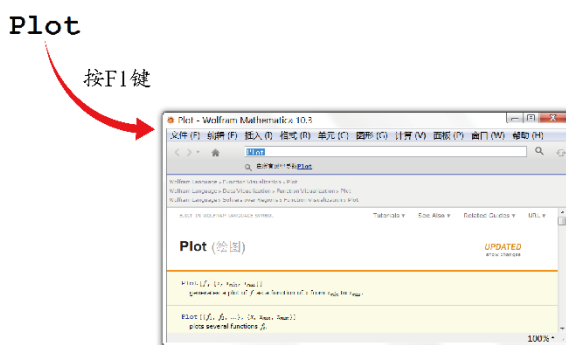


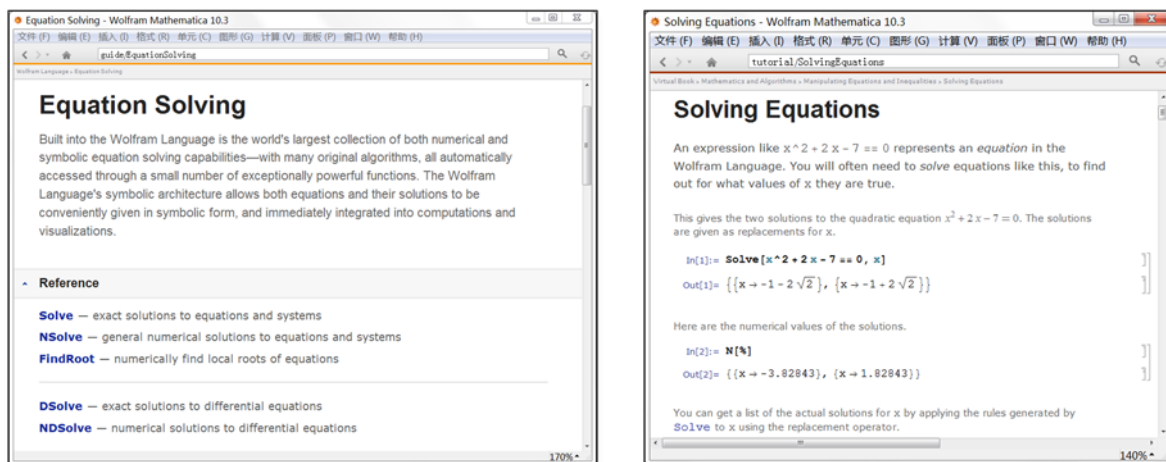
一些小窍门

1. 如何获取帮助

Mathematica 在安装时就自带了详细的帮助文档，其中包含函数的使用说明和大量的例子，按 F1 键即可调出。如果你了解某个函数的用法，只需要选中该函数（光标移动到函数名上任意位置即可），再按 F1 键就能进入该函数的帮助页面。



如果不知道该用哪个函数，你可以根据需求搜索。例如你想解一个方程，可以在帮助文档中搜索“solve equations”（解方程）。在搜索结果中你可以找到应该使用哪个函数，如下图左所示。根据你方程的类型应该选择相应的函数（是代数方程还是微分方程？如果是代数方程，是线性方程还是非线性方程？想要近似的数值解还是精确的符号解？）。你还能了解更多基础知识，例如在 Mathematica 中怎样定义一个方程，如下图右。



如果在自带的帮助文档中找不到想要的，你可以在网站 mathematica.stackexchange.com 上搜索。如果仍然没有，你还可以在该网站上提问（类似于百度知道）。对你的问题感兴趣的网友会给出解决方法。解答人多是 Mathematica 的爱好者，身份以教师和研究生为主，他们免费提供帮助，并从别人的回答中交流经验并相互学习。你的问题描述越具体，得到的解答会越贴切。但由于是义务解答，所以请给出你尝试过的解决方法。尽量避免只描述你的问题，把“解题过程”抛给别人。网站 <http://demonstrations.wolfram.com/> 上有很多志愿者提供的交互式小程序，其代码可以免费下载（点击右侧的 Download Author Code），通过分析别人的代码能够快速提高。你也可以进入聊天室和别人讨论你遇到的问题 <http://chat.stackexchange.com/rooms/2234/wolfram-mathematica>

2. 函数名自动补齐

Mathematica 以函数名过长而臭名昭著（比如这个函数：`BetaNegativeBinomialDistribution`）。但幸运的是你不需要将函数名完整地打出来，只需要打出几个首字母，软件会给出智能提示，选择即可。



3. 停止当前计算

如果你想中止正在运行的程序（例如程序可能陷入死循环），有两种方法：

- ① 按快捷键 **Alt + .**，中止程序。这样停止后变量的值依然存在。
- ② 退出内核。要小心，因为这样做所有的变量定义和计算结果都会被清空。

4. 常用快捷键

◆ 退出内核

退出内核的操作方法是在“计算”菜单中选择“退出内核—local”后确认即可。也可以执行 `Quit[]` 函数。但这样都太麻烦，我们可以给它设置一个快捷键，过程如下：

找到 Mathematica 安装目录下的 `KeyEventTranslations.tr` 文件，比如我的在这里：

`C:\Program Files\Wolfram Research\Mathematica\10.3\SystemFiles\FrontEnd\TextResources\Windows`
用记事本打开它，找到 `EventTranslations[{` 语句，在它后面插入以下内容并保存：

```
Item[KeyEvent["q",Modifiers->{Control}],FrontEndExecute[FrontEndToken[SelectedNotebook[],"EvaluatorQuit",Automatic]]],
```

以后我们只需按下 **Ctrl + q** 就能退出内核了。（重启软件或退出内核后生效）

注意 2 点：1. 结尾的逗号不要丢；2. 语句必须要放在 `EventTranslations[{` 后面，如下图

```
8 EventTranslations[{
9
10 Item[KeyEvent["q",Modifiers->
    {Control}],FrontEndExecute[FrontEndToken[SelectedNotebook[],"EvaluatorQuit",Automatic]]]
11
12 (* Evaluation *)
13 Item[KeyEvent["Enter"], "EvaluateCells"],
14 Item[KeyEvent["KeypadEnter"], "EvaluateCells"],
15 Item[KeyEvent["Return", Modifiers -> {Shift}], "HandleShiftReturn"],
```

说明：`Quit[]`和 `Clear["Global`*"]`命令都可以清空符号变量。如果我们想在程序开始时预先清空所有变量再初始化，应该用 `Clear["Global`*"]`，而不能用 `Quit[]`，否则后面的程序根本不会运行。

◆ 双层方括号

双层的方括号表示按索引位置从列表中取出元素，右图所示的是一个简单的例子，从列表 `a` 中取出第二个元素。

取元素是常用的操作，但它的输入有些繁琐，需要按 4 次键盘。其实，双方括号 `[[]]` 有一种更简洁的写法，就是 `[[]]`。我们可以为其设置快捷键，设置过程与“退出内核”相同，在 `KeyEventTranslations.tr` 文件的 `EventTranslations[{` 后加入以下语句即可：

```
In[7]:= a = {3, 6, 9};
a[[2]]
a[[2]]

Out[8]= 6
Out[9]= 6
```

```
Item[KeyEvent["[", Modifiers ->
{Control}], FrontEndExecute[{FrontEnd`NotebookWrite[FrontEnd`InputNotebook[], "
\[LeftDoubleBracket]", After]}]],

Item[KeyEvent["]", Modifiers ->
{Control}], FrontEndExecute[{FrontEnd`NotebookWrite[FrontEnd`InputNotebook[], "
\[RightDoubleBracket]", After]}]],
```

这样我们按下 **Ctrl + [** 和 **Ctrl +]** 就能输入双层方括号了。

◆ 添加注释

Mathematica 使用 (*这里是注释*) 表示注释的内容，其快捷键是 **Alt + /**

添加注释的方法就是选中要注释的语句然后按快捷键，去掉注释也用这个快捷键。

5. 自定义变量名

由于 Mathematica 所有的函数名（和一些常数，例如 E, Pi）首字母都是大写的。为了避免冲突，我们定义自己的变量和函数时尽量使用小写的首字母，例如：myFunction, controlParameter。

C 语言中可以在变量名定义中使用下划线，例如 student_Name, student_Age，这样更清晰，看到名字就能猜出来意思。但在 Mathematica 中，下划线专用于表示函数的参数，不能出现在变量名中。我们可以用符号 ` 替换下划线，例如：student`Name = Tom, student`Age = 18，这样是合法的。（当然这只是一选择，实际上直接用 studentName, studentAge 就可以了，使用 ` 有些多余）

6. 数据类型

Mathematica 与 Matlab 的一个区别是，Mathematica 本质上是一款符号计算软件。我们可以通过一个例子理解这一点。下图展示了两中计算方式，得到结果是一样的

```
In[1]:= a = N[Table[Tan[Sin[ $\pi$  + i]], {i, 10}]]
Out[1]:= {-1.11894, -1.28451, -0.142064, 0.944384, 1.42509,
0.286922, -0.771288, -1.52155, -0.437152, 0.604909}

In[2]:= b = Table[Tan[Sin[N[ $\pi$  + i]]], {i, 10}]
Out[2]:= {-1.11894, -1.28451, -0.142064, 0.944384, 1.42509,
0.286922, -0.771288, -1.52155, -0.437152, 0.604909}
```

但是如果增加计算长度（将 10 改为 100000），对比它们的计算时间：

```
In[1]:= a = N[Table[Tan[Sin[ $\pi$  + i]], {i, 100 000}]]; // AbsoluteTiming
Out[1]:= {1.3689, Null}

In[2]:= b = Table[Tan[Sin[N[ $\pi$  + i]]], {i, 100 000}]]; // AbsoluteTiming
Out[2]:= {0.0291681, Null}
```

我们发现前者所需的时间几乎是后者的 50 倍。原因就在于后者在计算时将 $\pi + i$ 视为近似的浮点数（函数 N 的功能是转化为浮点数），而前者在计算 $\pi + i$ 时使用的是精确值。对于我们来说，1.0 和 1 没有区别，更不影响计算结果，但在 Mathematica 中，二者是不同的数据类型（存储和表示方式也就不

同)：1.0 是浮点数，而 1 则是整数（可以用 Head 函数查看。类似的，1/2 和 0.5 也不同，前者是精确的有理数，后者是有限精度的浮点数）。不同数据类型对应的运算速度不同（虽然结果可能是一样的）。所以在变量赋值或计算时应注意，如果只需要浮点数计算，就尽量加上小数点（或者在计算时多使用 N 函数），以免中间按照符号计算导致程序异常缓慢。

我们可以用 Head 函数查看 {1,1.0, 1/2,Pi,Pi+1,Pi+1.0} 中数据的类型分别是：

```
In[5]:= Head /@ {1, 1.0, 1 / 2, Pi, Pi + 1, Pi + 1.0}
Out[5]= {Integer, Real, Rational, Symbol, Plus, Real}
```

说明：以圆周率 π 为例，对于大多数的数值计算，我们只需要取一个近似值，例如 3.1415926。没有必要纠结近似值与无理数 π 本身的区别，因为这种精度对我们来说已经足够了。但在符号运算中，则需要严格区分（Sin[π]必须严格等于 0）。

7. Mathematica 能设置断点吗？

Matlab 不只是个计算软件，它也是个开发平台，用户可以通过设置断点的方式调试代码，观察程序运行状况。而 Mathematica 的断点设置很难使用，所以如果程序出错，你只能使用 Break 和 Print 等函数试探找出可能出错的地方，这个过程可是相当痛苦。不过在 Mathematica 之外有一款调试平台，叫 Wolfram Workbench，但使用体验并不友好。

8. Mathematica 支持面向对象的编程吗？

Matlab 支持面向对象的编程范式，你可以定义类，类之间可以继承，其 Robotics Toolbox 工具箱就采用了这种方式。而 Mathematica 对面向对象技术的支持并不好，所以尽量不要用。

9. 函数前置和后置写法

Mathematica 的函数用法比较灵活，传统指定输入参数的方法是使用方括号[]，例如：Sin[π]

方括号的写法有些繁琐，我们也可以使用@ 符号，例如：Sin@ π

我们还可以使用 // 符号将函数和输入参数倒过来写，例如： π //Sin

以上三种写法得到的结果是一样的，但是使用后两种写法时应注意与其它运算符的优先级，例如下图左所示的加法。为保险起见，我们可以用小括号将输入项包裹起来，如下图右所示：

In[1]:= Sin@ π + 1.0	In[1]:= Sin@ (π + 1.0)
Out[1]= 1.	Out[1]= -0.841471
In[2]:= Sin[π + 1.0]	In[2]:= Sin[π + 1.0]
Out[2]= -0.841471	Out[2]= -0.841471
In[3]:= π + 1.0 // Sin	In[3]:= (π + 1.0) // Sin
Out[3]= -0.841471	Out[3]= -0.841471

初学者容易犯的错误

1. MatrixForm 函数

Mathematica 使用列表 (List) 统一表示集合、数组、向量、矩阵等各种数据对象，这也方便我们统一地处理数据。但如果计算结果仍是以列表的形式显示的，我们理解和阅读起来比较困难。为此 Mathematica 定义了一些函数，用来将数据以方便人阅读的方式显示出来，其中 MatrixForm 函数的功能就是将矩阵以行列的形式显示出来，例如：

```
In[1]:= a = {{1, 2, 3}, {4, 5, 6}};

In[2]:= MatrixForm[a]

Out[2]/MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```

但是，注意 MatrixForm 只能用来查看矩阵，在赋值时不要用。假如我们定义 $b = \text{MatrixForm}[a]$ ，再对 b 进行矩阵操作是错的，因为此时 b 已经不是一个矩阵列表了，如下图所示：

```
In[12]:= b = MatrixForm[a]

Out[12]/MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$


In[17]:= Transpose[b]

Out[17]= Transpose[ $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ ]

In[18]:= Transpose[a]

Out[18]= {{1, 4}, {2, 5}, {3, 6}}
```

说明：如果觉得每次都用 MatrixForm 显示矩阵太麻烦，可以运行以下命令

```
$PrePrint = # /. {expr_ /; Head[expr] != List >:> expr, m_?MatrixQ >:> MatrixForm[m]} &;
```

这样每次矩阵都会以行列形式显示，缺点是每次退出内核后都要重新执行一次上述命令

2. 无穷大

Matlab 中，符号 Inf 表示无穷大， $1/\text{Inf}$ 等于 0， $1/0$ 则等于 Inf。

Mathematica 中用 Infinity 表示无穷大，但在 Mathematica 中计算 $1/0$ 会报错：

```
1 / 0

Power::infy : Infinite expression  $\frac{1}{0}$  encountered. >>

ComplexInfinity
```

无穷大相减也会报错：

Infinity - Infinity

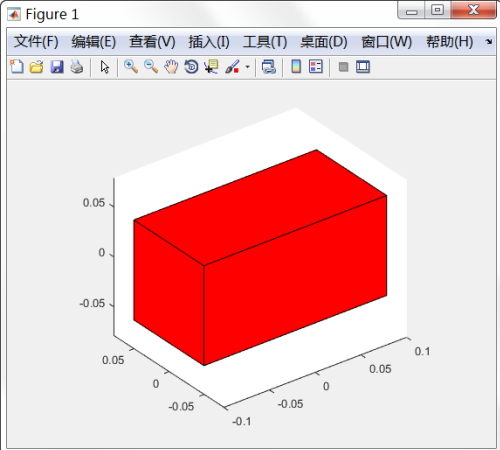
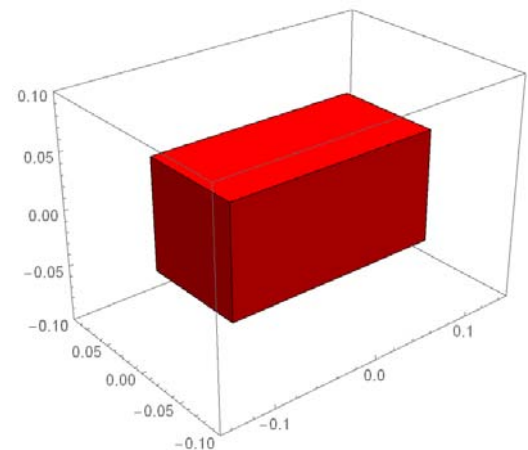
Infinity::indet : Indeterminate expression $-\infty + \infty$ encountered. >>

Indeterminate

但在 Matlab 中无穷大相减（等于 NaN）不会报错。这也说明 Matlab 的底层计算是以数值的方式，而 Mathematica 的底层计算是以符号的方式。

3. 画图

在 Matlab 中画三维立体图形需要自己写代码，例如我们想画一个长方体，我们可以定义长方体的顶点和面，然后借助 patch 函数绘制长方体。而在 Mathematica 中，只需要利用自带的 Cuboid 函数即可。代码和显示效果如下表所示。在 Matlab 中我们用结构体定义了一个长方体，而在 Mathematica 中，为了与 Matlab 保持一致，我们基于 Cuboid 函数定义了一个自己的长方体函数。

Matlab	Mathematica
<pre>sl=[0.2 0.1 0.1]; % 定义长方体 cuboid 的边长 sidelength（长宽高） cuboid.vertex=0.5*[-sl(1) -sl(2) -sl(3); -sl(1) sl(2) -sl(3); sl(1) sl(2) -sl(3); sl(1) -sl(2) -sl(3); -sl(1) -sl(2) sl(3); -sl(1) sl(2) sl(3); sl(1) sl(2) sl(3); sl(1) -sl(2) sl(3)]; %定义长方体的 8 个顶点 cuboid.face=[1 2 3 4;2 6 7 3;4 3 7 8;1 5 8 4;1 2 6 5;5 6 7 8]; %定义长方体的 6 个面 cuboid.color=[1 0 0]; %定义长方体的颜色为红色 patch('faces',cuboid.face,'vertices',cuboid.vertex,'FaceColor',cuboid.color) axis equal; view(3);</pre>	<pre>(*cuboid 函数的输入为长方体的中心坐标、边长和颜色*) cuboid[center_, sl_, color_] := {color, Cuboid[center - sl/2, center + sl/2]}; Graphics3D[cuboid[{0, 0, 0}, {0.2, 0.1, 0.1}, Red], PlotRange -> {{-0.15, 0.15}, {-0.1, 0.1}, {-0.1, 0.1}}, Axes -> True]</pre>
	

通过上例的对比可以发现，Mathematica 相比 Matlab 作图更方便、代码更简洁，而且显示效果更好。这符合 Mathematica 创作者的口号——避免重新发明轮子，减少无意义的重复性劳动。

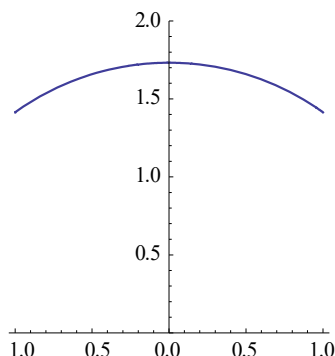
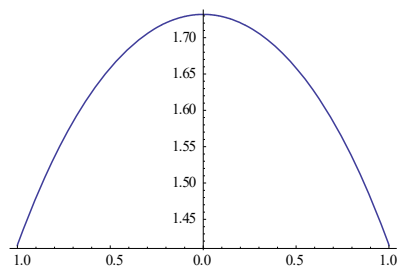
函数 Graphics (和 Graphics3D) 用于将图形显示出来。多个同类图形可以放在一起显示，但注意都要放在大括号中，并用逗号隔开，例如：

`Graphics[{Disk[], Red, Triangle[]}]`

在显示时，有时默认选项不是我们期望的效果，例如：

`Plot[Sqrt[3 - x^2], {x, -1, 1}]`画出的图形如右图所示。

该图有 2 个问题：首先 y 轴不是从 0 开始显示、其次 x 轴和 y 轴比例不一致。为此我们要设置显示选项：`Plot[{Sqrt[3 - x^2]}, {x, -1, 1}, AspectRatio -> Automatic, PlotRange -> {0, 2}]`

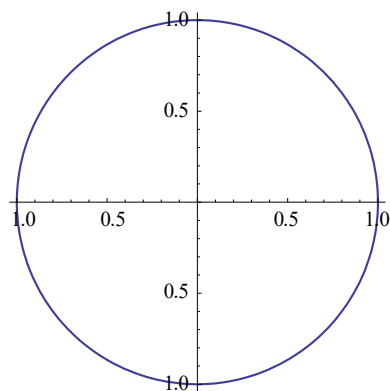


此时，x,y 轴比例是 1:1，
而且 y 轴从 0 开始

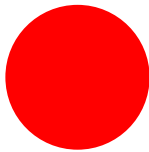
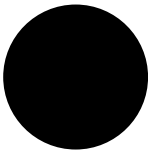
想同时显示多个图形可以用 Show 函数，但注意最终显示的效果采用第一个图像的设置，例如：

`Show[ParametricPlot[{Sin[u], Cos[u]}, {u, 0, 2Pi}], Plot[-x^2 + 3, {x, -3, 3}]]`的显示效果如右图所示。

我们发现只显示了前者 ParametricPlot 部分，而第二幅图 Plot 没有显示，原因是：ParametricPlot 中的 PlotRange 在作怪，修改选项增大显示范围即可：`Show[ParametricPlot[{Sin[u], Cos[u]}, {u, 0, 2 Pi}], Plot[-x^2 + 3, {x, -3, 3}], PlotRange -> {{-3, 3}, {-3, 3}}]`。



图形的显示属性很多，例如颜色，线宽等等。注意在设置属性的时候要放在对象的前面，放在后面是无效的，可以对比下面两个语句的显示效果：

<code>Graphics[{Red, Disk[]}]</code>	<code>Graphics[{Disk[], Red}]</code>
	

4. 所见非所得

Matlab 的编程界面 Editor 是纯文本编辑器，所见即所得（看到的是什么，实际就是什么）。而 Mathematica 的编程界面 Notebook 支持更复杂的数学公式显示，所见非所得。在下图所示的例子中

In[1]:= $a \rightarrow b$
In[2]:= $a \rightarrow b$

2 个输入中的箭头看上去完全一样,但如果用 FullForm 函数来显示它们的完全格式就会发现差别:

```
In[3]:= FullForm[a → b]
Out[3]/FullForm=
  RightArrow[a, b]

In[4]:= FullForm[a -> b]
Out[4]/FullForm=
  Rule[a, b]
```

2 个箭头的功能不同,在设置函数的选项时需要用后者,例如 Plot[Sin[x], {x, 0, 6}, Axes → True]。如果你误输入前者(这种情况一般发生在代码是从别的文件中直接复制过来的),软件会报错。你会奇怪为什么报错,因为你完全发现不了错在哪。因此有时出现奇怪的报错,却怎么也找不出问题的时候,你可以将代码复制到 txt 文本编辑器中看看到底是不是你期望的样子。上面的两行输入在 txt 文本编辑器中的模样是

```
1 a \[RightArrow] b
2 a -> b
```

5. True 等价于 1 吗?

在 Matlab 中用 1 表示逻辑值 true,用 0 表示 false,但在 Mathematica 中 True 不等价于 1,如右图所示的例子:

可以看到,第一个 If 语句并没有对 a 赋值,因为 Mathematica 认为 1 不是条件表达式。

```
In[1]:= Clear[a];
If[1, a = 3, a = 4];
a
If[True, a = 3, a = 4];
a

Out[3]= a

Out[5]= 3
```

6. 行向量、列向量

在 Matlab 中是区分行向量和列向量的:

行向量	列向量
<pre>>> a=[1 2 3] a = 1 2 3</pre>	<pre>>> b=[1; 2; 3] b = 1 2 3</pre>

但在 Mathematica 中是不区分行向量和列向量的。向量就是一维的列表,例如: a = {1,2,3}。

矩阵与向量相乘使用符号 . 也就是英文句号,也用于两个向量做内积,而乘号*用于元素依次相乘,如下图:

```
In[1]:= a = {1, 2, 3};
a.a

Out[2]= 14
```

```
In[3]:= a = {1, 2, 3};
a * a

Out[4]= {1, 4, 9}
```


注意：别混淆向量与单行矩阵和单列矩阵。虽然行列向量不用区分，但是单行矩阵和单列矩阵要区分。我们对向量 $a = \{1, 2, 3\}$ 转置是非法的，软件会报错，但是对单行矩阵和单列矩阵是合法的，如下图所示：

<pre>In[1]:= a = {{1, 2, 3}}; Transpose[a] // MatrixForm Out[2]//MatrixForm= $\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$</pre>	<pre>In[2]:= a = {{1}, {2}, {3}}; Transpose[a] // MatrixForm Out[3]//MatrixForm= $(1 \ 2 \ 3)$</pre>
---	---

我们可以通过 Dimensions 函数判断一个列表是向量还是矩阵（向量是一维的，而矩阵是二维的）：

<pre>In[5]:= a = {1, 2, 3}; Dimensions[a] Out[6]= {3}</pre>	<pre>In[7]:= a = {{1, 2, 3}}; Dimensions[a] Out[8]= {1, 3}</pre>	<pre>In[1]:= a = {{1}, {2}, {3}}; Dimensions[a] Out[2]= {3, 1}</pre>
---	--	--

7. 矩阵操作

我们经常要对矩阵进行组合等操作，下面讨论几种组合方式并给出其实现代码：

◆ 添加行

例子	代码
$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$	<pre>a = {{1, 2}, {3, 4}}; b = {{5, 6}}; Join[a, b] // MatrixForm stackRows[a, b] // MatrixForm</pre>

注意：如果 a, b 是一行也要写成矩阵的形式，不能写成向量（例如 $b = \{5, 6\}$ 是错的）。

Join 要快 10 倍，如下例：

```
n=10000;
mat=Table[RandomInteger[{1,9},{2,3}],{i,n}];
math=Table[RandomInteger[{1,9},{1,3}],{i,n}];
a=Table[StackRows[mat[[i]],math[[i]]],{i,n}];//AbsoluteTiming
b=Table[Join[mat[[i]],math[[i]]],{i,n}];//AbsoluteTiming
a==b
```

◆ 添加列

例子	代码
$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$	<pre>a = {{1, 3}, {2, 4}}; b = {{5, 6}}; stackCols[a, b] // MatrixForm</pre>

注意：如果 a, b 是一列必须写成向量的形式，不能写成矩阵（例如 $b = \{\{5, 6\}\}$ 是错的）。

◆ 组成对角矩阵

例子	代码
$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \& \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 4 & 5 & 6 \end{bmatrix}$	<pre>a = {{1, 2}, {3, 4}}; b = {{1, 2, 3}, {4, 5, 6}}; diagF[a, b] // MatrixForm</pre>

注意：如果 a, b 是一行或一列必须写成矩阵的形式（例如单行矩阵 $b = \{\{1,2,3\}\}$ 或单列矩阵 $b = \{\{1\}, \{2\}, \{3\}\}$ ）。我们要区分单行矩阵和单列矩阵，因为组合得到的结果不同。

所需的代码见下表

<pre>(* Stack matrix columns together *) stackCols[mats_] := Block[{i,j}, Table[Join[Flatten[Table[{mats}][[j]][[i]], {j,Length[{mats}]}], 1], {i, Length[{mats}][[1]] }]];</pre>	<pre>(* Stack matrix rows together *) stackRows[mats_] := Join[Flatten[{mats}, 1]]; (* Construct diagonal matrix *) diagF=SparseArray[Band[{1,1}]- >{##}]]&;</pre>
---	--

8. Table、Do、For 有什么区别？

Table、Do、For 三者都可用于建立一个循环过程。但 Table 得到一个列表，其它二者不会返回任何值，例如可以这样给 a 赋值：

```
a=Table[i^2,{i,10}]
```

但是下面这样无法给 a 赋值：

```
a = Do[i^2, {i, 10}]
```

Table 中可以放任意多个中间过程语句，但是要用分号 ; 隔开，而且要将需要返回的值放在最后（最后一句不要加分号，如果加了就不会返回值），例如：

```
In[1]:= Table[j = 2 i; j^2, {i, 10}]
Out[1]= {4, 16, 36, 64, 100, 144, 196, 256, 324, 400}
```

Table 中用于控制循环次数的变量 i 是内部变量，不受外部定义的影响，例如：

```
In[1]:= i = 123;
Table[i^2, {i, 10}]
Out[2]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

9. 如何保存计算历史

在某些场合下（例如仿真），我们希望记录计算过程中某些变量的变化，用于后续的分析。我们有

多种选择。

- ① 初学者可以使用 `AppendTo` 函数，用法很简单：首先定义一个用于保存数据的空列表 `ahistory={}`;

随后将想保存的变量依次添加进此列表中即可

```
Do[a=i^2;AppendTo[ahistory,a],{i,10}]
```

- ② 使用 `Reap` 和 `Sow` 函数。`Sow` 的功能就是抛出数据，`Reap` 则负责接收抛出的数据，二者必须成对使用。相比 `AppendTo` 函数，`Reap-Sow` 函数的效率更高，如下例所示：

```
In[1]:= ahistory = {};
n = 10000;
{t1, null} = AbsoluteTiming@Do[a = i^2; AppendTo[ahistory, a], {i, n}];
{t2, ahistory2} = AbsoluteTiming@Reap[Do[a = i^2; Sow[a], {i, n}]] [[2, 1]];
t1
t2
ahistory == ahistory2

Out[5]= 0.475929

Out[6]= 0.0250215

Out[7]= True
```

`AppendTo` 版本用时 0.476s，而 `Reap-Sow` 版本只用时 0.025s，二者得到的结果完全相同。

如果我们要保存多个变量怎么办？`Reap-Sow` 可以做到，如下例：

数据
标签

```
In[1]:= Reap[Sow[x1, 1]; Sow[y1, 2]; Sow[x2, 1]; Sow[y2, 2];]
Out[1]= {Null, {{x1, x2}, {y1, y2}}}
```

保存的数据

`Sow[Data, Tag]`函数中第一个是被抛出的数据，第二个是数据的标签（Tag），软件会将有相同标签的数据自动放到一起。如果没有标签，那所有被 `Sow` 的数据将被放到一起。

- ③ 使用嵌套列表。

```
In[1]:= ahistory = {};
n = 10000;
{t1, null} = AbsoluteTiming@Do[a = i^2; ahistory = {ahistory, a}, {i, n}];
t1

Out[4]= 0.0222524
```

速度同样很快，缺点是需要对嵌套列表进行后续处理，例如展平 `Flatten`。

10. 如何自定义函数？

自定义函数的方法有很多种，下面分别讨论：

- ① 最简单且最容易理解的方式是（以乘方函数为例）：

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

其中各部分的含义是：

f	函数名
x₋	输入参数（注意参数后必须有以下划线）
:=	延迟赋值符号（别写成等号）
x^2	函数体（注意参数后没有下划线）

调用方法与 Mathematica 自带的函数一样，例如：f[5]

Mathematica 支持函数名重载，我们可以再定义一个有两个输入的同名函数（注意多个输入参数中间用逗号隔开）

```
f[x_, y_] := x^2 + y^2;
```

```
In[1]:= f[x_Integer] := x^2;
f[2]
f[2.5]

Out[2]= 4

Out[3]= f[2.5]
```

如果我们执行 f[3,4]，Mathematica 知道应该选择使用后者进行计算。

此外，我们还可以限制输入参数的类型，例如右图中输入参数下划线后面的 *Integer* 规定输入必须是整数时才调用函数计算，不满足要求的输入则不计算。

② 第二种方法更简洁，但初学者可能不容易理解。

```
f = (#^2) &;
```

小括号包含的部分 (*#^2*) 是函数体，用于实现该函数的功能，其中井号 *#* 代表输入参数，

末尾的 *&* 符号表示 f 是一个函数。调用方式与第一种定义相同，也是 f[2]

③ 第三种方法适用于比较复杂的函数

借助 Module 或 Block 等函数实现，我把定义时应该注意的地方在下图中标出来了：

函数定义用 :=

```
f[x_] :=
Module[{a = 3, b},
  b = a^2;
  x^2 + b
]
```

局部变量放在大括号内部

局部变量与函数体之间用逗号隔开

局部变量之间用逗号隔开

局部变量赋值

函数返回值结尾不用分号

整个函数定义由局部变量部分和函数体部分组成，它们之间要用逗号隔开。函数体内部的语句用分号隔开，只有最后需要返回值的语句不需要加分号。

注意：在所有函数定义中，输入参数只能引用，不能修改。比如输入值 x，后面不能再对其赋值了。函数调用时，传递参数类型应符合它的定义，比如定义时是 f[x_, y_]，在调用时就应该是 f[3, 4]，不能写成 f[{3, 4}]。

11. @、@@、/@、#、&都是什么意思？

使用 Mathematica，最让初学者抓狂的是面对一堆奇怪的符号：@ /@ ## & /. ~ //，让人如同读天

书。由于 Mathematica 具有函数式编程风格，所有的功能都作为函数对待，而上述这些符号就是表示函数作用于数据的不同方式。

@ —— 我们在函数前置中提到过@，它的功能就是将它后面的数据直接“喂给”函数“吃”：

```
In[1]:= Sqrt@2.0
Out[1]= 1.41421
```

@@ —— 函数可能有不止一个输入参数（有好几张嘴），@@的功能是将数据中的元素一次性地挨个放入函数的每个“嘴”：

```
In[1]:= f@@{1, 2, 3}
Out[1]= f[1, 2, 3]
```

/@ —— 前面我们都只调用了一次函数，有时我们希望将同一函数用到很多不同的数据上，/@的功能就是将很多“数据”挨个“喂给”函数：

```
In[1]:= Sqrt /@ {1, 4, 9, 16, 25}
Out[1]= {1, 2, 3, 4, 5}
```

Mathematica 中有些函数有 Listable 性质，意思就是函数既支持输入是单个数据元素也支持输入是列表的形式。例如 Sqrt 函数就有这个性质，所以我们可以直接给它一个列表，如下图所示。注意输出的列表保留了输入列表的“层次”。

```
In[1]:= Sqrt@{{1}, 4, 9, 16, {25, {36}}}}
Out[1]= {{1}, 2, 3, 4, {5, {6}}}
```

强大的 Mathematica 还支持函数的嵌套，所以上述用法可以叠加起来。试着理解下面语句的结果：

```
In[1]:= f@@# & /@ {{1, 2}, {3, 4}}
Out[1]= {f[1, 2], f[3, 4]}
```

解释：在函数定义时我们知道，&符号表示一个函数，因此我们可以将 f@@#&整体视为一个函数，记为 f1，作用的结果是：

```
In[1]:= f1 /@ {{1, 2}, {3, 4}}
Out[1]= {f1[{1, 2}], f1[{3, 4}]}
```

然后再将 f1 替换回 f@@#&

```
In[2]:= {f@@# &[{1, 2}], f@@# &[{3, 4}]}
```

也就等价于 f@@

```
In[5]:= {f@@{1, 2}, f@@{3, 4}}
Out[5]= {f[1, 2], f[3, 4]}
```

初学者容易混淆&和@后面分别作用什么括号，&后面应该跟中括号[]，而@后应跟大括号{}。

12. 自定义 Package 程序包

Mathematica 中的函数都按照不同的功能组织成一个个的程序包，我们也可以定义自己的程序包 (Package)。自定义程序包需要四部分：

```
BeginPackage["myPackage`"];
myFunction1::usage="这里是注释，一般介绍程序包中的各函数都有什么功能"
myFunction2::usage=""
Begin["`Private`"];
myFunction1 [input_]:=
  Module[{},
    具体代码
  ];
myFunction2 [input_]:= ... ;
End[];
EndPackage[];
```

① BeginPackage["myPackage`"];

myPackage 是自定义程序包的名字，也就是程序包文件保存的名字 myPackage.m (二者要一致，注意后缀名是 m 而不是 Notebook 的后缀名 nb)

如果你的程序包调用了其它程序包中的函数，需要在加上该程序包的名字。例如依赖 otherPackage.m 包就改为：BeginPackage["myPackage`",{otherPackage`}];

② myFunction1::usage="这里是注释，一般介绍程序包中的各函数都有什么功能"

后面定义的所有函数声明和注释。双引号里面是函数的说明，你可以随便写，也可以空着，但不能不写。

③ Begin["`Private`"];

双引号里的`Private`的意思是，程序包中的变量都是局部变量，其值只能在此包内部使用，变量值的有效范围在 Mathematica 中称为上下文 (Context)。默认的上下文是全局的，即`Global`，如果在定义程序包的时候，不写 Begin["`Private`"];和 End[];，那么包里的数据是全局的，不管哪个文件都可以访问调用。

④ End[];

EndPackage[];

结束程序包。

调用程序包的方法：先指定程序包存放的目录，然后用<<运算符调用即可，例如：

```
SetDirectory["C:\\packagePath\\"];
```

```
<<myPackage.m;
```

如果程序包与你的主程序在同一文件夹下可以使用 `SetDirectory[NotebookDirectory[]]`;指定程序包存放的目录与主程序地址相同。我们有时会经常移动程序文件夹，这时需要经常改变目录。我的经验是在主程序文件夹下新建一个 `packages` 文件夹，将所有的程序包放在 `packages` 文件夹下。然后主程序中可以用以下命令调用程序包：

```
SetDirectory[NotebookDirectory[] <> "packages"];
```

```
<< myPackage.m;
```

```
SetDirectory[NotebookDirectory[]];
```

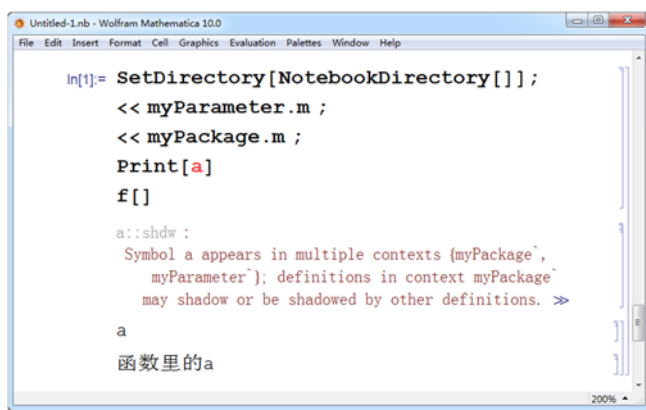
这样再移动文件时，我们只需要移动主程序所在的整个文件夹即可，而不需要修改程序包地址了。

13. 上下文 (Context)

其作用与 C++语言中的 `namespace` 一样，用来避免命名冲突。存在命名冲突时软件会将变量名置为红色。一个有可能引起冲突的情况如下例：我们定义了两个程序包：`myPackage.m` 和 `myParameter.m`，前者用于保存自定义函数，后者只用于保存参数：

myPackage.m 程序包	myParameter.m 程序包
<pre>BeginPackage["myPackage`"]; f::usage = "" f[]:= Module[{a="函数里的a"}, Print[a];]; EndPackage[];</pre>	<pre>BeginPackage["myParameter`"]; a=2; EndPackage[];</pre>

`myPackage.m` 中定义了一个函数 `f`，它包含一个局部变量 `a`，其值是一个字符串。`myParameter.m` 中定义了一个变量 `a`，其值被赋为 2。当我们调用这两个包时发现变量 `a` 显示为红色，说明变量 `a` 的定义存在冲突。当我们想打印出 `a`，发现它在 `f` 中被赋值了，说明 `myParameter.m` 中的赋值被遮盖了。



解决方法是：在 `myPackage.m` 的定义中指定变量的作用范围，即加入语句：`Begin["Private"];`和 `End[];`；这样 `myPackage.m` 中的变量 `a` 仅在 `myPackage` 包中有效。

14. 带上下标的变量

Mathematica 的编程界面 (notebook) 支持丰富的符号显示效果，例如带上下标。我们应该如何输入呢？

上标: Ctrl+6

下标: Ctrl+- (减号)

但是这样只能输入单独的上标或下标, 如果变量既有上标也有下标需要通过“面板”——“数学助手”——“排版”中的方法了。但是在 Mathematica 使用上下标是很麻烦的, 尽量避免使用带上下标的变量。因为你不能用 Clear 命令清除它的值, 如下左图所示。你也无法改变带上下标的变量中元素的值, 例如下右图所示。

```
In[1]:= x1 = 2;
Clear["Global`*"];
x1^2
Out[3]= 4
```

```
In[3]:= ma = {{1, 2, 3}, {4, 5, 6}};
ma[[2, 3]] = 0;
ma
Out[5]= {{1, 2, 3}, {4, 5, 6}}
```

如果你实在想给成员赋值, 你可以借助 Symbolize 包。此时, 所有带标符号成为一个整体。什么意思? 在声明之前, 我们查看变量的类型, 如右图所示:

使用 Symbolize 包之后, 变量的类型改变了: 也就是说, 声明之前 x_1 在 Mathematica 中是 Subscript[x, 1], 声明之后变成 x_Subscript_1, 这样的符号和普通的不带标符号一样。

```
In[1]:= FullForm[x1]
Head[x1]
Out[1]/FullForm= Subscript[x, 1]
Out[2]= Subscript
```

```
<< Notation`
Symbolize[x1];
OutputForm[x1]
Head[x1]
Out[3]/OutputForm= x_Subscript_1
Out[4]= Symbol
```

15. 导入导出数据

Mathematica 支持丰富的数据格式, 常用的格式有 txt、csv、xlsx、dat 等。

根据导入数据的存储格式应使用不同的函数。

txt 格式: a = ReadList["C:\\data.txt"]; 注意地址用双斜杠。如果文件较大 (>5MB) 这种方法较慢, 我们需要一种更高效的方法 (假设导入的数据是数值):

```
readstream = OpenRead["C:\\data.txt"];
a = ReadList[readstream, Number];
```

csv 或 xlsx 格式: a = Import["C:\\data.csv"];

stl 格式 (3 维图形): a = Import["C:\\data.stl", "PolygonObjects"]; 注意如果属性一项 (即 "PolygonObjects") 没有则默认导入的是 Graphics3D 图形类型。

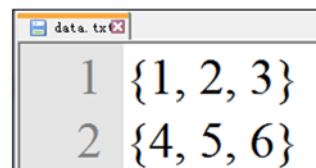
说明: 导出数值数据最好采用 csv 格式, 尽量不用 xlsx 格式, 因为存储同样的内容, xlsx 格式比 csv 占用硬盘空间更大, 处理更慢。C++ 支持导出数据格式为 csv 和 txt, 代码如下:

```
FILE *fp;
fp = fopen("C:\\data.csv", "w");
fprintf(fp, "%f\n", a); // a 是要导出的数据变量名
fclose(fp);
```

从 Mathematica 中导出数据的方式是:

```
a = {{1,2,3},{4,5,6}};
```

```
Export["C:\\data.txt", a]; 导出 csv 格式的同理
```



```
1 {1, 2, 3}
2 {4, 5, 6}
```

注意 导出的数据包含大括号，见右图：

导出数据的精度对文件大小有影响，例如：

代码	<code>a = Table[N[Pi, 10], {i, 1, 10^4}]; Export[C:\\data.csv, a];</code>	<code>a = Table[N[Pi, 2], {i, 1, 10^4}]; Export[C:\\data.csv, a];</code>
文件大小	126KB	48.8KB

Mathematica 自定义的 mx 数据格式支持导出任意类型的数据（包括图像），例如：

```
a = Graphics3D[{Blue,Cylinder[],Red,
Sphere[{0,0,2}],Black,Thick,Dashed,Line[{{-2,0,
2},{2,0,2},{0,0,4},{-2,0,2}}],Yellow,Polygon[{{-
3,-3,-2},{-3,3,-2},{3,3,-2},{3,-3,-2}}],Green,Opac
ity[.3],Cuboid[{-2,-2,-2},{2,2,-1}]}];
Export["C:\\data.mx",a];
Import["C:\\data.mx"]
```

