的文本数据;而语句“Import["http://www.wolfram.com","Images"]”将读取该网页上的图像;语句“Import["http://www.wolfram.com","Hyperlinks"]”将读取该网页上的超链接信息。此外,与函数Import功能相对的函数为Export函数,用于输出笔记本中的列表数据。

3.1.2 Table函数

列表数据有规律的列表可用Table函数生成。Table函数是Wolfram语言最常用的函数,它的用法和Do函数相似,可以实现循环功能,但需注意:①在Table函数内部不能使用Continue函数和Break函数,Do函数中可以使用这两个函数;②Do函数没有输出,Table函数输出一个列表;③Table函数的执行速度远远高于Do函数、For函数和While函数等循环控制函数,所以,常使用Table函数替换这些循环控制函数。

Table函数的语法如下:

(1) Table[表达式, n],表示循环执行“表达式”n次,生成一个长度为n的列表,每个元素为“表达式”在循环时计算得到的值。

(2) Table[表达式, n_1, n_2,…,n_k],这是上述第(1)个语法的扩展版本,表示循环执行“表达式”$n_1\times n_2\times\cdots\times n_k$次,生成一个大小为$n_1\times n_2\times\cdots\times n_k$的列表,每个元素为“表达式”在循环时计算得到的值。

(3) Table[表达式, {i, i_{max}}],表示循环变量i从1按步长1递增到i_{max},循环执行“表达式”,生成一个列表,列表的每个元素为“表达式”在循环执行时计算得到的值。这里的循环变量i为属于Table表的局部变量,所以,在Table函数的嵌套中(Table表达式中还有Table函数),两个Table函数可以使用相同的局部变量。但是,为了增强程序的可读性,建议嵌套的Table函数使用不同的循环变量。

(4) Table[表达式, {i, i_{min}, i_{max}}],表示循环变量i从i_{min}按步长1递增到i_{max},循环执行“表达式”,生成一个列表,列表的每个元素为“表达式”在循环执行时计算得到的值。这里的循环变量i为属于Table表的局部变量。

(5) Table[表达式, {i, i_{min}, i_{max}, di}],表示循环变量i从i_{min}按步长di递增到i_{max},循环执行“表达式”,生成一个列表,列表的每个元素为“表达式”在循环执行时计算得到的值。这里的循环变量i为属于Table表的局部变量。

(6) Table[表达式，{i，{i_1，i_2，…}}]，表示循环变量 i 从列表{i_1，i_2，…}中依次取值，循环执行“表达式”，生成一个列表，列表的每个元素为“表达式”在循环执行时计算得到的值。这里的循环变量 i 为属于 Table 表的局部变量。

上述的第(3)～(6)种语法均生成一维列表，这些语法均可以扩展为生成多层列表的版本，如下所示：

(7) Table[表达式，{i_1，$i_{1,max}$}，{i_2，$i_{2,max}$}，…，{i_n，$i_{n,max}$}]，上述第(3)种语法的扩展版本。

(8) Table[表达式，{i_1，$i_{1,min}$，$i_{1,max}$}，{i_2，$i_{2,min}$，$i_{2,max}$}，…，{i_n，$i_{n,min}$，$i_{n,max}$}]，上述第(4)种语法的扩展版本。

(9) Table[表达式，{i_1，$i_{1,min}$，$i_{1,max}$，di_1}，{i_2，$i_{2,min}$，$i_{2,max}$，di_2}，…，{i_n，$i_{n,min}$，$i_{n,max}$，di_n}]，上述第(5)种语法的扩展版本。

(10) Table[表达式，{i_1，{$i_{1,1}$，$i_{1,2}$，…}}，{i_2，{$i_{2,1}$，$i_{2,2}$，…}}，…，{i_3，{$i_{3,1}$，$i_{3,2}$，…}}]，上述第(6)种语法的扩展版本。

在上述 Table 函数的 10 种语法中，其中“表达式”的值是“串行”计算的。这里用第(3)种语法“Table[表达式，{i，i_{max}}]”为例说明一下：“Table[表达式，{i，i_{max}}]”的执行将得到一个长度为 i_{max} 的列表，该列表具有 i_{max} 个元素，每个元素依次计算，即当 i 取值 1 时，计算一次“表达式”的值(表达式中可能会用到 i 的值)，得到列表的第 1 个元素；然后 i 递增 1，此时再计算一次“表达式”的值(表达式中可能会用到 i 的值)，得到列表的第 2 个元素；以此类推，直到 i 取 i_{max}，再最后计算一次“表达式”的值(表达式中可能会用到 i 的值)，得到列表的最后一个元素。

Table 的这种计算方式与 Do 函数计算方式相同，着重强调这种计算方式的目的在于两种原因：

其一，Wolfram 语言中还有一个并行计算函数 ParallelTable，它的语法和用法与 Table 的语法和用法相同，但可以并行执行。例如下面的两条语句：

(1) Table[Pause[1];RandomInteger[10],10]//AbsoluteTiming。

(2) ParallelTable[Pause[1];RandomInteger[10],10]//AbsoluteTiming。

上述这两条语句中的表达式均为“Pause[1]; RandomInteger[10]”，这里的“Pause[1]”表示暂停 1 秒，“RandomInteger[10]”表示生成一个 0～10 的随机整数。对于使用 Table 函数的第(1)条语句，由于“表达式”被串行 10 次，所花费的时间至少为 10 秒，因为其中的延时一秒函数“Pause[1]”被串行执行了 10 次。而第(2)条语句，由于使用了并行计算，所有的表达式均并行执行，只花费了一秒多的运行时间。注意：并行计算在计算前需要做大量的数据预处理工作，当执行简单的计算时，ParallelTable 函数可能不如 Table 函数快。

其二，还有一个 ConstantArray 函数可以像上述 Table 函数的第(1)种语法一样，生成常数列表。但是，在 ConstantArray 函数中表达式只计算一次。

下面通过典型实例依次介绍 Table 函数各种语法的应用方法，先讨论 Table 函数的第(1)和第(2)种语法的实例，如图 3-6 所示。

图 3-6 中，“In[2]”中使用语句“t1＝Table[0，5]”生成长度为 5 的元素全为 0 的一维列表，如“Out[2]”所示。在“In[3]”所示语句“(t2＝Table[0,2,3])//MatrixForm”中，先执行括号内部的“t2＝Table[0,2,3]”，生成一个两层嵌套列表赋给全局变量 t2，它包括 2 个子列

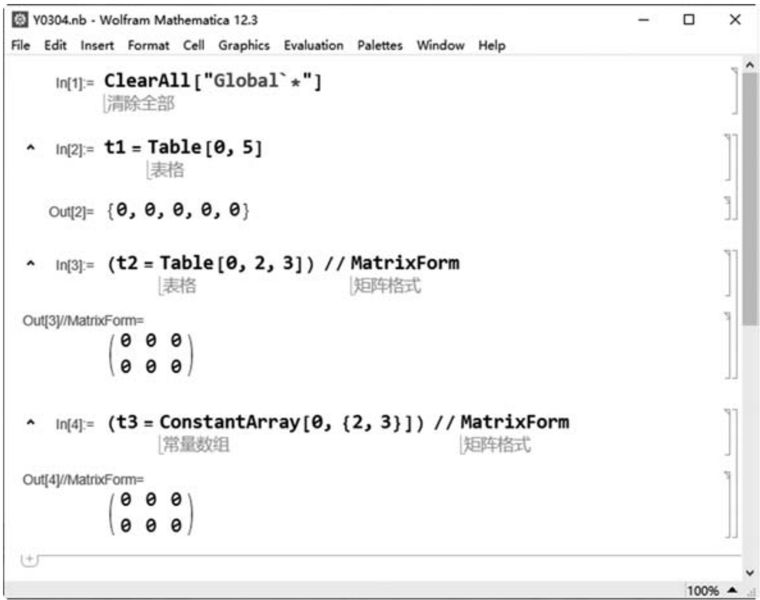


图 3-6　Table 函数实例

表，每个子列表包括 3 个元素。然后，将 MatrixForm 函数作用于 t2 上，以矩阵的形式显示列表 t2。注意，如果去掉“In[3]”中的括号，表达式变为“t2 = Table[0, 2, 3]// MatrixForm”，相当于“t2=(Table[0,2,3]//MatrixForm)”（赋值号优先级最低），此时，t2 将保存列表的矩阵显示格式，如“Out[3]”所示，而不是保存着列表本身，即此时的 t2 不能参与列表的运算。

除了使用 Table 函数可以生成常数形式的列表外，还可以使用 ConstantArray 函数生成这类列表，例如，在图 3-6 的“In[4]”中，使用 ConstantArray 函数生成了列表 t3，t3 和 t2 完全相同。

ConstantArray 函数的语法如下：

（1）ConstantArray[表达式，n]，生成一个长度为 n 的列表，每个元素为“表达式”的值。注意：这里的“表达式”只会被计算一次。

（2）ConstantArray[表达式，{n_1，n_2，…，n_k}]，生成一个大小为 $n_1 \times n_2 \times \cdots \times n_k$ 的列表，每个元素为“表达式”的值。注意：这里的“表达式”仅被计算一次。

例如，如图 3-7 所示的两条语句。

在图 3-7 中，函数 SeedRandom 用于为伪随机数发生器设为“种子”，即初始值。凡是具有相同“种子”的伪随机数发生器生成的随机数都是相同的。在“In[5]”中，调用语句“SeedRandom[899789]”后，再调用语句“Table[RandomInteger[100],5]”，生成一个长度为 5 的列表，每个列表元素为一个由 RandomInteger 函数生成的 0～100 的随机整数，如“Out[5]”所示，列表中的随机数不同，表明 RandomInteger 被重复执行了 5 次。在“In[6]”中再次调用语句“SeedRandom[899789]”，以确保后续重复调用“RandomInteger[10]”时将依次生成序列“62，84，31，45，83，…”，即生成与“Out[5]”相同的序列。但是，在“In[6]”中执行语句“ConstantArray[RandomInteger[100]，5]”时，RandomInteger[100]仅被计算

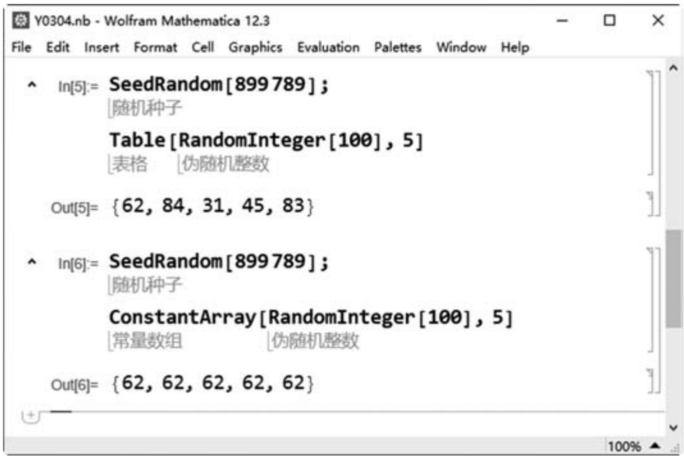


图 3-7　Table 和 ConstantArray 创建常数列表的区别

一次，即得到第一个元素 62，所以得到的列表为一个长度为 5 的元素全为 62 的列表，如“Out[6]”所示。

Table 函数的第(3)～(5)种语法的用法如图 3-8 所示。

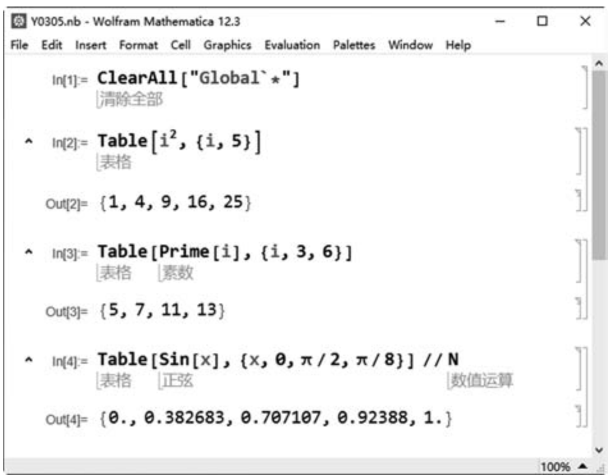


图 3-8　Table 函数生成列表实例

在图 3-8 中，“In[2]”调用语句“Table[i^2,{i,5}]”生成一个列表，循环变量 i 从 1 按步长 1 递增至 5，列表的每个元素为循环变量 i 的平方，即“{1,4,9,16,25}”，如“Out[2]”所示。在“In[3]”中使用了 Prime 函数，Prime[n]用于生成第 n 个素数，这里的语句“Table[Prime[i],{i,3,6}]”将产生由第 3～第 6 个素数为元素的列表，如“Out[3]”所示。在“In[4]”中，循环变量为 x，从 0 按步长 π/8 递增到 π/2，循环执行 Sin[x]。这里的“表达式//N”是一种函数调用的后缀写法，相当于“N[表达式]”。使用函数 N 将 Table 函数生成的列表转化为浮点数显示，如“Out[4]”所示。

由图 3-8 的实例可知，在 Table 函数中，循环变量的名称可以使用任意合法的标识符，在 Table 函数的第(5)种语法的循环变量部分“{i, i_{min}, i_{max}, di}”中，变量变化的起始值 i_{min}

和最大值 i_{max}(可能取不到,与步长有关)以及步长 di,可以取为任意的数值类型。

图 3-9 为 Table 函数的第(6)~第(10)种语法的典型实例。

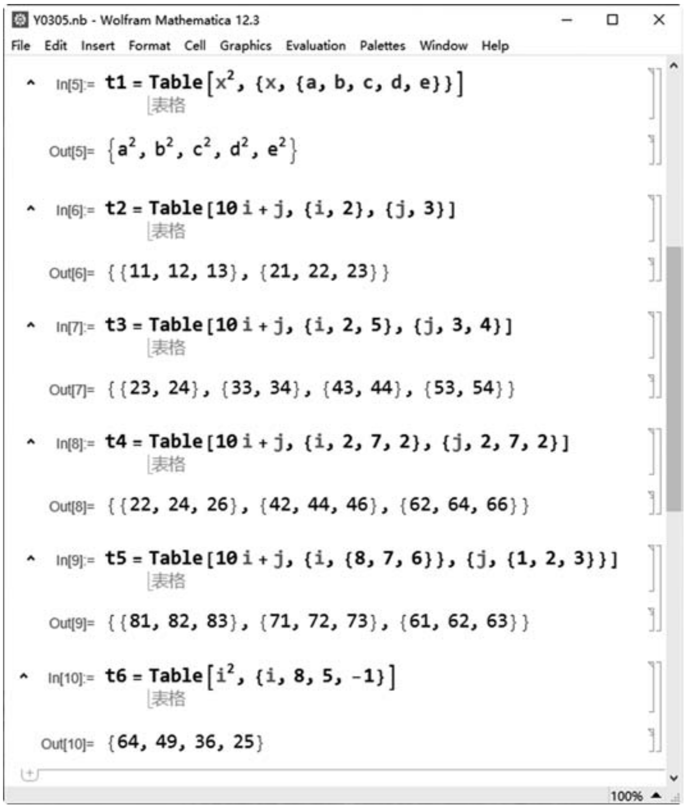


图 3-9　Table 函数生成列表实例

在图 3-6 中,“In[5]”中的 Table 函数的循环变量为 x,取值为列表“{a, b, c, d, e}”,其生成的列表如“Out[5]”所示,其各个元素顺次对应着 x 依次遍历列表“{a, b, c, d, e}”中的符号,这种用法表明当循环变量取值为列表时,可以使用符号,并生成符号表示的列表。在“In[6]”~“In[9]”中,各个 Table 函数的表达式均为“10i+j”,循环变量 i 和 j 均为个位数,这样,表达式“10i+j”相当于把 i 作为十位数、j 作为个位数生成一个整数,以显示 i 和 j 的“增长”规律。在“In[6]”的 Table 函数中,循环变量 i 从 1 按步长 1 递增到 2,对于每个 i,j 都从 1 按步长 1 递增到 3,因此得到的列表为“{{11,12,13},{21,22,23}}”,如“Out[6]”所示。在“In[7]”的 Table 函数中,循环变量 i 从 2 按步长 1 递增到 5,对于每个 i,j 都从 3 按步长 1 递增到 4,对于每个 i 和 j 的组合,都计算一次表达式“10i+j”,得到的列表为“{{23,24},{33,34},{43,44},{53,54}}”,如“Out[7]”所示,其中包含 4 个子列表(与 i 的取值个数相等),每个子列表具有 2 个元素(与 j 的取值个数相等)。在“In[8]”的 Table 函数中,循环变量 i 从 2 按步长 2 递增到 6(最大值 7 取不到),对于每个 i,j 都从 2 按步长 2 递增到 6(最大值 7 取不到),对于每个 i 和 j 的组合,都计算一次表达式“10i+j”,得到的列表为“{{22,24,26},{42,44,46},{62,64,66}}”,如“Out[8]”所示,其中包含 3 个子列表(与 i 的取值个数相等),每个子列表具有 3 个元素(与 j 的取值个数相等)。在“In[9]”的 Table 函数中,循

环变量 i 顺序从列表“{8, 7, 6}”中取值，对于每个 i,j 都顺序遍历列表“{1, 2, 3,}”中的全部元素，对于每个 i 和 j 的组合，都计算一次表达式“10i+j”，因此得到的列表为“{{81,82,83},{71,72,73},{61,62,63}}”，如“Out[9]”所示。

当 Table 函数中的循环变量按“递减”的方式变化时，必须指定步长，如图 3-9 的“In[10]”所示。在“In[10]”的 Table 函数中，循环变量 i 从 8 按步长 −1 递减到 5，对于每个 i，计算表达式“i^2”的值，得到如“Out[10]”所示的列表。注意：这里 Table 函数中的循环变量递减的步长“−1”不能省略。

上述方法中生成的两层嵌套列表具有一个共同点，即它们的子列表具有相同的元素，用矩阵形式表示时，将呈现为矩形的阵列。图 3-10 中的 Table 函数将生成一个不规则的列表，即九九乘法表。

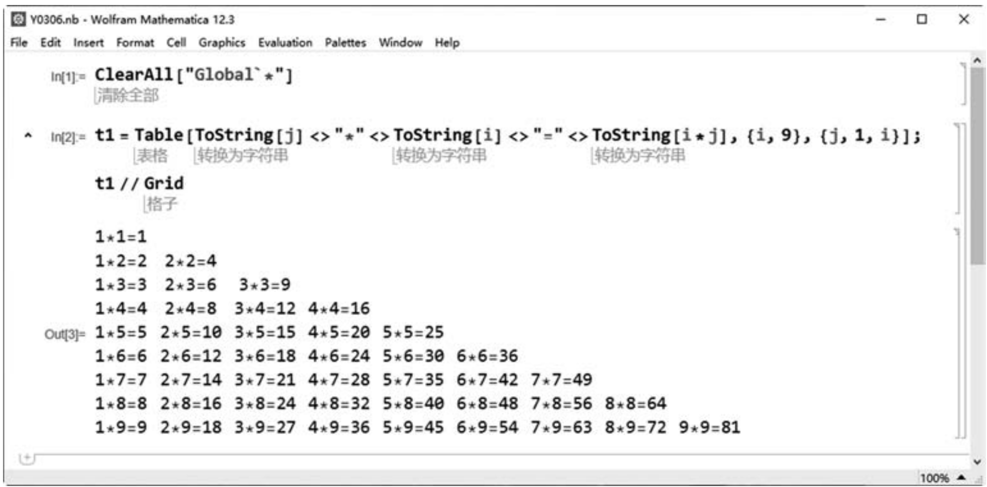


图 3-10 九九乘法表

在图 3-10 的“Table”函数中，循环变量 i 从 1 按步长 1 递增到 9，对于每个 i，循环变量 j 都从 1 按步长 1 递增到 i，然后，对于每组 i 和 j，计算表达式“ToString[j]<>" * "<>ToString[i]<>" = "<> ToString[i * j]”，该表达式中“ToString[表达式]”将“表达式”转化为字符串，“<>”表示连接两个字符串成为一个更长的字符串。Table 函数的计算结果保存在全局变量 t1 中。语句“t1//Grid”与“Grid[t1]”等价，Grid 函数将 t1 以二维的网格状呈现，如“Out[3]”所示。

最后，关于 Table 函数需要说明的是，在其语法中的“表达式”均可以为语句组，语句组中的各条语句以分号分隔，语句组最后一条语句的计算结果作为本次循环的输出，即作为 Table 函数输出的列表中的一个元素。图 3-11 给出了一个典型的实例，求 10 层的杨辉三角形。

在图 3-11 中，“In[4]”定义了函数 t2(无参数)，使用模块 Module 实现。在模块 Module 内部，定义了局部变量 y1、y2、y3 和 yang，其中 y1 初始化为{0, 1, 0}。然后，Table 函数的执行结果赋给 yang，Table 函数中包含了语句组，即“Table[y2=Partition[y1,2,1]; y3=Map[Total,y2]; y1=Join[{0},y3,{0}]; y3,{i,2,10}]”，这里，循环变量 i 从 2 按步长 1 递增到 10，共循环执行语句组“y2=Partition[y1,2,1]; y3=Map[Total,y2]; y1=Join[{0},

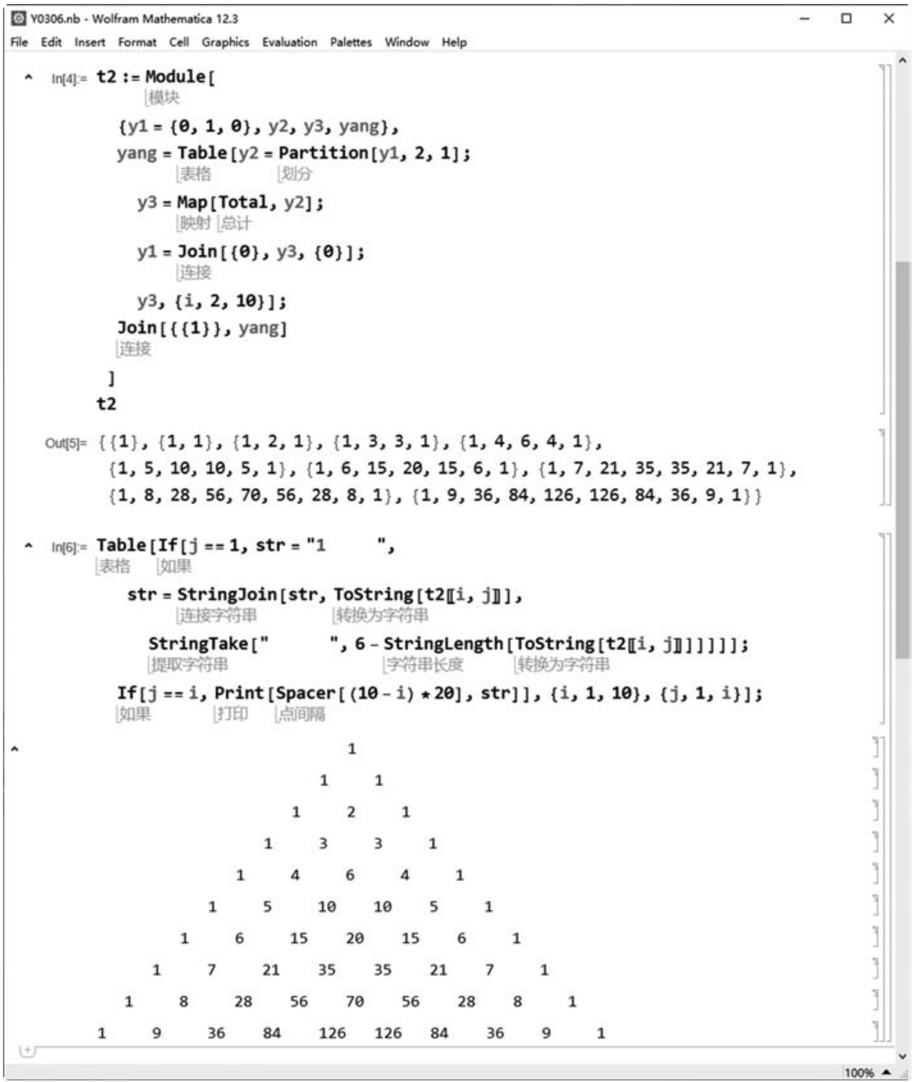


图 3-11 Table 函数包括语句组的实例

y3,{0}]; y3”9 次,每次循环中:先用 Partition 函数将 y1 分成包含两两一组的子列表的一个新列表,并赋给 y2,例如,若 y1={0,1,0},则 y2={{0,1},{1,0}},Partition 函数将在 4.1.1 节详细介绍;然后,执行“Map[Total,y2]”,并将结果赋给 y3,Total[列表]函数用于计算“列表”元素的累加和,而“Map[Total,y2]”将“Total”函数作用于 y2 的每个子列表上,即对其每个子列表求累加和,例如,y2={{0,1},{1,0}},则 y3={1, 1},y3 为杨辉三角的第 2 层元素;然后,语句“y1=Join[{0},y3,{0}]”借助于 Join 函数将两个 0 元素分别添加到 y3 的头部和尾部,并赋给变量 y1,如果 y3={1, 1},则 y1={0, 1, 1, 0};最后,y3 作为 Table 函数的输出。Module 模块的输出为“Join[{{1}},yang]”,即将列表“{1}”添加到列表 yang 的头部,因此由 Table 函数生成的 yang 仅包含杨辉三角的第 2~第 10 层,这里添加上第一层的元素。计算结果如“Out[5]”所示。

图 3-11 的“In[6]”中的 Table 函数的作用在于将 10 层的杨辉三角形输出。在该 Table

函数中,循环变量 i 从 1 按步长 1 递增到 10,对于每个 i,j 从 1 按步长 1 递增到 i,对于每一组 i 和 j,循环执行语句组“If[j==1, str="1 ", str = StringJoin[str, ToString[t2[[i,j]]], StringTake[" ", 6-StringLength[ToString[t2[[i,j]]]]]]]; If[j==i, Print[Spacer[(10-i)*20],str]]”,这里为两条 If 语句:①第一条 If 语句为“If[j==1, str="1 ", str = StringJoin[str, ToString[t2[[i,j]]], StringTake[" ", 6-StringLength[ToString[t2[[i,j]]]]]]]”,表示如果 j 为 1,则令 str="1 ",表示每一行显示的第一个元素;否则,随着 j 的递增,将同一行的其他元素合并到 str 中。其中,函数 StringJoin 用于连接多个字符串,使其成为一个字符串;ToString 函数将其参数转化为字符串;StringTake[字符串, n]表示从“字符串”中提取前 n 个字符的子字符串;StringLength[字符串]返回“字符串”的长度。②第二条 If 语句为“If[j==i, Print[Spacer[(10-i)*20],str]]”,表示当 j 等于 i 时输出一堆空格点和字符串 str,其中,空格点用函数 Spacer 生成。经过上述的格式化输出方法,可以得到如图 3-11 中所示的标准杨辉三角形。

3.1.3 数组与矩阵

元素为数值类型的列表均可以称为数组。如果认为数组元素可以为符号,那么列表均可以视为数组。而二维数组中的矩形数组,即二维数组的每行都具有相同的元素数,这类二维数组称为矩阵。数组中只有少量非零元素的数组称为稀疏数组;同样,矩阵中具有少量非零元素的矩阵称为稀疏矩阵。本节将介绍构造数组和矩阵的一些方法。

首先,介绍生成一维等差序列数组的函数 Range。Range 函数的语法如下:

(1) Range[i_{max}],生成列表{1, 2, …, i_{max}}。

(2) Range[i_{min}, i_{max}],生成列表{i_{min}, $i_{min}+1$, $i_{min}+2$, …, i_{max}}。

(3) Range[i_{min}, i_{max}, di],生成一个列表,第一个元素为 i_{min},步长为 di,最后一个元素为小于或等于 i_{max}、但是其与 di 之和大于 i_{max}。注意,i_{max} 可能不在生成的列表中。当 di 为 1 时可以省略,退化为第(2)种形式,但是当 di 为非 1 的数值或负数时,di 不能省略。

上述 Range 函数生成的序列公差相同。此外,Wolfram 语言中还有一个函数 PowerRange 可以生成公比相同的一维序列,PowerRange 函数的语法如下:

(1) PowerRange[b],生成列表{1, 10, 100, …, 10^k}, k=Floor[Log10[b]],即 k 为 b 的常用对数的整数部分。这个列表的公比为 10。

(2) PowerRange[a, b],生成列表{a, 10a, 100a, …, $10^k a$}, k 满足 $10^k a \leqslant b < 10^{k+1} a$,这个列表的公比为 10。

(3) PowerRange[a, b, r],生成列表{a, r×a, $r^2 a$, …, $r^k a$},k 满足 $r^k a \leqslant b < r^{k+1} a$,这个列表的公比为 r。

除了上述生成等差序列和等比序列的函数外,Wolfram 语言还提供了公差为零的一维等分函数 Subdivide,该函数的语法如下:

(1) Subdivide[n],生成 1 的 n 等分数组,即{0, 1/n, 2/n,…,1}。

(2) Subdivide[x_{max}, n],生成 0~x_{max} 的 n 等分数组,即{0, x_{max}/n, 2x_{max}/n, …, x_{max}}。

(3) Subdivide[x_{min}, x_{max}, n],生成 x_{min}~x_{max} 的 n 等分数组,即{x_{min}, ($x_{max}-x_{min}$)/n,

$2(x_{max}-x_{min})/n$，…，x_{max}}。

另一个快速创建特殊数组的函数为 CenterArray，其语法如下：

(1) CenterArray[n]，生成一个长度为 n 的一维列表，其中间位置(按 Floor[(n+1)/2]计算)的元素为 1，其余元素为 0。

(2) CenterArray[x, n]，生成一个长度为 n 的一维列表，其中间位置(按 Floor[(n+1)/2]计算)的元素为 x，其余元素为 0。

(3) CenterArray[x, n, y]，生成一个长度为 n 的一维列表，其中间位置(按 Floor[(n+1)/2]计算)的元素为 x，其余元素为 y。

上述 3 种语法均为生成一维列表，这 3 种语法均可以扩充至高维，其扩展后的语法如下：

(4) CenterArray[{n_1, n_2, …, n_k}]，CenterArray 第(1)种语法的扩展版本，生成大小为 $n_1\times n_2\times\cdots\times n_k$ 的数组，其中心位置(按(Floor[(n_1+1)/2], Floor[(n_2+1)/2], …, Floor[(n_k+1)/2])计算)的元素为 1，其余元素为 0。

(5) CenterArray[x, {n_1, n_2, …, n_k}]，CenterArray 第(2)种语法的扩展版本，生成大小为 $n_1\times n_2\times\cdots\times n_k$ 的数组，其中心位置(按(Floor[(n_1+1)/2], Floor[(n_2+1)/2], …, Floor[(n_k+1)/2])计算)的元素为 x，其余元素为 0。

(6) CenterArray[x, {n_1, n_2, …, n_k}, y]，CenterArray 第(3)种语法的扩展版本，生成大小为 $n_1\times n_2\times\cdots\times n_k$ 的数组，其中心位置(按(Floor[(n_1+1)/2], Floor[(n_2+1)/2], …, Floor[(n_k+1)/2])计算)的元素为 x，其余元素为 y。

图 3-12 展示了上述函数的典型用法。

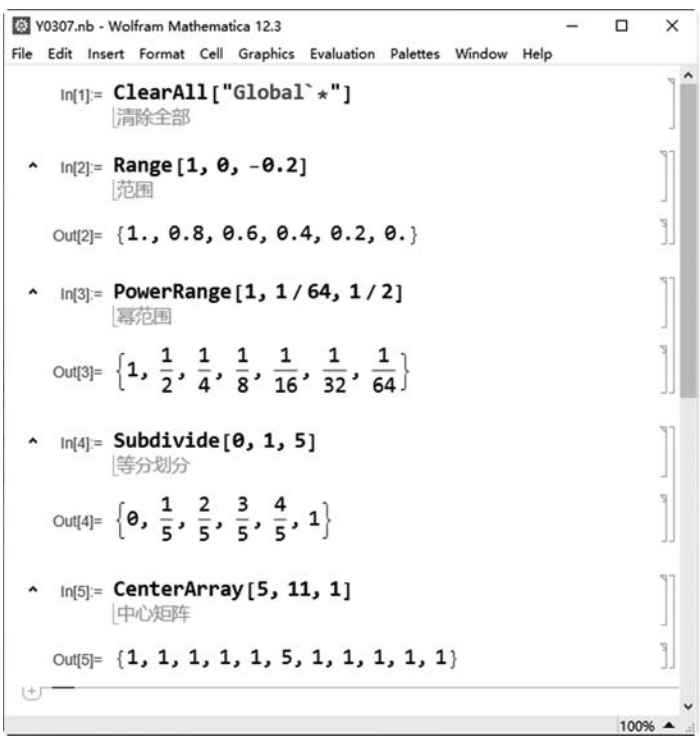


图 3-12　等差序列和等比序列的列表生成函数实例

在图 3-12 中，“In[2]”中的语句“Range[1，0，－0.2]”生成了一个首元素为 1、公差为－0.2 且最后一个元素为 0 的等差序列，如“Out[2]”所示。“In[3]”中的语句“PowerRange[1，1/64，1/2]”生成了一个首项为 1、公比为 1/2、尾项为 1/64 的等比序列，如“Out[3]”所示。“In[4]”中的语句“Subdivide[0,1,5]”将 0～1 等分为 5 等份得到一个序列，如“Out[4]”所示。“In[5]”中的语句“CenterArray[5,11,1]”生成了一个长度 11 的列表，中间位置的元素为 5，其余的元素均为 1，如“Out[5]”所示。

下面介绍一个生成稀疏数组的函数 SparseArray，该函数的主要语法如下：

(1) SparseArray[{pos_1－> val_1，pos_2－> val_2，…，pos_n－> val_n}，{d_1，d_2，…，d_k}]，生成一个维数为 $d_1 \times d_2 \times \cdots \times d_k$ 的稀疏数组，位置 pos_1 处的值为 val_1，pos_2 处的值为 val_2，…，pos_n 处的值为 val_n，其余元素的值为 0。

(2) SparseArray[{pos_1，pos_2，…，pos_n}－>{val_1，val_2，…，val_n}，{d_1，d_2，…，d_k}]，含义与(1)相同。

(3) SparseArray[{pos_1－> val_1，pos_2－> val_2，…，pos_n－> val_n}，{d_1，d_2，…，d_k}，val]，生成一个维数为 $d_1 \times d_2 \times \cdots \times d_k$ 的稀疏数组，位置 pos_1 处的值为 val_1，pos_2 处的值为 val_2，…，pos_n 处的值为 val_n，其余元素的值为 val。

(4) SparseArray[{pos_1，pos_2，…，pos_n}－>{val_1，val_2，…，val_n}，{d_1，d_2，…，d_k}，val]，含义与(3)相同。

(5) SparseArray[arr]，将一个普通的数组 arr 转化为稀疏数组。

在上述语法中，形如“pos1－> val1”的表达式称为规则 Rule，6.2.1 节将详细介绍。

在上述稀疏数组中常使用 Band 函数，Band 函数的语法如下：

(1) Band[{i,j}]表示在二维矩阵中从位置(i,j)开始，i 和 j 的值同时按步长 1 递增，直到矩阵的边缘，这条斜线上遍历的位置均属于 Band[{i,j}]。

(2) Band[{i_{min},j_{min}},{i_{max},j_{max}}]，表示在二维矩阵中从位置{i_{min},j_{min}}开始，i 和 j 的值同时按步长 1 递增，直到遇到 i_{max} 或 j_{max} 指定的边缘为止，这条斜线上的位置均属于此 Band。

(3) Band[{i_{min},j_{min}},{i_{max},j_{max}}，{di，dj}]，表示在二维矩阵中从位置{i_{min},j_{min}}开始，i 和 j 的值同时分别按步长 di 和 dj 递增，直到遇到 i_{max} 或 j_{max} 指定的边缘为止，这条斜线上的位置均属于此 Band。

图 3-13 和图 3-14 分别列出了稀疏数组函数 SparseArray 和 Band 函数的用法。

在图 3-13 中，“In[2]”调用了语句“m1＝SparseArray[{{1,2}－> 3,{2,4}－> 5,{4,3}－> 7},{5,5}]”，创建了一个 5×5 的稀疏矩阵 m1，其中，位置(1,2)的元素为 3，位置(2,4)的元素为 5，位置(4,3)的元素为 7，其余的元素为 0。在“Out[2]”中显示了 m1 的主要信息，共有 3 个非零元素，矩阵为{5,5}，表示大小为 5×5。使用 Normal 函数可以得到稀疏矩阵 m1 的普通形式，在“In[3]”中，使用 Normal 函数将 m1 转换为普通矩阵，然后，借助于函数 MatrixForm 显示该矩阵的内容，如“Out[3]”所示，可见，只有 3 个非零元素，分别位于位置(1,2)、(2,4)和(4,3)处。

在图 3-14 中，“In[4]”使用稀疏数组函数 SparseArray 的第(2)种语法生成一个稀疏矩阵 m2，稀疏矩阵仍然是矩阵，可以直接借助于 MatrixForm 函数查看其内容，如“In[5]”和“Out[5]”所示。稀疏矩阵 m2 只有 3 个非零元素，分别位于位置(2,1)、(4,4)和(5,3)处。在“In[6]”中，使用稀疏矩阵函数 SparseArray 的第(4)种语法生成一个稀疏矩阵 m3，m3 与

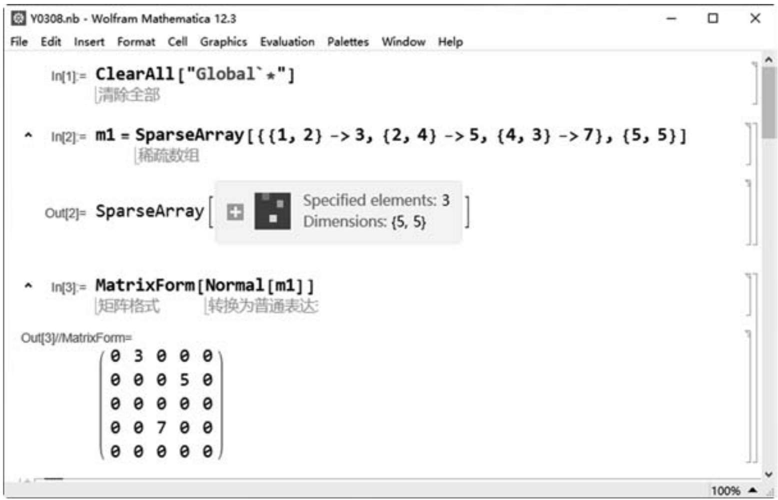


图 3-13　稀疏数组函数 SparseArray 用法实例

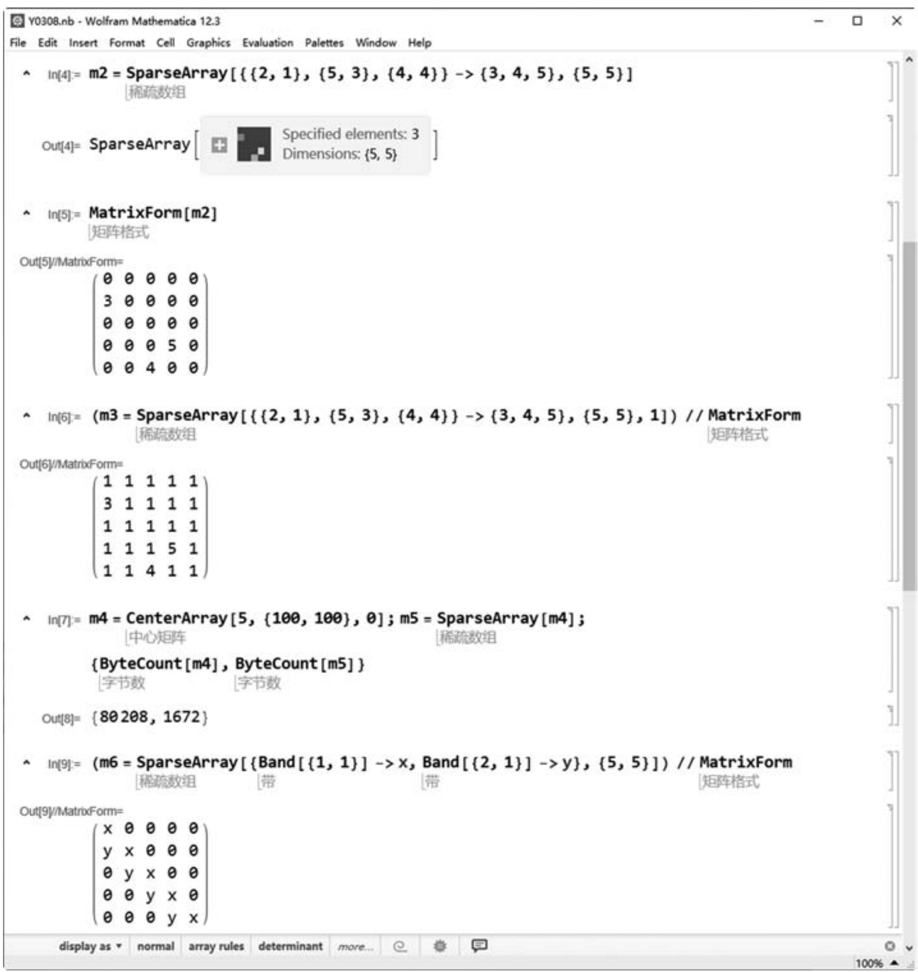


图 3-14　稀疏数组函数 SparseArray 和 Band 函数用法实例

m2 的指定位置元素相同,但是 m3 的非指定位置元素均为 1,如"Out[6]"所示。

稀疏矩阵具有存储优势,在"In[7]"中,使用 CenterArray 函数创建了一个大小为 100×100 的普通矩阵 m4,然后,借助于 SparseArray 将 m4 转化为稀疏矩阵 m5,接着,使用函数 ByteCount 计算 m4 和 m5 的存储字节数。由"Out[8]"可知,m4 占据 80 208 字节,而其对应的稀疏矩阵 m5 只占据 1672 字节。

在图 3-14 的"In[9]"中使用 Band 函数指定稀疏矩阵中的特定位置,这里 Band[{1,1}]对应于主对角线上的全部位置,而 Band[{2,1}]表示主对角线下方的次对角线上的全部位置。然后,在"In[9]"中,"{Band[{1,1}]-> x,Band[{2,1}]-> y}"表示主对角线上全部元素设为 x,其下方的次对角线上全部元素设为 y。创建的稀疏矩阵如"Out[9]"所示。

本节最后需要介绍的两个函数为"IdentityMatrix"和"DiagonalMatrix",分别用于生成单位矩阵和对角矩阵,它们的语法如下:

(1) IdentityMatrix[n],生成一个 n×n 的单位矩阵。

(2) IdentityMatrix[n, SparseArray],生成一个 n×n 的单位矩阵,以稀疏矩阵形式存储。

(3) DiagonalMatrix[一维列表],生成一个矩阵,其对角线元素为"一维列表"的元素,其他元素为 0。

(4) DiagonalMatrix[一维列表, k],生成一个矩阵,其第 k 条对角线上的元素为"一维列表"的元素,其余元素为 0。这里的第 k 条对角线的含义为:k=0 表示主对角线;k=1 表示主对角线上方元素位置(1,2)所在的斜对角线;当 k 为大于 1 的整数时,第 k 条对角线为主对角线上方元素位置(1,k+1)所在的斜对角线;当 k 为小于 0 的整数时,第 k 条对角线为主对角线下方元素位置(|k|+1,1)所在的斜对角线。

这两个函数的实例如图 3-15 所示。

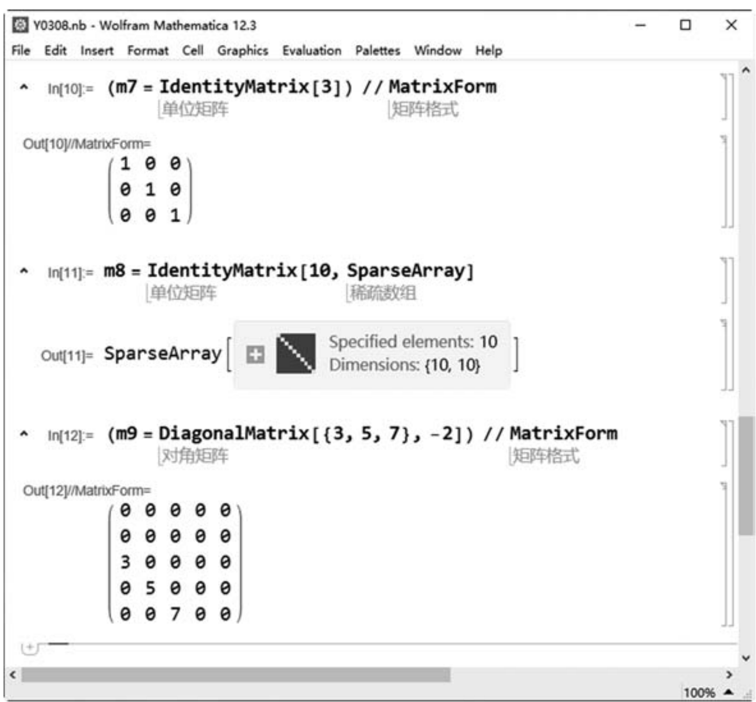


图 3-15 单位矩阵和对角矩阵实例

在图 3-15 中，“In[10]”使用函数 IdentityMatrix 生成了一个三阶单位矩阵 m7，如“Out[10]”所示，其主对角线元素为 1，其他元素为 0；“In[11]”生成了一个十阶单位矩阵，使用稀疏矩阵形式存储，如“Out[11]”所示；“In[12]”使用函数 DiagonalMatrix 生成一个矩阵 m9，该矩阵的元素位置(3,1)所在的斜对角线上的元素为{3,5,7}，其余位置的元素均为 0，如“Out[12]”所示。

除了上述介绍的各种数组和矩阵函数外，Wolfram 语言中还有一些几何变换矩阵和代数变换矩阵，这里限于篇幅不再介绍。

3.1.4 字符列表

字符串是用双引号括起来的一串符号，而字符是由双引号括起来的单个符号。在 Wolfram 语言中，函数 Characters 用于把字符串转化为字符列表，而函数 CharacterRange 可生成规则排序的字符列表。这两个函数的用法实例如图 3-16 所示。

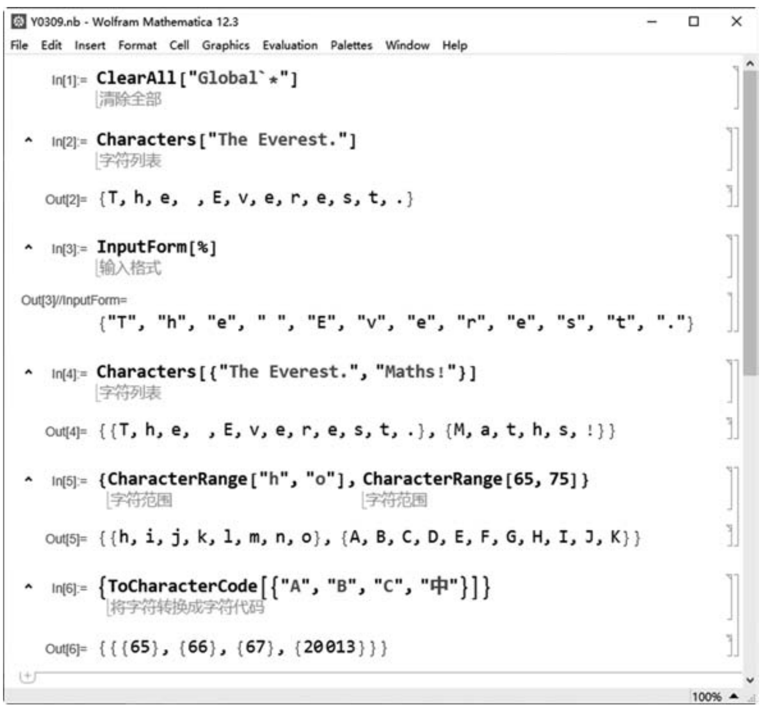


图 3-16　字符列表函数用法实例

在图 3-16 中，“In[2]”执行“Characters["The Everest."]”将字符串“The Everest.”转化为字符列表“{T，h，e，，E，v，e，r，e，s，t，.}”，如“Out[2]”所示。“Out[2]”中每个字符均带有双引号，其形式通过“InputForm”函数查看，如“Out3”所示。函数 Characters 具有“Listable”属性，其参数为列表时，函数 Characters 作用于列表的每个字符串上，如“In[4]”所示。“Characters[{"The Everest.", "Maths!"}]”将返回每个字符串的字符列表，如“Out[4]”所示。

CharacterRange 函数若指定两个字符作为参数，将生成这两个字符间按字符编码排列的全部字符列表；若指定两个整数作为参数，将生成这两个整数间的全部整数作为编码对

应的字符列表。如图 3-16 中“In[5]”所示的“{CharacterRange["h","o"], CharacterRange[65,75]}”,这里生成字符“h”至“o”的全部字符列表以及编码 65 对应的字符“A”至编码“75”对应的“K”的字符列表,如“Out[5]”所示。需要注意:CharacterRange 函数的参数的两个字符或两个数值,必须满足后者大于前者,否则返回空列表。这里的字符的编码是 Unicode 编码,查看一个字符的编码的方式为“ToCharacterCode[字符]”,该函数返回“字符”的编码。在图 3-16 的“In[6]”中,使用语句“{ToCharacterCode[{"A","B","C","中"}]}”显示字符“A”“B”“C”和“中”的编码,如“Out[6]”所示。

借助于英文字符的编码规律,即大写英文字符的编码加上 32 为其对应的小写英文字符的编码,可实现英文字符的大小写转换。例如:“FromCharacterCode[ToCharacterCode["A"]+32]”,将“A”转换为“a”。这里的函数“FromCharacterCode”为由编码得到相对应的字符的函数。Wolfram 语言中集成了实现英文字符大小写转换的函数,即函数 ToLowerCase 和 ToUpperCase,其参数均为字符或字符串,执行后将分别得到小写形式和大写形式的字符串。

3.1.5 随机数列表

随机数是一种重要的计算资源。Wolfram 语言中集成了各种随机数发生器函数,本节将详细介绍这些函数的实现方法。

首先是随机整数发生器函数 RandomInteger,其语法如下:

(1) RandomInteger[],随机生成 0 或 1。

(2) RandomInteger[i_{max}],随机生成一个位于{0, 1, 2, …, i_{max}}中的整数。

(3) RandomInteger[{i_{min}, i_{max}}],随机生成一个位于{i_{min}, $i_{min}+1$,…,i_{max}}中的整数。

(4) RandomInteger[{i_{min}, i_{max}}, n],随机生成一个长度为 n 的列表,每个元素均为随机取自{i_{min}, $i_{min}+1$,…,i_{max}}中的整数。

(5) RandomInteger[{i_{min}, i_{max}}, {n_1, n_2,…,n_k}],随机生成一个大小为 $n_1 \times n_2 \times \cdots \times n_k$ 的列表,每个元素为随机取自{i_{min}, $i_{min}+1$,…,i_{max}}中的整数。

(6) RandomInteger[概率分布函数],随机生成按“概率分布函数”分布的一个整数。

(7) RandomInteger[概率分布函数, n],随机生成一个长度为 n 的整数列表,其中的元素服从“概率分布函数”给定的分布规律。

(8) RandomInteger[概率分布函数, {n_1, n_2,…,n_k}],随机生成一个大小为 $n_1 \times n_2 \times \cdots \times n_k$ 的整数列表,其中的元素服从“概率分布函数”给定的分布规律。

上述 RandomInteger 函数的数值参数必须为整数,概率分布函数必须为定义在整数域上的分布。

伪随机浮点数发生器函数 RandomReal 的语法与 RandomInteger 有相似之处,如下所示:

(1) RandomReal [],随机生成区间[0, 1]中的一个浮点数。

(2) RandomReal [x_{max}],随机生成一个位于区间[0, x_{max}]中的浮点数。

(3) RandomReal [{x_{min}, x_{max}}],随机生成一个位于区间[x_{min}, x_{max}]中的浮点数。

(4) RandomReal [{x_{min}, x_{max}}, n],随机生成一个长度为 n 的列表,每个元素均为随机

取自区间[x_{min}，x_{max}]中的浮点数。

(5) RandomReal [{x_{min}，x_{max}}，{n_1，n_2，…，n_k}]，随机生成一个大小为 $n_1 \times n_2 \times \cdots \times n_k$ 的列表，每个元素为随机区间[x_{min}，x_{max}]中的浮点数。

(6) RandomReal [概率分布函数]，随机生成按“概率分布函数”分布的一个浮点数。

(7) RandomReal [概率分布函数，n]，随机生成一个长度为 n 的浮点数列表，其中的元素服从“概率分布函数”给定的分布规律。

(8) RandomReal [概率分布函数，{n_1，n_2，…，n_k}]，随机生成一个大小为 $n_1 \times n_2 \times \cdots \times n_k$ 的浮点数列表，其中的元素服从“概率分布函数”给定的分布规律。

在上述的第(6)～第(8)种情况下，“概率分布函数”必须为定义在浮点数域上的分布。

随机复数发生器函数 RandomComplex 的语法如下所示：

(1) RandomComplex []，随机生成一个复数，其实部和虚部均为区间[0，1]中的浮点数。

(2) RandomComplex [z_{max}]，随机生成一个复数，其位于复坐标系中以复数 z_{max} 为对角线的矩形内。

(3) RandomComplex [{z_{min}，z_{max}}]，随机生成一个复数，其位于复坐标系中以 z_{min} 和 z_{max} 为端点，以连接这两个端点的线段为对角线的矩形内。

(4) RandomComplex [{z_{min}，z_{max}}，n]，随机生成一个长度为 n 的复数列表，其每个元素位于复坐标系中以 z_{min} 和 z_{max} 为端点，以连接这两个端点的线段为对角线的矩形内。

(5) RandomComplex [{z_{min}，z_{max}}，{n_1，n_2，…，n_k}]，随机生成一个大小为 $n_1 \times n_2 \times \cdots \times n_k$ 的复数列表，其每个元素位于复坐标系中以 z_{min} 和 z_{max} 为端点，以连接这两个端点的线段为对角线的矩形内。

在 RandomReal 和 RandomComplex 函数中，可以指定浮点数的工作精度，使用选项“WorkingPrecision－>精度值”，其中，精度值可以为小数。

上述伪随机数发生器函数的典型实例如图 3-17 所示。

在图 3-17 中，“In[2]”调用了函数 SeedRandom，该函数将其参数设为伪随机数发生器的“种子”值(即初始值)，而所有的随机数函数均由同一个伪随机数发生器得到，因此，设定了“种子”值后，其后的各个伪随机数函数生成的随机数随“种子”值的确定而确定了。从而在任何计算机上(或任何时间)，图 3-17 中的执行结果都是相同的。如果没有“In[2]”中的语句“SeedRandom[299792458]”(这里的参数值为光在真空中的传播速率)，图 3-17 中的执行结果在不同的计算机上(甚至不同的执行时间)是不同的。

在图 3-17 中详细介绍了 RandomInteger 函数的各种用法，由于函数 RandomReal 与 RandomComplex 的用法和函数 RandomInteger 类似，这两个函数仅作了简单介绍。在“In[3]”中语句“{RandomInteger[]，RandomInteger[100]，RandomInteger[{10，20}]}”用于产生一个列表，如“Out[3]”所示，第一个元素 0 由“RandomInteger[]”生成；第 2 个元素 13 由“RandomInteger[100]”生成；第 3 个元素 17 由“RandomInteger[{10，20}]”生成。上述三个调用对应于 RandomInteger 函数的第(1)～第(3)种语法。

在“In[4]”中，“RandomInteger[{1，10}，6]”将生成一个长度 6 的列表，每个元素从{1，2，…，10}中随机取得，其执行结果如“Out[4]”所示。“In[5]”中“r1＝RandomInteger[{1，10}，{2，4}]”生成了 2×4 的矩阵，并赋给全局变量 r1，每个元素为随机取自 1～10 的整数，使用

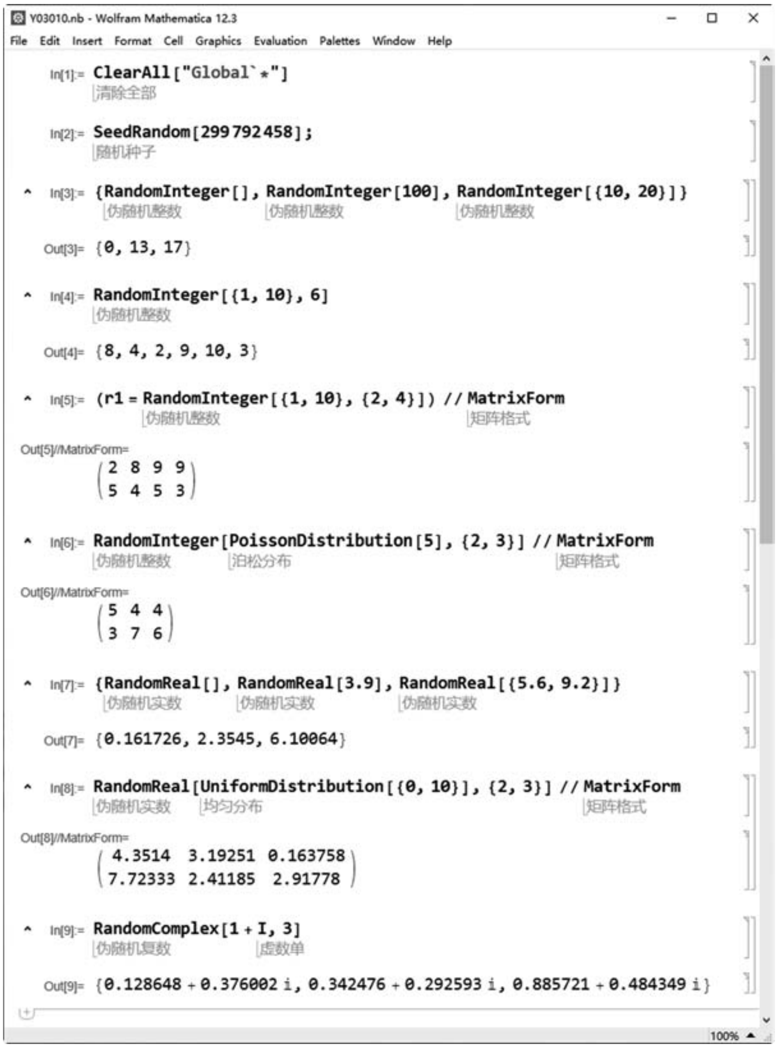


图 3-17　伪随机数发生器典型实例

MatrixForm 函数将 r1 显示为矩阵形式，如“Out[5]”所示。在“In[6]”中，“PoissonDistribution”为泊松分布函数，具有一个表示均值的参数，其中，“PoissonDistribution[5]”为一个均值为 5 的泊松分布。而“RandomInteger[PoissonDistribution[5],{2,3}]”生成一个大小为 2×4 的列表，其元素服从均值为 5 的泊松分布，然后，借助于 MatrixForm 函数显示为矩阵形式，如“Out[6]”所示。

在“In[7]”中，语句“{RandomReal[]，RandomReal[3.9]，RandomReal[{5.6,9.2}]}”对应着 RandomReal 函数第(1)～第(3)种语法，执行结果如“Out[7]”所示。在“In[8]”中，函数“UniformDistribution[{min,max}]”用于产生一个在区间[min，max]上的均匀分布，其中“RandomReal[UniformDistribution[{0,10}],{2,3}]”生成一个 2×3 的列表，其元素服从区间[0,10]上的均匀分布，执行结果借助 MatrixForm 函数以矩阵形式呈现，如“Out[8]”所示。

在图 3-17 的“In[9]”中，调用“RandomComplex[1+I，3]”生成一个包括 3 个复数的列

表，每个复数随机生成，且取自以复数“1+I”为对角线的矩阵区域内。执行结果如“Out[9]”所示。

在 Wolfram 语言中，综合了上述 RandomInteger[概率分布函数]和 RandomReal[概率分布函数]两个函数功能的函数称为 RandomVariate，该函数的语法如下：

(1) RandomVariate[概率分布函数]，按“概率分布函数”随机生成一个数值；

(2) RandomVariate[概率分布函数，n]，按“概率分布函数”随机生成一个包含 n 个数值的列表；

(3) RandomVariate [概率分布函数，{n_1，n_2，…，n_k}]，按“概率分布函数”随机生成一个大小为 $n_1 \times n_2 \times \cdots \times n_k$ 的数值列表。

这样，在图 3-17 中的“In[6]”等价于语句“RandomVariate[PoissonDistribution[5]，{2,3}]//MatrixForm”；而“In[8]”等价于语句“RandomVariate[UniformDistribution[{0，10}]，{2,3}]//MatrixForm”。

除了上述随机数发生器函数外，Wolfram 语言中还有一个生成随机颜色的函数 RandomColor，调用“RandomColor[n]”将返回一个长度为 n 的颜色列表，每个颜色值取自 RGBColor 函数的颜色空间。

Wolfram 语言中，可以将伪随机器发生器函数的作用局部化，通过 BlockRandom 函数实现。Block 模块函数将在 5.2 节深入介绍，它具有将全局变量的值局部化的作用，即不影响全局变量的全局值，在 Block 模块内部可以给该全局变量赋一个新的局部值。类似于 Block 函数，BlockRandom 函数使用的伪随机数发生器函数，对“伪随机数发生器”的“全局状态值”没有影响，这样，类似于下面的两个 RandomInteger 函数的调用一定会返回相同的伪随机数：

```
"{BlockRandom[RandomInteger[100]],RandomInteger[100]}"
```

每次执行其结果都会不同，但是，返回的两个随机数一定相同。假设执行到这个语句时，“伪随机数发生器”的当前状态值为 x，进入这个语句后，在“BlockRandom[RandomInteger[100]]”中“伪随机数发生器”将使用 x 进行迭代生成下一个伪随机数供这个 RandomInteger 函数使用，但是这个 RandomInteger 函数在 BlockRandom 中，它不会改变“伪随机数发生器”的原有状态 x。而在执行语句“RandomInteger[100]”，“伪随机数发生器”仍然是从状态“x”开始迭代，但是迭代之后，状态 x 被新的状态取代。

另一类型的随机函数为随机抽样函数，包括 RandomChoice 和 RandomSample 函数，前者为有放回的随机取样，即每次取样都是从全部样本空间中取样；后者为无放回的随机取样，即每次取样是从前面已取样后的剩余样本空间中取样。这两个函数的语法相近，下面一起介绍：

(1) RandomChoice[{e_1，e_2，…，e_n}]，从“{e_1，e_2，…，e_n}”中随机抽样一个数据，返回这个数据。

RandomSample[{e_1，e_2，…，e_n}]与上述函数意义完全不同，它是对“{e_1，e_2，…，e_n}”的全部元素作一个随机排列，返回一个包含重新排列后的全部元素的新列表。事实上，RandomSample[{e_1，e_2，…，e_n}]与 RandomSample[{e_1，e_2，…，e_n}，n]含义相同。

(2) RandomChoice[{e_1，e_2，…，e_n}，m]，从“{e_1，e_2，…，e_n}”中有放回地随机抽样 m 个

数据，组成一个列表，这里的 m 可以大于 n。

RandomSample[{e_1, e_2, …, e_n}, m]，从“{e_1, e_2, …, e_n}”中无放回地随机抽样 m 个数据，组成一个列表，这里的 m≤n，当 m=n 时，相当于“{e_1, e_2, …, e_n}”的一个全排列。

(3) RandomChoice[{e_1, e_2, …, e_n}, {m_1, m_2, …, m_k}]，生成一个大小为 $m_1 \times m_2 \times \cdots \times m_k$ 的列表，每个元素均从“{e_1, e_2, …, e_n}”中有放回地随机抽样得到。

RandomSample 函数无此对应语法。

(4) RandomChoice[{p_1, p_2, …, p_n}->{e_1, e_2, …, e_n}]，按可能出现的比例“p_1:p_2:…:p_n”从“{e_1, e_2, …, e_n}”中随机抽样一个数据，返回这个数据。这里的“p_1, p_2, …, p_n”可以为整数也可以为浮点数。

RandomSample[{p_1, p_2, …, p_n}->{e_1, e_2, …, e_n}]，与上述函数意义不同，按可能出现的比例“p_1:p_2:…:p_n”对“{e_1, e_2, …, e_n}”的全部元素作一个随机排列，返回一个包含重新排列后的全部元素的新列表。这种情况下，每个元素被抽样的概率越大，其出现在新列表中的位置越靠前。事实上，RandomSample[{p_1, p_2, …, p_n}->{e_1, e_2, …, e_n}]与 RandomSample[{p_1, p_2, …, p_n}-> {e_1, e_2, …, e_n}, n]含义相同。

(5) RandomChoice[{p_1, p_2, …, p_n}->{e_1, e_2, …, e_n}, m]，按可能出现的比例“p_1:p_2:…:p_n”从“{e_1, e_2, …, e_n}”中有放回地随机抽样 m 个数据，组成一个列表。

RandomSample[{p_1, p_2, …, p_n}->{e_1, e_2, …, e_n}, m]，按可能出现的比例“p_1:p_2:…:p_n”从“{e_1, e_2, …, e_n}”中无放回地随机抽样 m 个数据，组成一个列表。这里的 m≤n。当 m=n 时与“RandomSample[{p_1, p_2, …, p_n}->{e_1, e_2, …, e_n}]”含义相同。

(6) RandomChoice[{p_1, p_2, …, p_n}->{e_1, e_2, …, e_n}, {m_1, m_2, …, m_k}]，生成一个大小为 $m_1 \times m_2 \times \cdots \times m_k$ 的列表，每个元素均按它们出现的比例“p_1:p_2:…:p_n”从“{e_1, e_2, …, e_n}”中有放回地随机抽样得到。

RandomSample 函数无此对应语法。

图 3-18 列举了几个随机取样的实例。

在图 3-18 中，“In[2]”中语句“RandomChoice[Range[6]]”从{1, 2, …, 6}中随机选取一个数值；“RandomSample[Range[6]]”对{1, 2, …, 6}进行一次随机全排列。“In[2]”的执行结果如“Out[2]”所示。在“In[3]”所示的语句“{RandomChoice[Range[6], 4], RandomSample[Range[6], 4]}”中，两个函数都是随机从{1, 2, …, 6}中抽取 4 个数值，只是前者是有放回抽样，数据可能重复抽取到；后者是无放回抽样，数据不可能重复。“In[3]”的执行结果“Out[3]”也印证了上述分析。

在图 3-18 中，“In[4]”中的语句“RandomChoice[Range[6], {2, 3}]//MatrixForm”，有放回地从{1, 2, …, 6}中随机选取了 6 个数据，构成一个 2×3 的列表，以矩阵形式显示于“Out[4]”处。“In[5]”中，“RandomChoice[RandomInteger[10, 6]-> Range[6]]”以随机生成的权重从{1, 2, …, 6}中随机选取一个数值；而“RandomSample[RandomInteger[10, 6]-> Range[6]]”以随机生成的权重对{1, 2, …, 6}进行了一次随机全排列。“In[5]”的结果如“Out[5]”所示。“In[6]”与“In[5]”类似，只是从中随机抽样 4 个数据；同样地，“RandomChoice[RandomInteger[10, 6] -> Range[6], 4]”可能会重复抽样，而“RandomSample[RandomInteger[10, 6]-> Range[6], 4]”不会重复抽样。

在图 3-18 的“In[7]”中，语句“RandomChoice[RandomInteger[10, 6]-> Range[6],

图 3-18 随机取样函数实例

{2, 3}] //MatrixForm”，以“RandomInteger[10, 6]”生成的随机序列作为权重，从{1, 2, …, 6}中随机有放回地抽取 6 个数值，组成一个 2×3 的列表，以矩阵的形式显示在“Out[7]”中。“In[7]”对应着 RandomChoice 函数的第(6)种语法，而 RandomSample 在当前的 Mathematica 版本中无此功能。

3.2 列表操作

列表是 Wolfram 语言中的基础数据类型，是数据的组织方式，也是内置函数的主要处理对象。本节的“列表操作”仅介绍一些与列表元素处理直接相关的内置函数，那些使用列表进行数学计算的内置函数将在下一章介绍。

3.2.1 列表元素访问

在 Wolfram 中所有表达式都有一个标头，用函数“Head[表达式]”获取。列表的标头为“List”，例如，“Head[{1,2,3}]”返回“List”。列表的标头为列表的第 0 号位置的元素，通过 Extract 可以读出标头，函数“Extract[列表，{位置}]”读出该“位置”处的元素。借助于 TreeForm 可以查看列表的结构，“TreeForm[列表]”将以树和层的形式显示列表结果。“列表”的层数用 Depth 函数统计。用 Level 可读取列表的某一层的数据，例如“Level[列表，{层号}]”读出“列表”中第“层号”层的数据，当层号为−1 时，读出列表中的全部原子类型数据。下面借助于图 3-19 具体说明上述函数的用法。

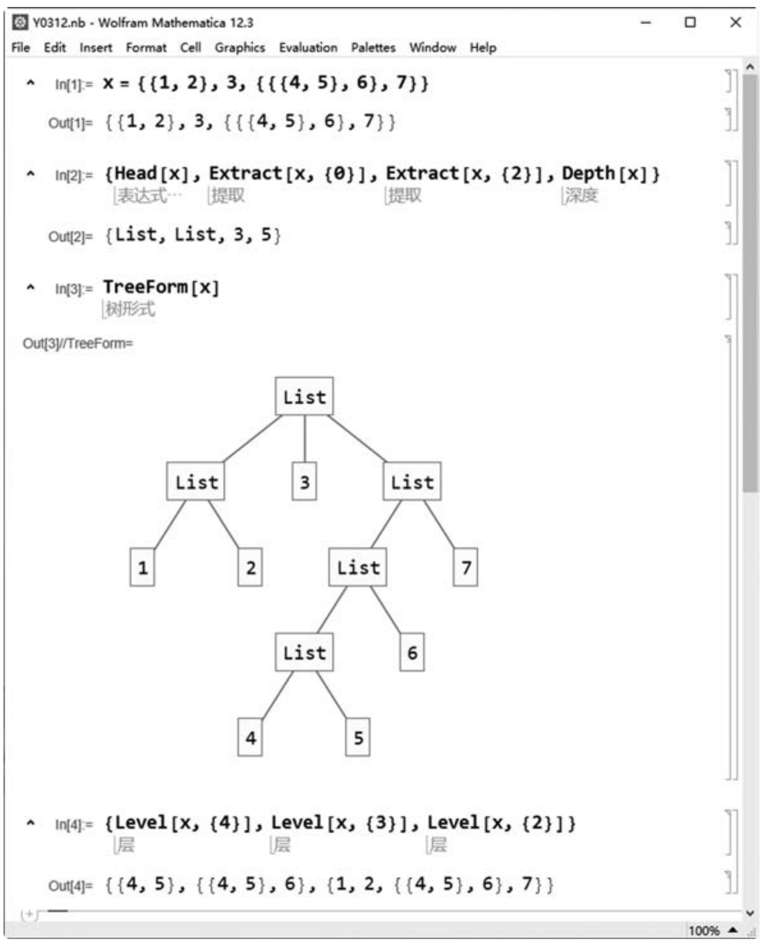


图 3-19 列表标头和层的函数

在图 3-19 中，“In[1]”定义了全局变量 x 为一个列表“{{1,2},3,{{{4,5},6},7}}”。在“In[2]”中，调用“Head[x]”和“Extract[x,{0}]”时均得到“List”，因为 x 是一个列表，其标头为 List，而标头 List 位于 x 的 0 号位置，故 Extract[x，{0}]也返回 List。“Extract[x，{2}]”返回列表 x 的第 2 个元素，由“In[1]”或“Out[1]”可知，列表 x 的第 2 个元素为 3。Depth[x]返回列表 x 的深度，也就是 x 的层数，由“Out[3]”可以看到 x 共 5 层，即深度为 5。

这样,“In[2]”的执行结果为“{List, List, 3, 5}”,如“Out[2]”所示。在“In[3]”中调用“TreeForm[x]”显示列表 x 的树形结构。注意两点:①TreeForm 可以用于显示任意表达式的树形结构;②树形结构中,层号从 0 开始标注,在“Out[3]”中,第 0 层就是整个 List 列表,访问方式“Level[x,{0}]”,将返回“{{{1,2},3,{{{4,5},6},7}}}”,相当于“{x}”,因为 Level 以列表的形式返回值,故多加了一层“{ }”;第 1 层为“List[1,2]”“3”和“List[List[List[4,5],6],7]”,访问方式为“Level[x,{1}]”,返回“{{1,2},3,{{{4,5},6},7}}”,相当于 x,实际上是返回 x 的各个元素,并添加了一层“{ }”,注意,如果某层中有一个 List,那么 List 下面的“树枝”和“树叶”都算在这一层中;第 2 层为“1”“2”“List[List[4,5],6]”和 7,访问方式为“Level[x,{2}]”,返回“{1,2,{{4,5},6},7}”;第 3 层为“List[4,5]”和 6,访问方式为“Level[x,{3}]”,返回“{{4,5},6}”;第 4 层为“4”和“5”,访问方式为“Level[x,{4}]”,返回“{4, 5}”。在“In[4]”中,语句“{Level[x,{4}],Level[x,{3}],Level[x,{2}]}”返回 x 的第 4、3、2 层上的内容,如“Out[4]”所示。

若拟访问列表中指定位置的元素,使用 Part 函数。在列表中,各个元素的索引号从左向右依次为“1、2、3、…”,而从右向左依次为“−1、−2、−3、…”。从列表的第 m 个元素向右到第 n 个元素,使用记号“m;;n”;如果从第 n 个元素向左至第 m 个元素,使用记号“n;;m;;−1”;如果从第 m 个元素以步长 k 到第 n 个元素,使用记号“m;;n;;k”。Part 函数的常用表示形式为双方括号对“[[ ]]”。对于多层列表,使用逗号“,”分别表示不同层的索引号。图 3-20 进一步演示了列表的元素访问方法。

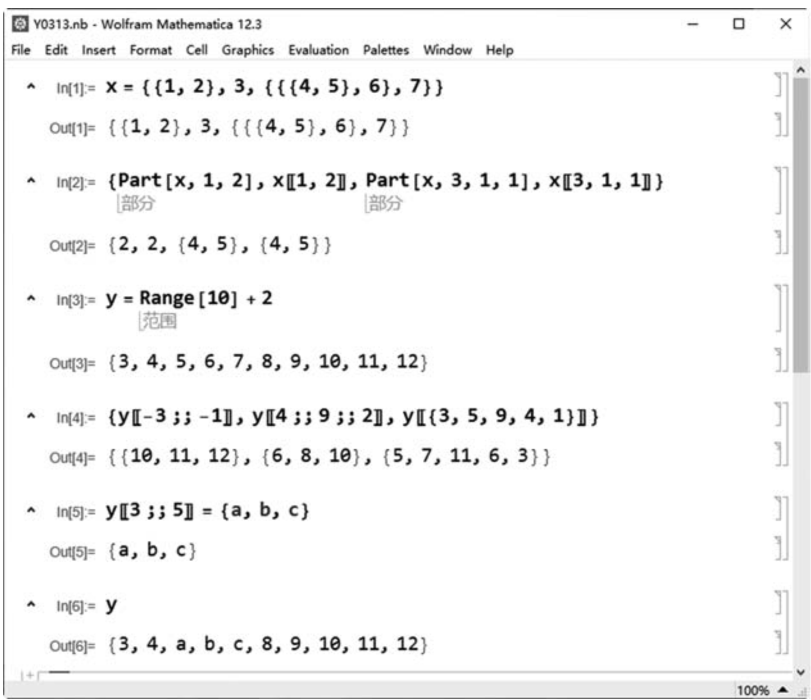


图 3-20 列表元素访问

在图 3-20 中,“In[1]”定义了 x 为列表“{{1,2},3,{{{4,5},6},7}}”。在“In[2]”中“Part[x,1,2]”和“x[[1,2]]”等价,都返回 x 列表的第 1 个元素(为列表)中的第 2 个元素,

为 2。图 3-20 中的紧凑形式的双方括号由“Esc＋[[＋Esc”键和“Esc＋]]＋Esc”键输入。在“In[2]”中的“Part[x,3,1,1]”和“x[[3,1,1]]”含义相同，都指返回 x 列表的第 3 个元素(为列表)中的第 1 个元素(仍为列表)中的第 1 个元素(仍然是列表)，即“{4, 5}”。“In[2]”的输出如“Out[2]”所示。

在“In[3]”中定义了列表 y 为“{3,4,5,6,7,8,9,10,11,12}”。在“In[4]”中“y[[－3;;－1]]”读取列表 y 的倒数(指从右向左数)的第 3 个元素按步长 1 至倒数的第 1 个元素；“y[[4;;9;;2]]”读取列表 y 的第 4 个元素按步长 2 至第 9 个元素(注：第 9 个元素取不到)；“y[[{3,5,9,4,1}]]”读取以列表“{3,5,9,4,1}”的元素为索引号的列表 y 中的元素。“In[4]”的输出为“{{10,11,12},{6,8,10},{5,7,11,6,3}}”，如“Out[4]”所示。

在“In[5]”中执行“y[[3;;5]]={a,b,c}”将列表{a, b, c}赋给列表 y 的第 3～第 5 个元素，然后，在“In[6]”中显示列表 y，其内容如“Out[6]”所示，为“{3,4,a,b,c,8,9,10,11,12}”。可见，y[[3;;5]]已被赋值为{a,b,c}，即可以通过赋值的方式修改列表元素的值。

除了按元素的索引号访问列表中的元素外，还可以根据元素获取它在列表中的索引号，用“Position[列表, 元素]”函数获得“元素”在列表中的所有索引号，用“FirstPosition[列表，元素]”函数获得“元素”在列表中的第一个索引号，也就是首先出现的位置。谓词函数“MemberQ[列表, 元素]”当“元素”在“列表”内时，返回真；否则返回假。函数“Sort”可对列表排序，默认为升序排列；函数“Ordering”和“Sort”是一对函数，“Ordering”函数用于保存排序后的元素在原列表中的索引号。图 3-21 列出了上述函数的一些典型用法。

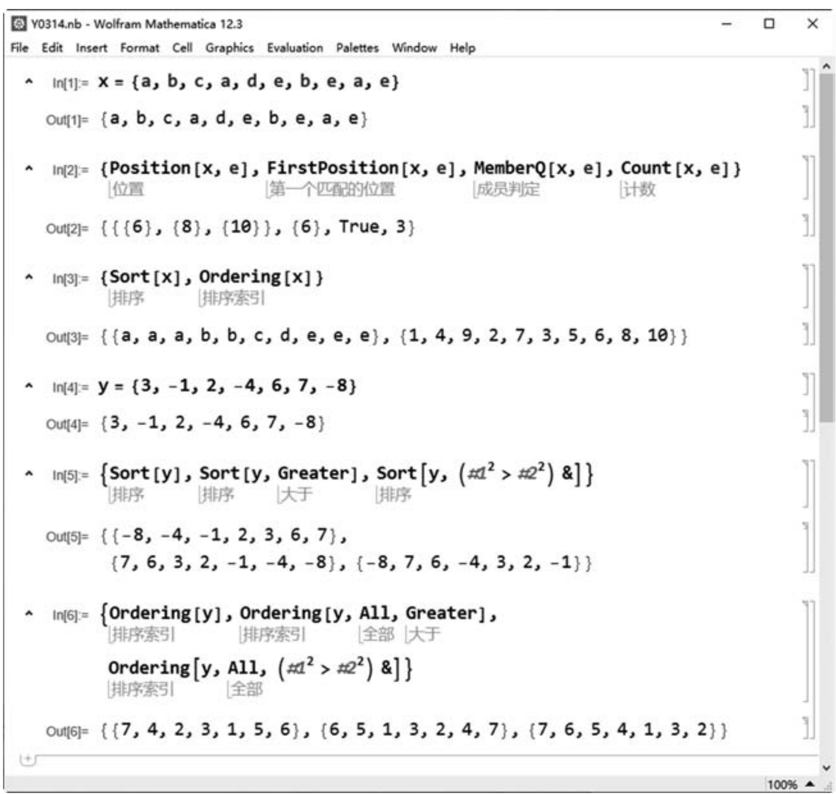


图 3-21 列表索引号相关的函数实例

在图 3-21 中,“In[1]”定义了列表 x 为“{a,b,c,a,d,e,b,e,a,e}”,在“In[2]”中“{Position[x,e],FirstPosition[x,e],MemberQ[x,e],Count[x,e]}”这 4 个函数的调用依次返回元素 e 在列表 x 中的全部索引号{{6},{8},{10}}、元素 e 在列表 x 中的第一个索引号{6}和元素 e 在列表 x 中出现的次数 3。函数 Count[列表,元素]用于统计“元素”在“列表”中出现的次数。函数 Position 和 FirstPosition 返回的结果均为列表的形式,这是因为对于多维列表而言,这两个函数的返回结果将不止一个数值,故把返回的全部索引号放在列表中。在“In[3]”的语句“{Sort[x],Ordering[x]}”中,“Sort[x]”对列表 x 以升序(这里按字符的编码值升序)排序,“Ordering[x]”获取排序后的元素索引号序列,运行结果如“Out[3]”所示。

Sort 函数可以指定排序方式。在图 3-21 中,“In[4]”定义了列表 y 为“{3,−1,2,−4,6,7,−8}”,在“In[5]”中,“{Sort[y], Sort[y,Greater], Sort[y,(#12>#22)&]}”依次对 y 按数值大小升序排列、按数值大小降序排列和按平方数大小降序排列,结果如“Out[6]”所示,为“{{−8,−4,−1,2,3,6,7},{7,6,3,2,−1,−4,−8},{−8,7,6,−4,3,2,−1}}”。Sort 函数可以指定第二个参数,为排序选项。当该选项为“Greater”时,表示以“Greater[#1,#2]&”即前一个元素大于后一个元素的方式排序。这里“Greater[#1,#2]&”为纯函数;如果选项为纯函数“(#12>#22)&”,表示按第一个元素的平方数大于第二个元素的平方数的顺序排序。可以为 Sort 函数的第二个参数指定所需要的排序纯函数实现特定的排序。当 Sort 函数使用排序选项时,相应的 Ordering 函数也需要使用相同的选项。例如,“In[6]”中的“Ordering[y,All,(#1^2>#2^2)&]”与“In[5]”中的“Sort[y,(#1^2>#2^2)&]”对应,在“Ordering”函数中,多了一个中间参数,这里该参数为“All”表示显示全部索引号,如果该参数为具体的整数,则只显示指定的整数个索引号。“In[5]”和“In[6]”的输出结果分别为“Out[5]”和“Out[6]”,对比这两个结果,可以直观地确认列表元素排序的变化与其引起的索引号的变化的关系。

3.2.2 Map 和 Apply 方法

Map 函数和 Apply 函数增强了访问列表元素的方法。其中,Map 函数的语法如下:

(1) Map[函数, 列表],表示“函数”将作用于列表的每个元素,相当于作用于列表第 1 层的元素。也可以用符号“/@”表示,即“函数/@列表”的形式。这里的“@”表示函数映射,即“函数[列表]”等价于“函数@列表”。

(2) Map[函数, 列表, {层号}],表示“函数”作用于“列表”的第“层号”层元素。

(3) Map[函数, 列表, 层号],表示“函数”作用于“列表”的第 1 层至第“层号”层的全部元素。

(4) Map[函数][列表],与第(1)种情况“Map[函数, 列表]”含义相同,有时称“Map[函数]”为“函数”的 Map 算子形式。

Apply 函数的语法如下:

(1) Apply[函数, 列表],表示使用“函数”替换“列表”的标头,即列表的第 1 层的全部元素作为“函数”的参数。也可以用符号“函数@@列表”表示。

(2) Apply[函数, 列表, {1}],表示使用“函数”替换“列表”的第 1 层的子列表的标头,也可以用符号“函数@@@列表”表示。

（3）Apply[函数，列表，{层号}]，表示使用“函数”替换“列表”的第“层号”层的子列表的标头。

（4）Apply[函数，列表，层号]，表示使用“函数”替换“列表”的第1层至第“层号”层的子列表的标头。

（5）Apply[函数][列表]，等价于第(1)种情况的“Apply[函数，列表]”，有时称“Apply[函数]”为“函数”的 Apply 算子形式。

图 3-22 给出了 Map 和 Apply 函数的典型实例。

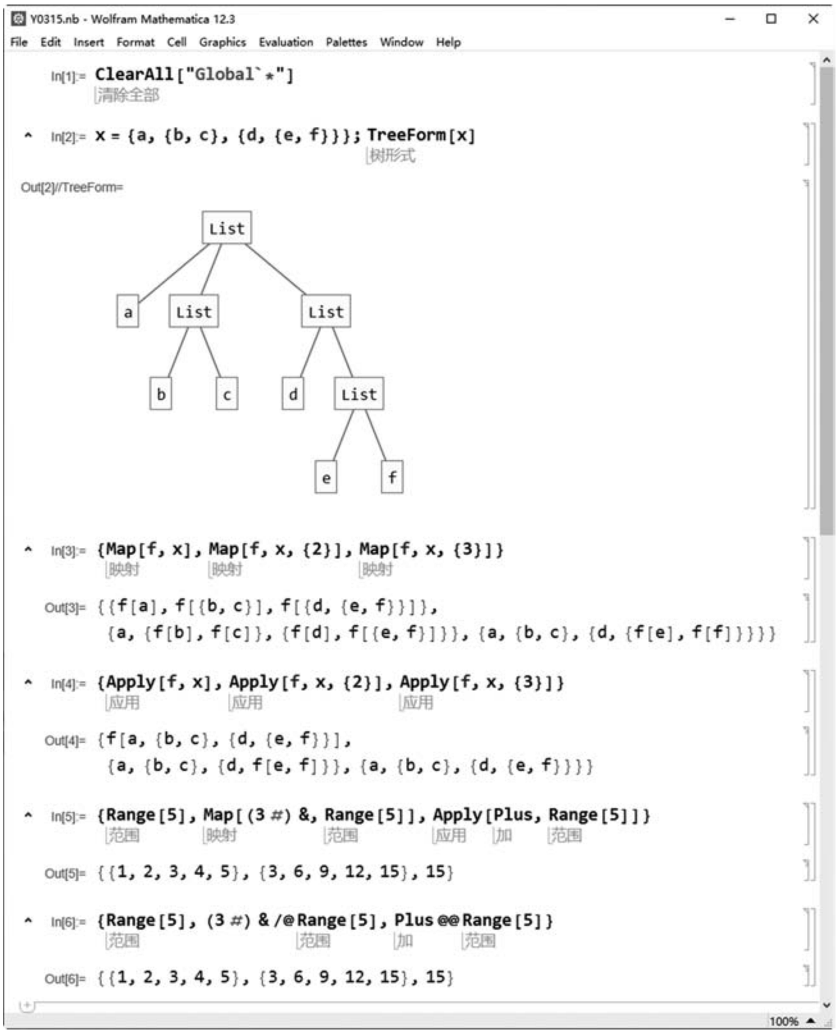


图 3-22　Map 和 Apply 函数典型实例

在图 3-22 中，“In[2]”定义了列表 x 并调用 TreeForm 显示了它的树形结构，如“Out[2]”所示，列表 x 共有 4 层，从第 0 层至第 3 层。由于“Map[f, x]”相当于“Map[f, x, {1}]”，所以，“In[3]”的语句“{Map[f,x], Map[f,x,{2}], Map[f,x,{3}]}”依次为将函数 f 作用于列表 x 的第 1、2 和 3 层上的元素，结果如“Out[3]”所示。这里以“Map[f, x, {2}]”为例详细介绍一下计算方式：由“Out[2]”可知，列表 x 的第 2 层为“b, c, d, List[e,f]”“Map[f, x, {2}]”，其中

函数 f 是作用于 x 的第 2 层上的元素，与第 2 层不相关的“树枝”和“树叶”上的元素将保留原样，故“Out[2]”中的元素“a”保留原样，所以，“Map[f, x, {2}]”将得到一个列表“{a,{f[b],f[c]},{f[d],f[{e,f}]}}”。

Apply 函数与 Map 函数的作用方式不同，在图 3-22 的“In[4]”中，“{Apply[f, x], Apply[f, x, {2}], Apply[f, x, {3}]}”表示依次将函数 f 替换列表 x 的第 1、2 和 3 层上的标头，这里的“Apply[f, x]”和“Apply[f, x, {1}]”含义相同。“In[4]”的执行结果如“Out[4]”所示。这里以“Apply[f, x, {2}]”为例详细介绍一下执行过程：列表 x 的第 2 层为“b, c, d, List[e,f]”，这里只有最后一个元素为列表，因此，Apply 函数仅把最后一个元素的标头替换为 f，其余元素保持不变，即得到“{a,{b,c},{d,f[e,f]}}”。在“In[3]”和“In[4]”中，有两个符号“f”，这不影响 Map 和 Apply 函数的正确执行，因为 Wolfram 语言可执行符号运算，这两个符号“f”是相同的，但是上述介绍中，把 Map 函数中的“f”称为函数，列表 x 中的“f”称为元素。

在图 3-22 的“In[5]”和“In[6]”中，列举了两个数值计算的 Map 和 Apply 实例，“In[5]”和“In[6]”含义相同，执行结果也相同，如“Out[5]”或“Out[6]”所示。其中，“Map[(3#)&,Range[5]]”或“(3#)&/@Range[5]”将纯函数“(3#)&”作用于列表“{1,2,3,4,5}”的每个元素，得到“{3,6,9,12,15}”，这里的纯函数“(3#)&”返回函数的参数的值的 3 倍。“Apply[Plus,Range[5]]”和“Plus@@Range[5]”含义相同，均表示使用函数“Plus”替换列表“{1,2,3,4,5}”的标头，即得到“Plus[1,2,3,4,5]”，为 15，这里的“Plus”函数为求和函数。

3.2.3 向量与矩阵操作

在 Wolfram 语言中，一维列表为列向量的形式，二维列表为矩阵的形式。本节将介绍向量与矩阵的基本运算，关于矩阵的变换等高级运算将在 4.1.4 节中介绍。

Wolfram 语言的基本运算函数和相应的简化符号为：加“Plus(+)”、减“Subtract(−)”、乘“Times(*)”、除“Divide(/)”、乘方“Power(^)”和开平方“Sqrt($\sqrt{\ }$)”，开平方的符号用“Ctrl+@”或“Ctrl+2”快捷键输入。这些运算均可以应用于向量与矩阵中。

图 3-23 为向量和矩阵的一些基本运算操作。

在图 3-23 中，在“In[1]”中“{2+3,2−3,2*3,2/3,2^3,$\sqrt{3}$}”演示了四则运算的应用，在 Wolfram 语言中，整数和有理数是无限精度表示和计算的，所以，“$\sqrt{3}$”的计算结果仍然为“$\sqrt{3}$”，如果查看这个无理数的浮点数近似值，需借助于“N”函数，“N[$\sqrt{3}$]”可显示“$\sqrt{3}$”的 6 位有效数字近似值。“In[2]”和“In[3]”使用随机整数发生器函数 RandomInteger 生成了两个长度为 5 的列表 x1 和 x2，以及两个 2 行 3 列的矩阵 y1 和 y2，分别如“Out[2]”和“Out[3]”所示。

在图 3-23 中，“In[4]”调用 Min 和 Max 函数计算了列向量 x1 和矩阵 y1 的最小值和最大值，如“Out[4]”所示。Wolfram 语言中还有一个函数“MinMax[列表]”，可以同时得到“列表”的最小值和最大值。在“In[5]”中，调用 Length 和 Dimensions 函数，分别获取了列表的长度(列表的第一层的元素个数)和矩阵的维数(矩阵的行和列的大小)。对于向量 x1，Dimensions 将返回其长度“{5}”；对于规则的矩阵 y1，将返回其行数和列数，即“{2,3}”。

在 Wolfram 语言中，“*”号，即 Times，用于向量或矩阵的操作时，是指向量或矩阵的

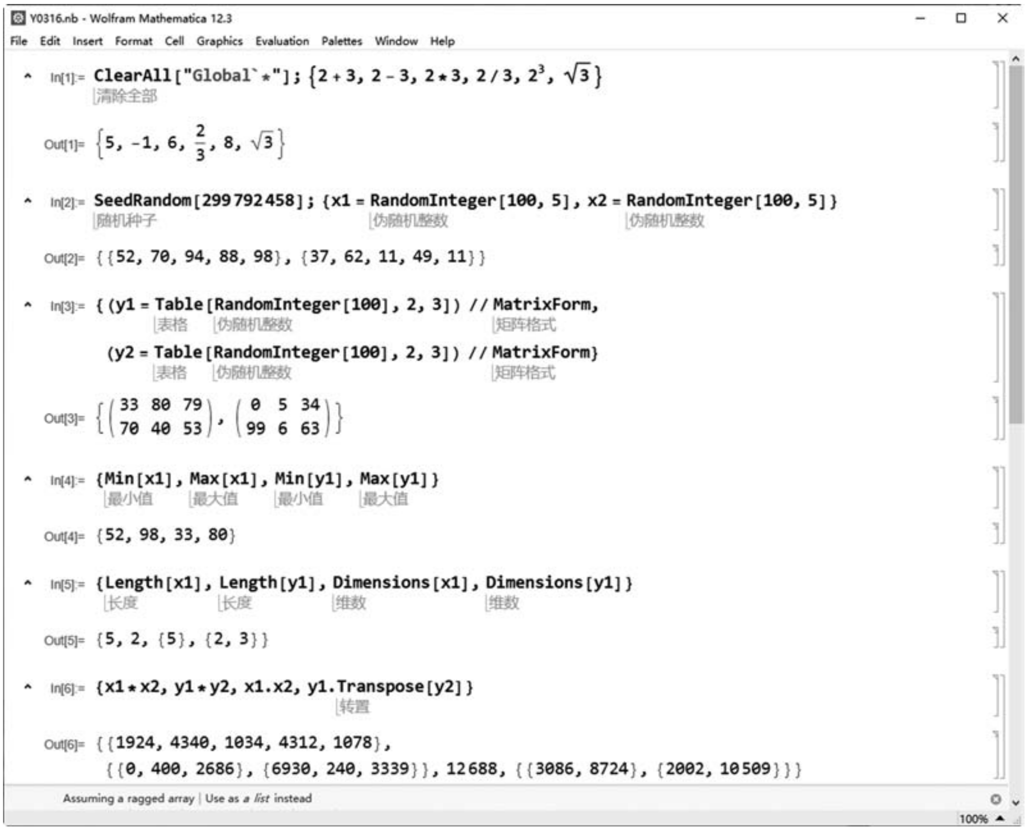


图 3-23　向量与矩阵的基本运算实例

对应元素相乘，此时要求参与相乘的两个向量的长度相同或者参与相乘的两个矩阵的结构相同(即具有相同的大小)。在图 3-23 的“In[6]”中，“x1 * x2”和“y1 * y2”实现了对应元素相乘的操作，返回的结果为“{1924，4340，1034，4312，1078}”和“{{0，400，2686}，{6930，240，3339}}”。而实现“矩阵乘法”的运算函数或符号为“Dot”或“.”，“x1. x2”即“Dot[x1，x2]”，执行 x1 与 x2 的内积运算(又称点积运算)，得到“12688”。“y1. Transpose[y2]”执行 y1 与 y2 的转置的矩阵乘法运算，得到“{{3086，8724}，{2002，10509}}”。这里的函数“Transpose[矩阵]”返回“矩阵”的转置。“In[6]”的执行结果如“Out[6]”所示。Dot 函数的扩展版本为“Inner”函数，称为广义内积函数，它的对偶操作函数为“Outer”函数，称为广义外积函数，这里不作详细说明。

3.2.4 集合操作

集合是由一组互不相同的元素组成的集体。对于列表而言，如果其元素互不相同，则可认为该列表为集合。集合的基本操作为求两个集合的并集和交集以及求一个集合的子集和它相对于全集的补集等。下面介绍基本的集合操作函数。

函数 DeleteDuplicates 用于删除列表中的重复元素，即重复出现的元素仅保留一个，其语法如下：

(1) DeleteDuplicates[列表]，删除“列表”中的重复元素，那些重复出现的元素仅保留

一个。

(2) DeleteDuplicates[列表, 条件],删除“列表”中满足“条件”的元素,而不是删除“列表”中的重复元素。

函数 Union 用于求列表的并集,其语法如下:

(1) Union[列表],与“DeleteDuplicates[列表]”作用类似,删除了“列表”中的重复元素,但是,还对“列表”中剩下的元素进行排序。

(2) Union[列表 1, 列表 2,…],对参数中的全部“列表”求并集,删除结果集合中的重复元素,并对结果集合中的元素进行排序。“Union”函数的运算符为符号“∪”(“Esc+un+Esc”键),“Union[列表 1, 列表 2,…]”等价于“列表 1 ∪ 列表 2 ∪ …”。

函数 Intersection 用于求列表的交集,其语法如下:

Intersection[列表 1, 列表 2,…],对参数中的全部“列表”求交集,删除结果集合中的重复元素,并对结果集合中的元素进行排序。“Intersection”函数的运算符为符号“∪”(“Esc+inter+Esc”键),“Intersection[列表 1, 列表 2,…]”等价于“列表 1 ∩ 列表 2 ∩ …”。

函数 Complement 函数用于求列表关于全集的补集,其语法如下:

Complement[表示全集的列表, 列表 1, 列表 2,…],返回在“表示全集的列表”中但不在“列表 1”“列表 2”等中的元素的集合,并对元素进行排序。

在上述求集合的并、交和补的函数中,可以使用“SameTest”相同性选项重新定义两个元素是否相同。例如,对于 Union 函数,增加了相同性选项的语法为:Union[列表 1, 列表 2, …, 列表 n, SameTest->用于判定相同的纯函数]。

图 3-24 给出一些集合的基本操作。

在图 3-24 中,“In[1]”定义了两个列表 x1 和 x2,分别为“{d,c,b,c,d,e,a,b,c}”和“{d,d,b,b,f,f}”。在“In[2]”中,语句“{DeleteDuplicates[x1], Union[x1], Union[x1, x2], Intersection[x1, x2], Complement[x1, x2]}”中各个函数依次实现了:删除 x1 中的重复元素、删除 x1 中的重复元素并对元素排序、求 x1 和 x2 的并集并对元素排序、求 x1 和 x2 的交集并对元素排序、求 x2 相对于 x1 的补集并对元素排序,其结果如“Out[2]”所示。

在图 3-24 中,“In[3]”生成两个序列 y1 和 y2,分别为“{6, 9, 1, 10, 5, 8, 7, 10, 3, 1}”和“{2, 5, 7, 2, 1, 1, 6, 8, 6, 8}”。“In[4]”中语句“{DeleteDuplicates[y1], Union[y1], Union[y1, y2], Intersection[y1, y2], Complement[y1, y2]}”依次实现了:删除 y1 中的重复元素、删除 y1 中的重复元素并排序、求 y1 和 y2 的并集并排序、求 y1 和 y2 的交集并排序、求 y2 相对于 y1 的补集并排序,其结果如“Out[4]”所示。

在“In[5]”“In[6]”和“In[7]”中对函数“Union”“Intersection”和“Complement”添加了选项“SameTest->((Mod[#1,5]==Mod[#2,5])&)”,表示如果两个数模 5 的余数相等,则判定其“相同”,按这个“相同性”判定条件分别在“In[5]”“In[6]”和“In[7]”中执行集合的并、交和补运算,其结果分别如“Out[5]”“Out[6]”和“Out[7]”所示。这里以“In[5]”中的“Intersection[y1, y2, SameTest->((Mod[#1,5]==Mod[#2,5])&)]”为例作详细介绍:在“相同性”条件下,y1 等价于列表“{6(1), 9(4), 1, 10(0), 5(0), 8(3), 7(2), 10(0), 3, 1}”,这里括号中的值表示模 5 后的余数;y2 等价于列表“{2, 5(0), 7(2), 2, 1, 1, 6(1), 8(3), 6(1), 8(3)}”。这样,在两个列表中所有的“6(1)和 1”是“相同”的,所有的“10(0)、5(0)”是“相同”的,所有的“7(2)、2”是“相同”的,所有“8(3)、3”是“相同”的。“相同”

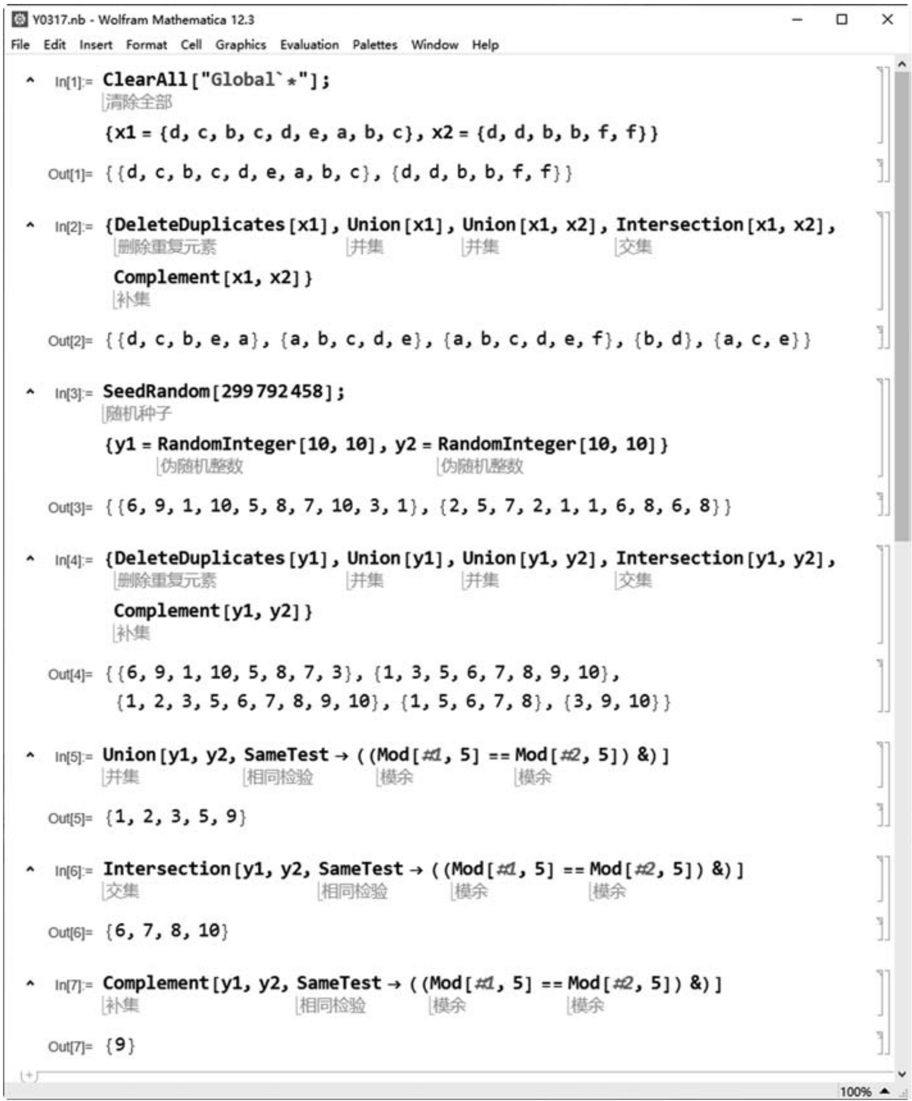


图 3-24 集合的基本操作实例

的元素在求交集时只保留 1 个元素，“In[5]”的执行结果为“{6，7，8，10}”，在模 5 求余的情况下相当于“{1，2，3，0}”，但是需要使用原列表中的元素。问题在于：既然“6”和“1”在模 5 求余的情况下是相同的，求交集应保留哪个呢？然而，Wolfram 语言并没有给出规律，从“In[5]”～“In[7]”返回的结果估计为：对于求模取余“相同”的元素，当求并集时，保留较小的数值；求交集时，保留较大的数值；而求补集时，保留较小的数值。这就是“In[6]”执行后得到“Out[6]”(即“{6，7，8，10}”，而不是{1，2，3，5})的原因。

最后，讨论一个求子集函数 Subsets 的功能，其语法如下：

(1) Subsets[列表]，返回“列表”的所有子集(包含空集)，此函数将“列表”中的重复元素视为有效元素，即这不是数学意义上求“集合”的全部子集的方法。如果进行数学意义上的求“集合”的子集的处理，需要使用 Union 函数或 DeleteDuplicates 将“列表”转换为

"集合"。

(2) Subsets[列表, {n}],返回"列表"中包含 n 个元素的所有子集,"列表"中的重复元素视为有效元素。

(3) Subsets[列表, n],返回"列表"中包含 0～n 个元素的所有子集,"列表"中的重复元素视为有效元素。

(4) Subsets[列表, {i_{min}, i_{max}}],返回"列表"中包含 i_{min}～i_{max} 个元素的所有子集,"列表"中的重复元素视为有效元素。

(5) Subsets[列表, {i_{min}, i_{max}, di}],返回"列表"中包含 i 个元素的所有子集,这里的 i 的取值为自 i_{min} 按步长 di 增加到 i_{max}(可能取不到)的所有整数,"列表"中的重复元素视为有效元素。

图 3-25 为函数 Subsets 的典型用法实例。

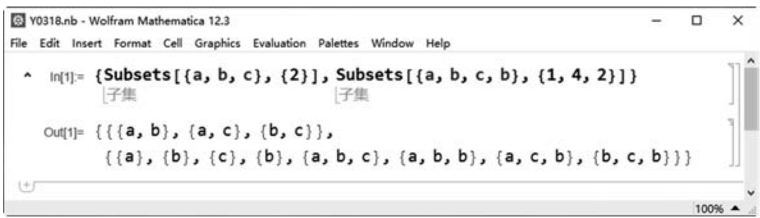


图 3-25　函数 Subsets 的典型用法实例

在图 3-25 中,"In[1]"中的"Subsets[{a, b, c},{2}]"取得列表"{a, b, c}"的包含 2 个元素的所有子列表,即"{{a, b},{a, c},{b, c}}",而"Subsets[{a, b, c, b},{1,4,2}]"生成列表"{a, b, c, b}"的包含 1 个元素或 3 个元素的所有子列表,即"{{a}, {b}, {c}, {b}, {a, b, c}, {a, b, b}, {a, c, b}, {b, c, b}}"。"In[1]"的执行结果如"Out[2]"所示。

本章小结

本章介绍了 Wolfram 语言中最重要的数据结构——列表,详细阐述了列表的结构、列表的创建、有规则列表的生成方法、数组和矩阵及其基本运算、字符列表和随机数列表的设计等,并深入讨论了列表元素的存储访问方法和集合的操作方法。本章的重点在于 Table 函数的应用技巧和列表元素的访问方法,需要做大量的编程练习以熟练掌握 Table 函数和 Part 函数的应用。除了本章介绍的列表操作函数外,Wolfram 语言还有大量用于列表处理的内置函数,例如,与列表元素的提取、增加、删除、压平和分解等相关的一些函数,这些内容将在下一章中作为内置函数的应用技巧详细介绍。除了列表之外,关联和数据集也是 Wolfram 语言中重要的数据结构,这部分内容需要用到纯函数,故将纯函数也安排在下一章中阐述。