Julia语言入门

作者：李东风

2020-04-05

目录

Julia程序语言介绍

Julia程序语言是一种计算机编程语言， 就像C、C++、Fortran、Java、R、Python、Matlab等程序语言一样。 Julia语言历史比较短，发布于2012年， 是MIT的几位作者（Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman）和全世界的参与者共同制作的。 主网站在https://julialang.org/ (https://julialang.org/)。

Julia与R、Python等一样是动态类型语言， 程序与R、Python一样简单， 但是它先进的设计使得Julia程序的效率基本达到和C、Fortran等强类型语言同样的高效率。 尤其适用于数值计算，在现今的大数据应用中也是特别合适的语言， 排在Python、R之后，已经获得广泛的关注， 现在用户较少只是因为历史还太短。

本文介绍Julia语言的软件安装设置， 基本的计算编程。 目标读者是做数值计算、统计分析等工作的实际用户而不是开发人员， 作者假定读者具有一定的编程语言基础， 比如，学过R、Python、Matlab、C、Java等编程语言。 本文作者后续还计划编写关于统计数据整理、作图、统计分析的入门教材。

注意， Julia使用即时编译技术， 用LLVM虚拟机执行， 这使得其程序在初次运行时好像与R、Python这些动态语言相比响应很慢， 这其实是在进行即时编译， 用编译的等待换取运行时的高效率， 对于不能向量化程序中Julia可以比R、Python这些动态语言快一个数量级。

Julia不同版本的程序用法有细微差别， 本文例子基于Julia v1.4。

Julia软件安装

Julia 现在的版本是1.4.0版， 虽然已经完全可以用来执行生产级的任务， 但是毕竟历史还太短， 软件的安装设置还比较复杂， 兼容性不够， 安装需要从国外的服务器下载许多文件来安装也使得安装设置容易出错， 需要反复尝试。

命令行程序的安装和使用

只要从Julia主网站https://julialang.org/ (https://julialang.org/)下载安装， 这是最容易的部分。 在安装时可以安装在默认位置，这一般是用户主目录， 在Windows下是“C:\Users\自己的用户名”， 也可以自己指定一个目录。

Julia命令行的运行方式是在一个字符型窗口中， 在提示行 `julia>` 后面键入命令， 回车后在下面显示结果。

在MS Windows操作系统中， 设存放Julia源程序和数据文件的目录为 `C:\work` , 将安装后显示在桌面上的Julia图标复制到目录中， 从右键选“属性”，将“起始位置”栏改为空白。 这样读写数据和程序文件的默认位置就是图标所在的目录。

在Julia命令行， 为了显示当前工作路径， 用命令

```
julia> pwd()
```

(其中 `julia>` 是自动显示的提示)。 为了将当前工作目录设置到 `C:\work` ， 命令如

```
julia> cd("C:/work")
```

注意路径分隔符用 `/` 而不是 `\` 。

Julia源程序用 `.jl` 作为扩展名， 为了运行当前目录中的“myprog.jl”文件， 在命令行用命令

```
julia> include("myprog.jl")
```

在命令行状态下，按Ctrl+D键可以退出REPL界面。 可以用上下光标键调回以前输入过的命令， 重新运行或者修改后再运行。

仅使用命令行还不够方便， 推荐使用如下两种功能更强的集成环境：

- Atom + Juno集成编辑环境；
- Jupyter笔记本环境。

关于JuliaPro套装

JuliaPro是Julia Computing公司（http://juliacomputing.com (http://juliacomputing.com)）的Julia集成环境， 从其网站下载JuliaPro的免费个人版， 其中包含了命令行程序， 以及基于Atom+Juno编辑器的集成编辑和运行环境。 目前最新版本是1.4.0版，对应Julia 1.4.0。

安装后， 运行JuliaPro， 可以进入一个编辑、运行Julia程序的集成环境(IDE)。

我安装了JuliaPro1.4.0版本后， 发现它更新包需要登录， 可以用github账户， 但是因为优先从公司网站下载， 经常遇到网络拥堵。 而且与单独安装的Anaconda3中的Python和Jupyter软件的兼容性也不好， 就卸载了JuliaPro。 个人不推荐使用。

安装设置Atom+Juno集成编辑环境

安装设置

Atom是一个程序编辑器， 功能强大， 配合其Juno包可以作为Julia的一个强大的集成编辑环境(IDE)。

从Atom网站https://atom.io/ (https://atom.io/)下载安装Atom软件。 安装后运行Atom， 选择“File--Settings”，找到“+install”选项， 在搜索框中输入“uber-juno”， 选择安装此包。 安装好以后， 再次进入Atom， 会自动下载安装与Julia相联系所需要的扩展包， 包括Atom的julia-client和julia-language包， Julia的juno包和Atom包。

因为需要从网上下载安装额外的包， 如果下载安装出错， Julia不能在Atom中运行， 可以在自己的主目录如“C:\用户\自己的用户名”中找到 `.atom\packages` ，删除其中的uber-juno，julia-client, julia-language， 然后重新在Atom中安装其uber-juno包。 或干脆卸载Atom后重新安装Atom和Atom的uber-juno包。

安装好Atom和uber-juno后， 应该在Atom的设置中找到julia-client包， 将其中的Julia可执行程序指向新安装的Julia可执行程序julia.exe的位置， 如 `D:\Julia\Julia1.4.0\bin\julia.exe` 。 也可以将该路径加入系统的 `PATH` 环境变量， 这样就不需要在Atom中设置Julia可执行程序的位置。

用法简介

在Atom+Juno中， 可以打开一个文件夹作为项目目录， 并选择菜单“Juno -- Working Directory -- Select Project Folder”。

编辑“.jl”源文件， 在源文件中用Ctrl+Enter可以运行当前行或当前结构或选定行。 可以用Shift+Enter运行当前行或当前结构或选定行， 并将光标移到当前结构后面。运行结果会以折叠方式显示在当前结构的右方。

用菜单“Juno -- Run All”运行整个源文件。 注意这样运行时表达式的值不自动显示。

Atom+Juno中有一个Console窗格，即内嵌的Julia命令行。 如果没有这个窗格，可以用“Juno -- Open REPL”菜单建立。 在Console窗格按ENTER键可以在此窗格中启动Julia命令行。 可以用“Juno -- Stop Julia”退出这里的命令行， 然后再按ENTER键可以重启Julia命令行。

程序运行的图形结果可以显示在Atom内的一个窗格中。 Plots包可以将结果显示到Atom的图形窗格内， 建议使用GR后端或者PyPlot后端。

Atom+Juno还有一个Workspace窗格， 可以显示当前定义的变量、函数的类型和内容。 如果没有这个窗格可以用“Juno -- Open Workspace”菜单建立。

Atom+Juno支持程序调试， 可以设置断点， 跟踪运行， 跟踪运行时自动显示当前的变量列表和变量值。

详见：

- http://docs.junolab.org/latest/ (http://docs.junolab.org/latest/)

Jupyter笔记本软件的安装和使用

Jupyter笔记本是一种将程序代码、说明文字、执行结果结合在一起的文档格式， 这使得用户很容易地编写带有程序结果的研究报告。 这种功能的实现由Python编写的网络后台应用程序（jupyter.exe程序）、Julia(或Python)后台执行程序（称为核心，kernel）和浏览器前端显示组成。 保存的内容是扩展名为.ipynb的文本格式的文件， 所以很容易地交换， 也支持版本控制。 核心可以是Python、Julia、R等语言。

JupyterLab是Jupyter的一个升级版本， 和Jupyter共用一个服务器后端。

为了在MS Windows环境下安装Jupyter软件， 有两种方法： 安装Anaconda 3， 这里仅介绍使用Anaconda 3的Jupyter的方法， 这种方法会有较强的功能， 带有好用的Python环境和科学计算的软件包， 但是与Julia的兼容性略差， 需要花一些时间去调整。 从Julia安装Jupyter的办法会安装一个Julia私有的Python环境， 与Julia的兼容性好但是通用性不够。

Anaconda3是Python的开发环境和软件管理环境， 可以为不同的软件组合与版本设置不同的“环境”， 使得不同版本的软件可以无冲突地共存于同一台电脑。

在安装Julia软件之前先安装Anaconda3， 从Anaconda3的程序组中启动Jupyter笔记本软件， 就已经可以支持Python语言的笔记本。

为了在Julia中安装对已有的Anaconda3中的Jupyter软件的支持， 在anaconda命令行用如下where命令找到python和jupyter软件的安装位置：

```
>where jupyter
D:\Anaconda3\Scripts\jupyter.exe
>where python
D:\Anaconda3\python.exe
C:\Users\user\AppData\Local\Microsoft\WindowsApps\python.exe
```

在Julia中，先设置Python和Jupyter的位置，然后安装和构建IJulia：

```
ENV["PYTHON"] = "D:\\Anaconda3\\python.exe"
ENV["JUPYTER"] = "D:\\Anaconda3\\Scripts\\jupyter.exe"
Pkg.add("IJulia")
```

如果中间出错，可以运行

```
Pkg.build("IJulia")`。
```

安装好IJulia以后， 可以从Anaconda3的Navigator程序中启动Jupyter或者JupyterLab。 也可以从Windows程序开始菜单启动Jupyter， Jupyter会自动显示在系统模式网络浏览器中（最好是Chrome浏览器）。

如果将浏览器地址栏中的 `.../tree` 改成 `.../lab` ，还可以进入JupyterLab。

JupyterLab是Jupyter的改进版本， 功能更强，与Jupyter使用同一个服务器后端， 从Anaconda3命令行启动的命令为：

```
jupyter lab
```

启动Jupyter后，Julia引擎在Jupyter中已经可以用了， 用“File -- New Notebook -- Julia 1.4.0”可以新建一个支持Julia程序在其中运行的笔记本。

Anaconda中的Jupyter笔记本是默认以用户个人主目录为工作目录的， 为了将工作目录设置为其它目录如“d:\work”， 可以右键单击Jupyter笔记本快捷图标， 将“目标”部分的的“jupyter-notebook-script.py”和“%USERPROFILE%”之间插入要用的目录如“d:\work”， 并将“起始位置”也改成这个目录。 这样再从快捷图标启动， 看到的目录就是自己指定的目录了。

Jupyter笔记本用法

在笔记本中可以运行程序， 并将程序的文字、表格、图形结果以富文本形式直接和其它说明文字、程序代码一起显示在浏览器窗口， 并可以转换为HTML、LaTeX、PDF等格式。

打开Jupyter程序后会自动调用一个浏览器窗口， 其中显示了用户目录。用“新建”功能可以生成Julia的笔记本， 也可以生成Python笔记本。 实际上，Jupyter程序原来主要是针对Python笔记本。 从文件资源管理器将笔记本文件拖入这个窗口可以将笔记本纳入Jupyter的管理。

新建或编辑旧有笔记本时，在浏览器的网页内会显示菜单和快捷图标栏。

笔记本的内容由“单元”组成。 笔记本单元分为Markdown和程序两种单元。 Markdown单元是文档的说明部分，采用markdown语法。 可以输入类似LaTeX格式的数学公式， 其显示由MathJax程序库支持。 程序是Julia程序。

单元工作状态分为“编辑”和“命令”两种状态。 编辑状态下单元左边框为绿色， 命令状态下单元左边框为蓝色。 在命令状态下除了菜单和快捷图标以外还可以用许多快捷键。

对于程序单元， 用鼠标单击左边编号部分可以选中单元并进入命令状态， 点击程序输入框可以选中该单元并进入编辑状态。 单元输入完成后， 用Shift+Enter将程序单元执行并在单元下方显示程序的结果。

Markdown单元可以以原始文字格式或者富文本格式存在。 从原始文本格式用Shift+Enter将Markdown单元显示为富文本格式。 如果Markdown单元显示为原始文字格式， 单击其输入框就可以选中该单元并进入编辑状态， 单击其输入框左边的空白可以选中该单元并进入命令状态； 如果该单元已经显示为富文本状态， 需要双击才能选中该单元并进入编辑状态， 单击可以选中该单元并进入命令状态。

在命令状态下按回车键可以进入编辑状态。 在编辑状态按Esc键可以进入命令状态。

缺省的单元类型是程序单元。 为了将程序单元修改为Markdown单元， 在命令状态下按m键可以将其切换为Markdown单元。 在编辑状态下先按Esc再按m键即可切换为Markdown单元。 从命令进入编辑状态只要按Enter键。

在命令状态下， 用b键在当前单元下方插入一个新单元， 用a键在当前单元上方插入一个新单元。 选中一个单元后按Shift+m可以将其与下一个单元合并。

Markdown标题如果单独占一个单元， 可以称这样的单元为标题单元， 在命令状态下用快捷键1,2,3,4,5,6可以将单元转换为一级到六级标题的单元。

Markdown单元支持LaTeX格式的数学公式， 如行内公式$e^x = \sum_{j=0}^{\infty} \frac{1}{j!} x^j$, 独立公式

$$\begin{aligned} x &= 10 \times (2 + 3) \\ &= 10 \times 5 = 50 \end{aligned}$$

笔记本的“Cell--Run All”菜单可以将整个笔记本的所有程序都依次运行， 并将Markdown单元都变成富文本。

在Julia程序版本升级以后， 基于老版本的笔记本可以用“Kernel -- Change kernel”菜单升级到使用新的Julia引擎。

扩展包

Julia的基本语言比较精炼， 许多功能都需要依赖扩展包(packages)完成， 一些基本的语言功能也放到了Base扩展包中， Base扩展包中的函数都是直接可以使用的， 不需要单独安装和声明。

其它的一些功能则需要从网上安装， 并且使用时需要声明。

为了调用某个扩展包的功能， 一般只要用 `using` 关键字将其提供的函数和全局变量调入到当前名字空间即可。 如

```
using DataFrames
```

Julia语言的扩展包在需要时从网上安装， 公布的扩展包都放在Github网站， 列表见：

- https://pkg.julialang.org/ (https://pkg.julialang.org/)

国内现在连接Github网站不太顺畅， 使得安装扩展包经常会因为网络阻塞而失败。 可以在清晨网络比较畅通时下载安装。

在Julia命令行安装扩展包，如DataFrames，命令如

```
using Pkg
Pkg.add("DataFrames")
```

如果要安装的扩展包依赖于其它的扩展包， 这些扩展包也会自动被安装。

安装的扩展包如果不能正常工作， 可以尝试重新构建，如：

```
using Pkg
Pkg.build("DataFrames")
```

为了更新已安装的所有包到最新版本， 用命令

```
Pkg.update()
```

安装经验

上面已经给出了安装Atom+Juno， Anaconda3，Jupyter笔记本功能的经验。 Julia现在不够成熟， 兼容性不够好， 软件包管理也还经常出错， 所以需要有出错之后查找各种解决方法的耐心。

除了从搜索引擎查找问题解决方法以外， 各个包的文档也给出了常见的问题和各种定制安装方法。 Julia的扩展包文档的网站在：

- https://pkg.julialang.org/docs (https://pkg.julialang.org/docs)

Conda包、PyCall包、ORCA包设置

Julia的某些包， 尤其是与Python有关的包， 需要安装一个Conda包， 此包默认会安装miniconda可执行程序并在用户主目录的.julia子目录下面安装私有的Python环境， Julia私有的Python环境如果靠Julia管理有时有兼容性问题。

这里介绍如何使用已有的Anaconda3环境的方法。 Anaconda是一个Python系统的复杂包依赖的安装管理系统， 也可以用来管理其它的复杂软件包， 特点是可以同时安装不同版本的软件而不会冲突。

为了利用已有的Anaconda3， 在Anaconda中运行如下命令：

```
conda create -n conda_jl python=3.7 conda
conda activate conda_jl
where python
## D:\Anaconda3\envs\conda_jl\python.exe
conda install matplotlib
```

其中 ## 后面是显示结果。 这会下载安装单独的一份必要的python环境和conda程序， 现在的最新的Python是3.8.x版但是Julia的PyPlot作图包需要的Python matplotlib包还不支持， 所以定制安装3.7.x版的Python。 这样安装的环境在Anaconda软件主目录下的envs目录中， 如果让Julia自动安装， 会安装在用户主目录的.julia目录下。

然后， 在Julia中指定私有Conda环境的路径、其中的Python程序的路径， 安装Conda包、PyCall包和PyPlot包（路径需要替换成自己的路径）：

```
ENV["CONDA_JL_HOME"] = "D:\\Anaconda3\\envs\\conda_jl"
ENV["PYTHON"] = "D:\\Anaconda3\\envs\\conda_jl\\python.exe"
using Pkg
Pkg.add("Conda")
Pkg.add("PyPlot")
Pkg.build("PyPlot")
```

在Julia命令行测试PyPlot:

```
using PyPlot
x = [1,3,5,8];
y = x .^ 2;
plot(x, y, color="blue")
```

在Julia命令行测试成功。 还可以在Anaconda3的Jupyter中测试。

在Atom+Juno中测试， 作为Plots后端：

```
using Plots
Plots.pyplot()
x = [1,3,5,8];
y = x .^ 2;
Plots.plot(x, y, color="blue")
```

结果可以显示在Atom的图形窗格中。

但是，在Atom+Juno中测试PyPlot， 图形不能显示在Atom的图形窗格中， 需要用 `pygui(true)` 打开Python自己的一个独立图形窗口：

```
using PyPlot
pygui(true)
x = [1,3,5,8];
y = x .^ 2;
plot(x, y, color="blue")
```

使用独立的Python的图形窗口也有一些好处， 内部的图形是可交互修改的， 比如放大某一矩形区域。

如何使PyPlot图形结果显示到Atom+Juno窗格内的问题待解决。

Julia的基本数据和相应运算

整数与浮点数

Julia程序中的整数值可以直接写成如 `123` 或者 `-123` 这样。 虽然整数有多种类型， 一般程序中不必特别关心整数常量的具体类型。 Julia允许使用特别长的整数，这时其类型为BigInt。

Julia的浮点数可以写成带点的形式如 `123.0` ， `1.23` , 也可以写成带有10的幂次如 `1.23e3` (表示1.23×10^3)， `1.23e-3` (表示1.23×10^{-3})。 这些写法都属于Float64类型的浮点数。 Julia还有其他类型的浮点数，但是科学计算中主要使用Float64类型， 在别的语言中这称为双精度浮点数。

Julia还提供了任意精度整数与任意精度浮点数。

布尔类型Bool只有两个值：true和false。

四则运算

表示加、减、乘、除、乘方的运算符分别为：

```
+  -   *  /  ^
```

浮点数的四则运算遵循传统的算数运算规则和优先级规定。 用圆括号改变优先级。如

```
In [1]: (1.3 + 2.5)*2.0 - 3.6/1.2 + 1.2^2
Out[1]: 6.039999999999999
```

表示

$$(1.3 + 2.5) \times 2.0 - 3.6 \div 1.2 + 1.2^2$$

注意浮点运算引起会造成数值计算误差。

整数的四则运算

整数加法、减法、乘法结果仍为整数， 这样因为整数的表示范围有限，有可能发生溢出。 如

```
In [2]: 10 + 2*3 - 3*4
```

```
Out[2]: 4
```

整数用“/”作的除法总是返回浮点数，即使结果是整数也是一样：

```
In [3]: 10/2
```

```
Out[3]: 5.0
```

求整数除法的商，用 ÷ 运算符, 如

```
In [4]: 5 ÷ 3
```

```
Out[4]: 1
```

其中 ÷ 的输入方法是在命令行中输入 `\div` 后按TAB键。 这种方法可以输入许多数学符号， 如 α (alpha)， π (pi)， $\sum$ (sum)，等等。

整数用 `a % b` 表示a整除b的余数，结果符号总是取a的符号。如

```
In [5]: 10 % 3
```

```
Out[5]: 1
```

整数与浮点数的混合运算会将整数转换成浮点数再计算。

数学函数

和其它科学计算语言类似， Julia也支持常见的数学函数， 如 `log` ， `exp` ， `sqrt` ， `sin` ， `cos` ， `tan` 等。

`round(x)` 将 `x` 四舍五入为整数， `round(x, digits=2)` 将 `x` 四舍五入到两位小数。 `floor(x)` 求小于等于 `x` 的最大整数， `ceil(x)` 求大于等于 `x` 的最小整数。

字符串

单个字符在两边用单撇号界定，如 `'A'` ， `'囧'` 。 字符都是用Unicode编码存储，具体使用UTF-8编码。 每个字符可能使用1到4个字节表示。 字符的类型为Char， 自定义函数中的字符参数可声明为AbstractChar。

零到多个字符组成字符串， 程序中的字符串在两边用双撇号界定，如 `"A cat"` ， `"泰囧"` 。 字符串数据类型名称为String， 自定义函数中的字符串参数可声明为AbstractString。

对于占据多行的字符串， 可以在两侧分别用三个双撇号界定。如

In [6]:
```
"""
这是第一行
这是第二行
三个双撇号界定的字符串中间的单个双撇号"不需要转义
"""
```

Out[6]: `"这是第一行\n这是第二行\n三个双撇号界定的字符串中间的单个双撇号\"不需要转义\n"`

注意多行内容的首行不需要紧挨着写在开头的三个双撇号后面同一行内。

字符串属于不可修改类型(immutable)， 即不能直接修改字符串的内容， 但可以给保存了字符串的变量赋值为一个新的字符串。

用星号“ `*` ”连接两个字符串，也可以将字符连接成字符串，如

In [7]:
```
'#' * "这是" * "美好的一天" * "。"
```

Out[7]: `"#这是美好的一天。"`

变量

变量名是一个标识符， 用来指向某个值在计算机内存中的存储位置。 变量名可以用英文大小写字母、下划线、数字、允许的Unicode字符， 区分大小写。 变量名不允许使用空格、句点以及其它标点符号和井号之类的特殊字符。

为兼容性起见，尽可能不要用汉字作为变量名。 变量名主要使用小写字母、数字和下划线构成， 两个英文单词之间可以用下划线连接， 如 `total_number` ， 也可以在单词之间使用开头字母大写来区分单词， 如 `totalNumber` ， 第一个单词仍全部用小写。

给变量**赋值**，即将变量名与一个内存中的内容联系起来，也称为绑定（binding）， 使用等号“=”，等号左边写变量名，右边写要保存到变量中的值。如

```
In [8]: x = 123
```

```
Out[8]: 123
```

```
In [9]: y = 1+3/2
```

```
Out[9]: 2.5
```

```
In [10]: addr10086 = "北京市海淀区颐和园路5号"
```

```
Out[10]: "北京市海淀区颐和园路5号"
```

变量的类型是由它保存的（指向的内存中的）值的类型决定的， 不需要说明变量类型（Julia允许说明变量类型，但一般不需要）。

变量赋值后，就可以参与运算，如：

```
In [11]: x = 123
         y = 1+3/2
         x + y*2
```

```
Out[11]: 128.0
```

变量名前面紧挨着数字表示相乘，如

```
In [12]: x + 2y
```

```
Out[12]: 128.0
```

赋值还有一种计算修改简写方式， 即将变量值进行四则运算后保存回原变量， 格式为 `x op= expr` ， 其中 `op` 是某种四则运算， 这等价于 `x = x op expr` ，如：

```
In [13]: x = 123
         x += 100
         x
```

```
Out[13]: 223
```

比较和逻辑运算

比较运算

比较运算

两个数值之间用如下的比较运算符进行比较：

```
==    !=    <    <=    >    >=
```

分别表示等于、不等于、小于、小于等于、大于、大于等于。 要特别注意“等于”比较用两个等号表示。

比较的结果是true(真值)或者false(假值)。 结果类型为**布尔型**（Bool）。

如

```
In [14]: 1 == 1.0
```

```
Out[14]: true
```

```
In [15]: 2 != 2.0001
```

```
Out[15]: true
```

```
In [16]: 3.5 > -1
```

```
Out[16]: true
```

```
In [17]: 3.5 < -1.5
```

```
Out[17]: false
```

```
In [18]: -3.5 >= 1.2
```

```
Out[18]: false
```

```
In [19]: -3.5 <= 1.2
```

```
Out[19]: true
```

两个字符串之间也可以比较， 比较时按字典序比较， 两个字符的次序按照其Unicode编码值比较。如

```
In [20]: "abc" == "ABC"
```

```
Out[20]: false
```

```
In [21]: "ab" < "abc"
```

```
Out[21]: true
```

```
In [22]: "陕西省" == "山西省"
```

```
Out[22]: false
```

逻辑运算

比较通常有变量参与。如

```
In [23]: age = 35; sex="F"
         age < 18
```

```
Out[23]: false
```

```
In [24]: sex == "F"
```

```
Out[24]: true
```

有时需要构造复合的条件， 如“年龄不足18岁且性别为女”， “年龄在18岁以上或者性别为男”等。

用 `&&` 表示要求两个条件同时成立， 用 `||` 表示只要两个条件之一成立则结果为真， 用 `!cond` 表示 `cond` 的反面。 如

```
In [25]: age < 18 && sex == "F"
```

```
Out[25]: false
```

```
In [26]: age >= 18 || sex == "M"
```

```
Out[26]: true
```

简单的输出

在Julia命令行，键入变量名或者计算表达式直接在下面显示结果。 可以用 `println()` 函数显示指定的变量和结果。如

```
In [27]: println(x + y*2)
```

```
228.0
```

在 `println()` 中多个要输出的项用逗号分开。 两项之间没有默认的分隔， 如果需要分隔可以自己写在输出项中。

```
In [28]: println("x=", x, " y=", y, " x + y*2 =", x+y*2)
```

```
x=223 y=2.5 x + y*2 =228.0
```

`println()` 函数输出会将后续输出设置到下一行， 而 `print()` 函数与 `println()` 类似但是将后续输出设置在当前行。

在命令行运行时， 表达式的值自动显示。 在表达式末尾用分号结尾表示不要显示该表达式的结果。

在用 `include()` 命令执行整个脚本文件时， 每个表达式的结果默认不自动显示。

字符串中可以用 `$变量名` 或 `$(表达式)` 的格式插入变量或表达式的值。例如

```
In [29]: name="John"; "My name is $name"
```

```
Out[29]: "My name is John"
```

```
In [30]: "100$(name)999"
```

```
Out[30]: "100John999"
```

向量

Julia的向量实际是一维数组。 在程序中直接定义一个向量， 只要用方括号内写多个逗号分隔的数值，如

```
In [31]: v1 = [1, 3, 4, 9, 13]
```

```
Out[31]: 5-element Array{Int64,1}:
           1
           3
           4
           9
          13
```

```
In [32]: v2 = [1.5, 3, 4, 9.12]
```

```
Out[32]: 4-element Array{Float64,1}:
          1.5
          3.0
          4.0
          9.12
```

其中v1是整数型的向量， v2是浮点型Float64的向量。

用 `length(x)` 求向量 `x` 的元素个数，如

```
In [33]: length(v1)
```

```
Out[33]: 5
```

可以用 `1:5` 定义一个范围， 在仅使用其中的元素值而不改写时作用与 `[1, 2, 3, 4, 5]` 类似。 `1:2:9` 定义带有步长的范围，表示的值与 `[1, 3, 5, 7, 9]` 类似。 范围只需要存储必要的开始、结束、步长信息， 所以更节省空间， 但是不能对其元素进行修改。

```
In [34]: 1:5
```

```
Out[34]: 1:5
```

```
In [35]: 1:2:9
```

```
Out[35]: 1:2:9
```

范围不是向量，用 `collect()` 函数可以将范围转换成向量，如：

```
In [36]: collect(1:5)
```

```
Out[36]: 5-element Array{Int64,1}:
          1
          2
          3
          4
          5
```

向量下标

若 x 是向量， i 是正整数， x[i] 表示向量的第 i 个元素。 第一个元素的下标为1，这种规定与R、FORTRAN语言相同， 但不同于Python、C、C++、JAVA语言。 如

```
In [37]: v1[2]
```

```
Out[37]: 3
```

用 end 表示最后一个元素位置，如：

```
In [38]: v1[end]
```

```
Out[38]: 13
```

对元素赋值将在原地修改元素的值，如

In [39]:
```
v1[2] = -999; v1
```

Out[39]:
```
5-element Array{Int64,1}:
    1
 -999
    4
    9
   13
```

用范围作为下标

下标可以是一个范围，如

In [40]:
```
v1[2:4]
```

Out[40]:
```
3-element Array{Int64,1}:
 -999
    4
    9
```

在这种范围中，用 end 表示最后一个下标，如

In [41]:
```
v1[4:end]
```

Out[41]:
```
2-element Array{Int64,1}:
  9
 13
```

In [42]:
```
v1[1:(end-1)]
```

Out[42]:
```
4-element Array{Int64,1}:
    1
 -999
    4
    9
```

数组作为下标

向量的下标也可以是一个下标数组，如

In [43]:
```
v1[[1, 3, 5]]
```

Out[43]:
```
3-element Array{Int64,1}:
  1
  4
 13
```

取出的多个元素可以修改，可以用 .= 运算符赋值为同一个标量，如：

In [44]:
```
v1[1:3] .= 0; v1
```

Out[44]:
```
5-element Array{Int64,1}:
  0
  0
  0
  9
 13
```

也可以分别赋值，如

In [45]:
```
v1[[1, 3, 5]] = [101, 303, 505]; v1
```

Out[45]:
```
5-element Array{Int64,1}:
 101
   0
 303
   9
 505
```

向量与标量的运算

向量与一个标量作四则运算， 将运算符前面加句点“.”:

```
.+   .-   .*   ./   .^
```

表示向量的每个元素分别与该标量作四则运算， 结果仍是向量。如

In [46]:
```
v1 = [1, 3, 4, 9, 13]
v1 .+ 100
```

Out[46]:
```
5-element Array{Int64,1}:
 101
 103
 104
 109
 113
```

In [47]:
```
100 .- v1
```

Out[47]:
```
5-element Array{Int64,1}:
 99
 97
 96
 91
 87
```

In [48]:
```
v1 .* 2
```

Out[48]:
```
5-element Array{Int64,1}:
  2
  6
  8
 18
 26
```

In [49]:
```
v1 ./ 10
```

Out[49]:
```
5-element Array{Float64,1}:
 0.1
 0.3
 0.4
 0.9
 1.3
```

In [50]:
```
v1 .^ 2
```

Out[50]:
```
5-element Array{Int64,1}:
   1
   9
  16
  81
 169
```

向量与向量的四则运算

两个等长的向量之间作加点的四则运算，表示对应元素作相应的运算。如

In [51]:
```
v1 = [1, 3, 4, 9, 13]
v3 = [2, 5, 6, 7, 10]
v1 .+ v3
```

Out[51]:
```
5-element Array{Int64,1}:
  3
  8
 10
 16
 23
```

向量加减法也可以用不加点的运算符，但对应元素间乘除必须用加点的“.*”和“./”：

In [52]:
```
v1 + v3
```

Out[52]:
```
5-element Array{Int64,1}:
  3
  8
 10
 16
 23
```

In [53]:
```
v1 .- v3
```

Out[53]:
```
5-element Array{Int64,1}:
 -1
 -2
 -2
  2
  3
```

In [54]:
```
v1 .* v3
```

Out[54]:
```
5-element Array{Int64,1}:
   2
  15
  24
  63
 130
```

In [55]:
```
v1 ./ v3
```

Out[55]:
```
5-element Array{Float64,1}:
 0.5
 0.6
 0.6666666666666666
 1.2857142857142858
 1.3
```

向量的比较运算

两个标量之间可以进行如下的比较运算：

```
==  !==  <  <=  >  >=
```

向量每个元素与标量之间、两个向量对应元素之间比较， 只要在前面的比较运算符前面增加句点：

```
.==  .!=  .<  .<=  .>  .>=
```

标量比较结果之间可以用 `&&` 表示“同时成立”， `||` 表示“至少其中之一成立”。 布尔型标量与向量、向量之间可以用 `.&` 表示元素间“与”， `.|` 表示元素间“或”。

向量初始化

用 `zeros(n)` 可以生成元素类型为Float64、元素值为0、长度为n的向量，如

```
In [56]: zeros(3)
```

```
Out[56]: 3-element Array{Float64,1}:
          0.0
          0.0
          0.0
```

用 `zeros(Int64, 3)` 可以生成指定类型的（这里是Int64）初始化向量。如

```
In [57]: zeros(Int64, 3)
```

```
Out[57]: 3-element Array{Int64,1}:
          0
          0
          0
```

用 `Vector{Float64}(undef, n)` 可以生成元素类型为Float64的长度为n的向量， 元素值未初始化，如

In [58]:
```
Vector{Float64}(undef, 3)
```

Out[58]:
```
3-element Array{Float64,1}:
 1.11758147e-315
 7.71951564e-316
 1.9275421e-315
```

类似可以生成其它元素类型的元素值未初始化向量，如

In [59]:
```
Vector{Int}(undef, 3)
```

Out[59]:
```
3-element Array{Int64,1}:
 362376768
 362376784
 362376800
```

用这样的办法为向量分配存储空间后可以随后再填入元素值。

可以用 `collect()` 将一个范围转换成可修改的向量。

向量的循环遍历

可以用 `for` 循环和 `eachindex()` 对向量的每个元素下标遍历访问。 使用 `in` 关键字，格式如

In [60]:
```
for i in eachindex(v1)
  println("v1[", i, "] = ", v1[i])
end
```

```
v1[1] = 1
v1[2] = 3
v1[3] = 4
v1[4] = 9
v1[5] = 13
```

这里 i 是循环产生的向量下标。

也可以直接写下标范围，使用“=”或者 in ，如：

In [61]:
```
for i = 1:length(v1)
    println("v1[", i, "] = ", v1[i])
end
```

```
v1[1] = 1
v1[2] = 3
v1[3] = 4
v1[4] = 9
v1[5] = 13
```

也可以不利用下标而是直接对元素遍历，如

In [62]:
```
for xi in v1
    println(xi)
end
```

```
1
3
4
9
13
```

向量的输出

在脚本文件中， 为了显示某个向量， 可以用 show() 函数， 比如，设 v2 = [1.5, 3, 4, 9.12] ：

In [63]:
```
v2 = [1.5, 3, 4, 9.12]
show(v2)
```

```
[1.5, 3.0, 4.0, 9.12]
```

为了将向量v2按文本格式保存到文件“tmp1.txt”中， 可用：

```
using DelimitedFiles
writedlm("tmp1.txt", v2, ' ')
```

结果文件中每个数占一行。

向量的输入

假设文件“vecstore.txt”中包含如下的内容：

```
1.2 -5  3.6
7.8 9.12 4.11
```

可以用如下代码将文件中的数据读入到一个向量v4中：

```
using DelimitedFiles
v4 = readdlm("vecstore.txt")[:]; v4
```

向量类型的变量

由于Julia的变量仅仅是向实际存储空间的绑定， 所以两个变量可以绑定到同一个向量的存储空间， 修改了其中一个变量的元素值，则另一个变量的元素也被修改了。 如

In [64]:
```
x1 = [1,2,3]
x2 = x1
x2[2] = 100
x1
```

Out[64]:
```
3-element Array{Int64,1}:
   1
 100
   3
```

用 === 可以比较两个变量是否同一对象， 如:

In [65]:
```
x2 === x1
```

Out[65]:
```
true
```

允许两个变量指向同一个对象是有用的， 尤其在函数自变量传递时， 但是在一般程序中这种作法容易引起混淆。 向量（或者数组）作为函数自变量时， 调用函数时传递的是引用， 在函数内可以修改传递进来的向量的元素值。

如果需要制作数组的副本， 用 `copy()` 函数。 如

In [66]:
```
x1 = [1,2,3]
x2 = copy(x1)
x2[2] = -100
x1
```

Out[66]:
```
3-element Array{Int64,1}:
 1
 2
 3
```

In [67]:
```
x2 === x1
```

Out[67]:
```
false
```

向量的有关函数

若 x 是向量， sum(x) 求各个元素的和， prod(x) 求各个元素的乘积。

rand(n) 可以用来生成 n 个标准均匀分布的随机数， 结果为双精度向量。 randn(n) 可以用来生成 n 个标准正态分布的随机数， 结果为双精度向量。

为了判断元素 x 是否属于数组 v ，可以用表达式 x in v 判断， 结果为布尔值。

若 v 是向量， x 是一个元素， push!(v, x) 修改向量 v ， 将 x 添加到向量 v 的末尾。 函数名以叹号结尾时此函数会修改其第一个自变量。 如

In [68]:
```
v3 = [2,3,5]
push!(v3, 7)
v3
```

Out[68]:
```
4-element Array{Int64,1}:
 2
 3
 5
 7
```

若 v 是向量， u 也是一个向量， append!(v, u) 修改向量 v ， 将 u 的所有元素添加到向量 v 的末尾。 如

In [69]:
```
v3 = [2,3,5]
append!(v3, [7,11])
v3
```

Out[69]:
```
5-element Array{Int64,1}:
  2
  3
  5
  7
 11
```

`pop!(v)` 可以返回 `v` 的最后一个元素并从 `v` 中删除此元素。 `popfirst!(v)` 类似。 `splice!(v, k)` 函数可以返回指定下标位置的元素并从 `v` 中删除此元素。 `insert!(v, k, xi)` 函数可以在向量 `v` 的指定下标插入指定的一个元素。

`sort(v)` 返回向量 `v` 按升序排序的结果； `sort!(v)` 直接修改 `v` ，将其元素按升序排序。 如果要用降序排序，可以加选项 `rev=true` 。 `sortperm(v)` 返回将 `v` 的元素从小到大排序所需要的下标序列， 在多个等长向量按照其中一个的次序同时排序时此函数有用。

向量化函数

许多现代的数据分析语言， 如Python, Matlab, R等都存在循环的效率比编译代码低一两个数量级的问题， 在这些语言中， 如果将对向量和矩阵元素的操作向量化，即以向量和矩阵整体来执行计算， 就可以利用语言内建的向量化计算获得与编译代码相近的执行效率。

Julia语言依靠其LLVM动态编译功能， 对向量和矩阵元素循环时不损失效率， 用显式循环处理向量、矩阵与向量化做法效率相近， 有时显式循环效率更高。 但是，向量化计算的程序代码更简洁， 比如上面的两个向量之间的四则运算的加点格式。

Julia中的函数， 包括自定义函数， 如果可以对单个标量执行， 将函数名加后缀句点后， 就可以变成向量化版本， 对向量和矩阵执行。 如

```
In [70]: sqrt.([1,2,3])
```

```
Out[70]: 3-element Array{Float64,1}:
          1.0
          1.4142135623730951
          1.7320508075688772
```

这种向量化对于多个自变量的函数也成立。

元组

与向量类似的一种数据类型称为元组（tuple）。 如

In [71]:
```
(1, 2, 3)
```

Out[71]:
```
(1, 2, 3)
```

In [72]:
```
(1, "John", 5.1)
```

Out[72]:
```
(1, "John", 5.1)
```

元组的元素不要求属于同一类型。

单个元素的元组要有逗号分隔符，如 `(1,)` 是单个元素的元组， 而 `(1)` 不是元组。

元组表面上类似于一维数组， 但是元组属于不可修改(immutable)类型， 不能修改其中的元素。 其存储也与数组不同。

可以用 `tuple()` 函数生成元组。 可以用类似一维数组的方法对元组取子集， 如 `x[1]` , `x[2:3]` 等。 如：

In [73]:
```
x = ('a', 'b', 'c', 'd')
typeof(x)
```

Out[73]:
```
NTuple{4,Char}
```

In [74]:
```
x[1]
```

Out[74]:
```
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

In [75]:
```
x[2:3]
```

Out[75]:
```
('b', 'c')
```

```
x[2] = 'x'
```

结果出错:

```
MethodError: no method matching setindex!(::NTuple{4,Char}, ::Char, ::Int64)

Stacktrace:
 [1] top-level scope at In[76]:1
```

元组可以看作一个整体参加比较， 比较方法类似于字典序，如:

In [76]:
```
(1, 3, 5) < (1, 3, 6)
```

Out[76]:
```
true
```

可以利用元组写法对变量同时赋值， 如

In [77]:
```
a, b = 13, 17
println("a=", a, " b=", b)
```

```
a=13 b=17
```

元组赋值的右侧也可以是数组等其它序列类型，如

In [78]:
```
a, b = [19, 23]
println("a=", a, " b=", b)
```

```
a=19 b=23
```

自定义函数可以返回元组， 从而返回多个值，见下面的自定义函数章节。

字典

Julia提供了一种Dict数据类型， 是映射的集合， 每个元素是从一个“键值”到另一个“值”的映射， 元素之间没有固定次序。如

In [79]:
```
d = Dict("name" => "Li Ming", "age" => 18)
```

Out[79]:
```
Dict{String,Any} with 2 entries:
  "name" => "Li Ming"
  "age"  => 18
```

访问单个元素如

In [80]:
```
d["age"]
```

Out[80]:
```
18
```

这种功能类似于R语言中用元素名作为下标, 但R中还可以用序号访问元素， 而字典中的元素没有次序，不能用序号访问。

读取字典中单个键的对应值也可以用 `get(d, key, default)` 的格式， 其中 `default` 是元素不存在时的返回值。如：

In [81]:
```
get(d, "age", "")
```

Out[81]:
```
18
```

可以用 `haskey(d, key)` 检查某个键值是否存在，如：

In [82]:
```
haskey(d, "gender")
```

Out[82]:
```
false
```

给不存在的键值赋值就可以增加一对映射，如

In [83]:
```
d["gender"] = "Male"
```

Out[83]:
```
"Male"
```

也可以用二元组的数组作为初值定义字典，如

In [84]:
```
d2orig = [('a', 1), ('b', 2), ('c', 3), ('d', 4)]
d2 = Dict(d2orig)
```

Out[84]:
```
Dict{Char,Int64} with 4 entries:
  'a' => 1
  'c' => 3
  'd' => 4
  'b' => 2
```

`delete!(d, key)` 可以删除指定的键值对。 `get!(d, key, default)` 可以在指定键值不存在时用 `default` 值填入该键值， 已存在时就不做修改， 两种情况下都返回新填入或原有的键对应的值。

可以用 `keys()` 函数遍历各个键值，次序不确定：

In [85]:
```
for k in keys(d2)
    println(k, " => ", d2[k])
end
```

```
a => 1
c => 3
d => 4
b => 2
```

字典存储并没有固定的存储次序。 为了在遍历时按键值的次序， 需要使用如下的效率较低的方法：

In [86]:
```
for k in sort(collect(keys(d2)))
    println(k, " => ", d2[k])
end
```

```
a => 1
b => 2
c => 3
d => 4
```

可以用 `values()` 遍历各个值，但也没有固定次序。比如

In [87]:
```
collect(values(d2))
```

Out[87]:
```
4-element Array{Int64,1}:
 1
 3
 4
 2
```

可以直接用二元组对字典遍历，如

In [88]:
```
for (k,v) in d2
    println(k, " => ", v)
end
```

```
a => 1
c => 3
d => 4
b => 2
```

查找某个键值或者值是否存在用 `in` 运算， 这比在数组中查找要高效， 因为字典的基础是杂凑表(hash table)， 其查找时间不随元素个数增加而增加。 如：

```
In [89]: "age" in keys(d)
```

```
Out[89]: true
```

```
In [90]: 19 in values(d)
```

```
Out[90]: false
```

字典的键值可以是字符串、整数值、浮点数值， 还可以是元组， 不允许取数组这样的可变类型（mutable）。

可以在生成字典时指定键值和值的数据类型， 格式为 `Dict{S, T}(...)` 。如：

```
In [91]: Dict{String, Int64}("apple" => 1, "pear" => 2, "orange" => 3)
```

```
Out[91]: Dict{String,Int64} with 3 entries:
           "pear"   => 2
           "orange" => 3
           "apple"  => 1
```

元组是可以命名的， 这使得其在一定程度上类似于字典。 如

```
In [92]: tn1 = (name="John", age=32)
         tn1[:name]
```

```
Out[92]: "John"
```

但是要注意有名元组用变量名访问时用的是符号(Symbol)， 即不写成字符串的变量名前面有冒号。

字典应用：频数表

在基本的描述统计中， 经常需要对某个离散取值的变量计算其频数表， 即每个不同值出现的次数。 如果不利用字典类型， 可以先找到所有的不同值， 将每个值与一个序号对应， 然后建立一个一维数组计数， 每个数组元素与一个变量值对应。

利用字典， 我们不需要预先找到所有不同值，而是直接用字典计数， 每个键值是一个不同的变量值， 每个值是一个计数值。 如

In [93]:
```
sex = ["F", "M", "M", "F", "M"]
freqs = Dict()
for xi in sex
    if xi in keys(freqs)
        freqs[xi] += 1
    else
        freqs[xi] = 1
    end
end
freqs
```

Out[93]:
```
Dict{Any,Any} with 2 entries:
  "M" => 3
  "F" => 2
```

对字典可以用 `get()` 函数提取某个键值对应的值， 并在键值不存在时返回指定的缺省值。 这样，上面的例子可以简化写成：

In [94]:
```
sex = ["F", "M", "M", "F", "M"]
freqs = Dict()
for xi in sex
    freqs[xi] = get(freqs, xi, 0) + 1
end
freqs
```

Out[94]:
```
Dict{Any,Any} with 2 entries:
  "M" => 3
  "F" => 2
```

集合类型

Julia中Set是集合类型。 用 `Set()` 生成一个集合，如 `Set(1:3)` ， `Set(['a', 'b', 'c'])` 。 支持集合的常见运算， `union()` ， `intersect()` ， `setdiff()` ， `symdiff()` ， `issetequal()` ， `issubset()` 。 子集关系也可以用运算符 ⊆ ， ⊇ ， ⊈ ， ⊉ 表示。 属于关系用 `in` ， ∈ ， ∋ ， ∉ ， ∌ 表示。

对数组 `x` ， `unique(x)` 返回由 `x` 的不同元素组成的数组。

矩阵和数组

前面讲的向量当元素类型相同时可以看作一维数组，不区分行向量还是列向量， 在参与矩阵运算时看作列向量。

矩阵是二维数组，有两个下标：行下标和列下标。

数组(Array)是Julia中的数据类型， 有一维、二维、多维等， 区别在于引用一个元素时所用下标个数， 数组中的元素属于相同的基本类型， 比如， 元素类型都是Int64， 都是Float64， 都是String， 等等。

为了在程序中直接输入一个矩阵， 可以在方括号内两个同行的元素之间用空格分隔， 两行之间用分号分隔，如

In [95]:
```
A1 = [1 2 3; 4 5 6]
```

Out[95]:
```
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

注意结果显示这是一个 `2×3 Array{Int64,2}` ， 即一个两行三列的元素都是Int64类型的二维数组。

对矩阵 `x` ， `size(x,1)` 返回行数， `size(x,2)` 返回列数。如

In [96]:
```
println("(", size(A1, 1), ", ", size(A1, 2), ")")
```

```
(2, 3)
```

输入行向量的例子：

In [97]:
```
[1 2 3]
```

Out[97]:
```
1×3 Array{Int64,2}:
 1  2  3
```

Julia将一维向量看作列向量。所以，如下的用分号分隔的输入列向量的程序得到的是一维数组：

In [98]:
```
[1; 2; 3]
```

Out[98]:
```
3-element Array{Int64,1}:
 1
 2
 3
```

矩阵下标

设 A 是矩阵，则 A[i,j] 表示 A 的第 i 行第 j 列元素，如:

In [99]:
```
A1[2,3]
```

Out[99]:
```
6
```

给元素赋值可以在矩阵中修改元素值，如：

In [100]:
```
A1[2,3] = -6; A1
```

Out[100]:
```
2×3 Array{Int64,2}:
 1  2   3
 4  5  -6
```

矩阵列和行

设 A 是矩阵， 则 A[:, j] 表示 A 的第 j 列元素组成的向量（一维数组），如

In [101]:
```
A1[:, 2]
```

Out[101]:
```
2-element Array{Int64,1}:
 2
 5
```

A[i, :] 表示 A 的第 i 行元素组成的向量（一维数组），如

In [102]:
```
A1[2, :]
```

Out[102]:
```
3-element Array{Int64,1}:
  4
  5
 -6
```

取出后的列或者行不分行向量和列向量。

子矩阵

如果 A 是矩阵， I 和 J 是范围或者向量， 则 A[I,J] 表示 A 的行号在 I 中的行与列号在 J 中的列交叉所得的子矩阵，如

In [103]:
```
A1[1:2, 2:3]
```

Out[103]:
```
2×2 Array{Int64,2}:
 2   3
 5  -6
```

用冒号":"作为行下标或列下标表示取该维的全部下标。 如果取出的子矩阵仅有一行或者仅有一列， 则结果退化成一维数组， 不再是矩阵。

给子矩阵赋值为一个同样大小的子矩阵给对应元素赋值，如

```
In [104]: A1[1:2, 2:3] = [102 103; 202 203]; A1
```

```
Out[104]: 2×3 Array{Int64,2}:
           1  102  103
           4  202  203
```

矩阵初始化

用 `zeros(m, n)` 可以生成元素类型为Float64、元素值为0的$m \times n$矩阵，如

```
In [105]: zeros(2, 3)
```

```
Out[105]: 2×3 Array{Float64,2}:
           0.0  0.0  0.0
           0.0  0.0  0.0
```

在用方括号格式生成向量或矩阵时， 可以在方括号前面写元素类型名称， 要求生成某种类型的数组。如：

```
In [106]: Float64[1,3,5]
```

```
Out[106]: 3-element Array{Float64,1}:
           1.0
           3.0
           5.0
```

在上例中，如果不指定类型，结果将自动解析为Int64类型， 因为给出的元素值都是整数。

又如：

In [107]:
```
Float64[1 3 5; 2 4 6]
```

Out[107]:
```
2×3 Array{Float64,2}:
 1.0  3.0  5.0
 2.0  4.0  6.0
```

用 `Array{Float64}(undef, m, n)` 可以生成元素类型为Float64的$m \times n$矩阵， 元素值未初始化，如

In [108]:
```
Array{Float64}(undef, 2, 3)
```

Out[108]:
```
2×3 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0
```

类似可以生成其它元素类型的矩阵，如

In [109]:
```
Array{Int64}(undef, 2, 3)
```

Out[109]:
```
2×3 Array{Int64,2}:
 0  0  0
 0  0  0
```

用这样的办法先为矩阵分配存储空间，然后可以再填入元素值。

`Array{T}(undef, m, n, k[, ...])` 这种格式可以生成元素类型为T， 初始值无定义，维数和每一维的下标长度由 `undef` 后面的参数给出， 如 `Array{Char}(undef, 99)` 是元素为字符的长度为99的字符数组， 元素初值无定义。 `Array{UInt8}(undef, 3, 2, 5)` 是元素类型为UInt8，$3 \times 2 \times 5$形状的三维数组， 可以看成是5个3×2矩阵， 元素初值无定义。

矩阵元素遍历

可以对行下标和列下标分别循环，行下标变化最快，如

In [110]:
```
A1 = [1 2 3; 4 5 6]
```

Out[110]:
```
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

In [111]:
```
for j = 1:size(A1,2), i = 1:size(A1,1)
  println("A1[", i, ", ", j, "] = ", A1[i, j])
end
```

```
A1[1, 1] = 1
A1[2, 1] = 4
A1[1, 2] = 2
A1[2, 2] = 5
A1[1, 3] = 3
A1[2, 3] = 6
```

这相当于：

In [112]:
```
for j = 1:size(A1,2)
    for i = 1:size(A1,1)
      println("A1[", i, ", ", j, "] = ", A1[i, j])
    end
end
```

```
A1[1, 1] = 1
A1[2, 1] = 4
A1[1, 2] = 2
A1[2, 2] = 5
A1[1, 3] = 3
A1[2, 3] = 6
```

之所以在两重循环时让行下标变化最快是因为矩阵的存储是按列存储的， 所以遍历时先遍历第一列的各个行，再遍历第二列的各个行，……，这样效率较高。 如果不考虑效率问题，也可以逐行遍历如下：

In [113]:
```
for i = 1:size(A1,1), j = 1:size(A1,2)
  println("A1[", i, ", ", j, "] = ", A1[i, j])
end
```

```
A1[1, 1] = 1
A1[1, 2] = 2
A1[1, 3] = 3
A1[2, 1] = 4
A1[2, 2] = 5
A1[2, 3] = 6
```

另一种办法是用类似于向量遍历的方法，如：

In [114]:
```
for i in eachindex(A1)
  println("A1[", i, "] = ", A1[i])
end
```

```
A1[1] = 1
A1[2] = 4
A1[3] = 2
A1[4] = 5
A1[5] = 3
A1[6] = 6
```

从上例可以看出矩阵是按列存储的。

矩阵读写

设当前目录中文件“vecstore.txt”中包含如下内容：

```
1.2 -5  3.6
7.8 9.12 4.11
```

将文件中的内容每一行看作矩阵的一行， 文件中保存了一个2×3矩阵。 读入方法如下:

```
using DelimitedFiles
Ain = readdlm("vecstore.txt"); Ain
```

考虑上面的Ain矩阵，为了将其按文本文件格式保存到“tmp2.txt”中， 用如下程序：

```
writedlm("tmp2.txt", Ain, ' ')
```

矩阵与标量的四则运算

矩阵与一个标量之间用加点的四则运算符号进行运算， 与向量和标量之间的运算类似， 表示矩阵的每个元素和该变量的四则运算， 结果仍为矩阵。如

In [115]:
```
A1 = [1 2 3; 4 5 6]
```

Out[115]:
```
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

In [116]:
```
A1 .+ 100
```

Out[116]:
```
2×3 Array{Int64,2}:
 101  102  103
 104  105  106
```

In [117]:
```
100 .- A1
```

Out[117]:
```
2×3 Array{Int64,2}:
 99  98  97
 96  95  94
```

In [118]:
```
A1 .* 2
```

Out[118]:
```
2×3 Array{Int64,2}:
 2   4   6
 8  10  12
```

In [119]:
```
A1 ./ 10
```

Out[119]:
```
2×3 Array{Float64,2}:
 0.1  0.2  0.3
 0.4  0.5  0.6
```

In [120]:
```
A1 .^ 2
```

Out[120]:
```
2×3 Array{Int64,2}:
  1   4   9
 16  25  36
```

两个矩阵之间的四则运算

两个同样大小的矩阵之间用加点的四则运算符号进行运算， 表示两个矩阵的对应元素的运算。如

In [121]:
```
A2 = A1 .* 100
```

Out[121]:
```
2×3 Array{Int64,2}:
 100  200  300
 400  500  600
```

In [122]:
```
A1 .+ A2
```

Out[122]:
```
2×3 Array{Int64,2}:
 101  202  303
 404  505  606
```

In [123]:
```
A2 .- A1
```

Out[123]:
```
2×3 Array{Int64,2}:
  99  198  297
 396  495  594
```

In [124]:
```
A1 .* A2
```

Out[124]:
```
2×3 Array{Int64,2}:
  100   400   900
 1600  2500  3600
```

In [125]:
```
A2 ./ A1
```

Out[125]:
```
2×3 Array{Float64,2}:
 100.0  100.0  100.0
 100.0  100.0  100.0
```

矩阵加减也可以用不加点的运算符:

In [126]:
```
A1 + A2
```

Out[126]:
```
2×3 Array{Int64,2}:
 101  202  303
 404  505  606
```

`A1 - A2` 也可以。

但是， `A1 * A2` 表示矩阵乘法， `A1 / A2` 表示解方程组或者最小二乘， 都不是对应元素间的运算。

矩阵乘法

用 `A * B` 表示矩阵乘法。 如

In [127]:
```
A3 = [11 12; 21 22]
```

Out[127]:
```
2×2 Array{Int64,2}:
 11  12
 21  22
```

In [128]:
```
A3 * A1
```

Out[128]:
```
2×3 Array{Int64,2}:
  59   82  105
 109  152  195
```

一个矩阵与一个向量（一维数组）作矩阵乘法， 向量自动变成列向量，如:

In [129]:
```
A3 * [1, -1]
```

Out[129]:
```
2-element Array{Int64,1}:
 -1
 -1
```

注意结果是向量（一维数组），而不是2×1矩阵（二维数组）。

行向量可以直接表示成方括号内多个数值之间用空格分隔的格式，如

In [130]:
```
[1,  -1] * [1  -1]
```

Out[130]:
```
2×2 Array{Int64,2}:
  1  -1
 -1   1
```

又如

In [131]:
```
[1  -1] * A3
```

Out[131]:
```
1×2 Array{Int64,2}:
 -10  -10
```

注意结果是1×2矩阵，即行向量，而不是向量。 向量是一维数组，行向量是二维数组。

从以上例子可以看出在矩阵运算中向量可以看成是列向量， 矩阵乘法结果如果是列向量，也会表示成向量（一维数组）。

矩阵转置

用 `A'` 表示矩阵 `A` 的共轭转置，对实值矩阵就是转置。 如

In [132]:
```
A1
```

Out[132]:
```
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

In [133]:
```
A1'
```

Out[133]:
```
3×2 LinearAlgebra.Adjoint{Int64,Array{Int64,2}}:
 1  4
 2  5
 3  6
```

两个向量x和y的内积用LinearAlgebra包的 `dot(x, y)` 表示，结果为一个标量。如

In [134]:
```
import LinearAlgebra
LinearAlgebra.dot([1, -1], [2, 3])
```

Out[134]:
```
-1
```

矩阵合并

设 `A`，`B`，`C` 是矩阵， `[A B C]` 将三个矩阵横向合并， 这等同于 `hcat(A, B, C)`； `[A; B; C]` 将三个矩阵纵向合并， 这等同于 `vcat(A, B, C)` 。

矩阵求逆和解线性方程组

`inv(A)` 表示A^{-1}。如

In [135]:
```
A4 = [1 3; 3 1]
```

Out[135]:
```
2×2 Array{Int64,2}:
 1  3
 3  1
```

In [136]:
```
inv(A4)
```

Out[136]:
```
2×2 Array{Float64,2}:
 -0.125   0.375
  0.375  -0.125
```

用 `A \ B` 表示$A^{-1}B$， 当 `B` 是向量或者列向量时， 就是求解线性方程组$Ax = B$中的x。 如

In [137]:
```
A4 \ [-2, 2]
```

Out[137]:
```
2-element Array{Float64,1}:
  1.0
 -1.0
```

Julia提供了线性代数计算所需的一些函数， 有些在Base包中，有些则在LinearAlgebra包中。 参见LinearAlgebra包的文档。

字符串处理

字符串下标与遍历

`length()` 求字符串中字符个数， 每个汉字算一个字符。 对字符串用下标访问，是按字节计算的， 因为一个汉字可能占据多个字节， 所以有些下标位置能返回字符， 有些下标会出错。 如

In [138]:
```
s = "汉字123"
length(s)
```

Out[138]:
```
5
```

In [139]:
```
s[1]
```

Out[139]:
```
'汉': Unicode U+6C49 (category Lo: Letter, other)
```

In [140]:
```
s[2]
```

```
StringIndexError("汉字123", 2)

Stacktrace:
 [1] string_index_err(::String, ::Int64) at .\strings\string.jl:12
 [2] getindex_continued(::String, ::Int64, ::UInt32) at .\strings\string.jl:220
 [3] getindex(::String, ::Int64) at .\strings\string.jl:213
 [4] top-level scope at In[140]:1
```

In [141]:
```
s[7]
```

Out[141]:
```
'1': ASCII/Unicode U+0031 (category Nd: Number, decimal digit)
```

用范围下标，返回字符串结果，但是下标的计算仍是字节，如：

In [142]:
```
s[1:1]
```

Out[142]:
```
"汉"
```

In [143]:
```
s[2:4]
```

```
StringIndexError("汉字123", 2)

Stacktrace:
 [1] string_index_err(::String, ::Int64) at .\strings\string.jl:12
 [2] getindex(::String, ::UnitRange{Int64}) at .\strings\string.jl:249
 [3] top-level scope at In[143]:1
```

这种设计是为了能够高效地访问字符串中的字符， 因为使用UTF-8编码，每个字符所用字节个数可能会不同。

函数 `firstindex(s)` 返回 `s` 中第一个字符的下标， `lastindex(s)` 返回 `s` 中最后一个字符的下标， `thisind(s, i)` 返回字节 `i` 所在字符的开始字节位置， `nextind(s, i)` 返回 `s` 中跟随在下标 `i` 后面的合法字符字节下标， `prevtind(s, i)` 返回 `s` 中跟随在下标 `i` 前面的合法字符字节下标。 如：

In [144]:
```
s[1]
```

Out[144]:
```
'汉': Unicode U+6C49 (category Lo: Letter, other)
```

In [145]:
```
nextind(s, 1)
```

Out[145]:
```
4
```

In [146]:
```
s[4]
```

Out[146]:
```
'字': Unicode U+5B57 (category Lo: Letter, other)
```

对字符串 `s`， `collect(s)` 可以将字符串 `s` 转换成一个字符数组， 每个元素是字符串中的一个字符：

In [147]:
```
collect(s)
```

Out[147]:
```
5-element Array{Char,1}:
 '汉'
 '字'
 '1'
 '2'
 '3'
```

对字符串中每个字符也可以用 `for` 循环遍历，如

In [148]:
```
for ch in "汉字123"
    println(ch)
end
```

```
汉
字
1
2
3
```

读写字符串

函数 `readlines(filename)` 指定一个输入的文本文件名， 将文件的各行读入为一个字符串数组， 每个元素保存一行，默认不带有换行标志。

可以用如下方式对输入文件的每行循环：

```
for line in eachline(filename)
    ## 对line进行一些操作
end
```

Julia没有提供一个 `writelines()` 函数， 自定义及测试如下：

```
function writelines(lines::Array{AbstractString, 1}, filename::AbstractString)
    open(filename, "w") do io
        for line in lines
            println(io, line)
        end
    end
    return
end

lines = map(string, 1:3)
writelines(lines, "tmp1.txt")
```

字符串函数

Julia有很多与字符串处理的函数。 如 `length` , `sizeof` , `^` 或 `repeat` , `lpad` , `rpad` , `strip` , `lstrip` , `rstrip` , `chop` , `chomp` , `uppercase` , `lowercase` , `titlecase` , `uppercasefirst` , `lowercasefirst` , `reverse` 。

`join(x, dlm)` 将字符串数组 `x` 的元素用指定的分隔符分隔后连接成一个长字符串，如

```
join(["abc", "汉字", "1"], "-+-")
## "abc-+-汉字-+-1"
```

`collect(s)` 将字符串 `s` 转换成一个字符数组。

`string(x)` 将数值 `x` 转换成字符串表示， `string(x, y, z)` 将 `x, y, z` 都转换成字符串然后连接起来。 `repr(x)` 将表达式 `x` 的值表示成字符串。 `parse(Int64, s)` 将字符串 `s` 中的数值转换成整数类型， `parse(Float64, s)` 将字符串 `s` 中的数值转换成Float64类型。

`occursin(needle, haystack)` (v1.0)返回 `needle` 是否出现在 `haystack` 中， 如

```
occursin("oar", "board")
## true
```

`startswith(s, prefix)` , `endswith(s, suffix)` 检查字符串前缀和后缀。

`first(s, n)` , `last(s, n)` 取出头或者尾部指定个数的字符。

`split()` 函数将分隔的字符串拆分成字符串数组，如

```
println(split("1, 2, 3", ","))
## SubString{String}["1", " 2", " 3"]
```

字符串插值

在字符串中可以用 `$变量名` 或 `$(表达式)` 的格式插入变量值或者表达式值。 如:

In [149]:
```
x = 123
"x = $x"
```

Out[149]: "x = 123"

In [150]:
```
"123 + 100 = $(123+100)"
```

Out[150]: "123 + 100 = 223"

正则表达式

Julia支持正则表达式功能。 Julia的正则表达式采用Perl规则。

正则表达式的写法对初学者比较困难， 这里不进行详细讲解， 读者可以找一本专门讲正则表达式的书， 或者其它编程语言中讲到正则表达式的书。 例如， 本文作者讲R语言的书中有一章讲文本处理， 其中比较详细地讲解了正则表达式的语法， 见：
http://www.math.pku.edu.cn/teachers/lidf/docs/Rbook/html/_Rbook/text.html (http://www.math.pku.edu.cn/teachers/lidf/docs/Rbook/html/_Rbook/text.html)

Julia中正则表达式模式字符串是双撇号界定的特殊字符串开始双撇号之前加 `r` 字母作为前缀， 还可以在字符串结尾的双撇号滞后增加一些表示选项的字母， 遵从Perl语言的约定， 如 `i` 表示不区分大小写， `m` 表示行首和行尾的匹配是针对每一行进行的， `s` 表示句点可以匹配换行符，等等。 比如，

```
pat=r"John"i
```

会不区分大小写地匹配 `John` 单词。

加了 `r` 前缀的正则表达式中的特殊字符 `\` 不需要写成两个， 如 `r"\w"` 就表示一个字母、数字、下划线， 而不是写成 `r"\\w"` 。

可以用 `Regex()` 函数将一个字符串型的表达式转换成正则表达式类型， 这样可以动态地构造正则表达式。

下面的正则表达式对电子邮箱地址进行简单地匹配：

In [151]:
```
pat = r".+@.+"
occursin(pat, "jason@abc.com")
```

Out[151]: true

`occursin()` 仅返回是否匹配。 用 `match()` 函数返回匹配结果， 设结果为 `m` ， 则 `m.match()` 返回匹配的整个字符串， `m.offset()` 返回匹配的开始字符下标， `m.offsets()` 返回匹配的各个子模式的开始字符下标， `m.captures()` 返回匹配的各个子模式。 如

In [152]:
```
pat = r"([a-zA-Z0-9_.]+)@([a-zA-Z0-9_.]+)"
m = match(pat, "==jason@abc.com==")
println(m.match)
println(m.offset)
println(m.offsets)
println(m.captures)
```

```
jason@abc.com
3
[3, 9]
Union{Nothing, SubString{String}}["jason", "abc.com"]
```

用 `findfirst` 找到某个模式首次出现，`findlast` 找到某个模式最后一次出现， `findnext` 找到某个模式在指定位置之后的首次出现， `findprev` 找到某个模式在指定位置之前的首次出现。 返回值为匹配的字节位置范围。 如:

In [153]:
```
pat = r"([a-zA-Z0-9_.]+)@([a-zA-Z0-9_.]+)"
s = "张三: jason@abc.com; 李四: tom@bde.com"
findfirst(pat, s)
```

Out[153]: `10:22`

In [154]:
```
s[10:22]
```

Out[154]: `"jason@abc.com"`

In [155]:
```
findnext(pat, s, 23)
```

Out[155]: `33:43`

In [156]:
```
s[33:43]
```

Out[156]: `"tom@bde.com"`

In [157]:
```
findlast(".com", s)
```

Out[157]:
```
40:43
```

`replace()` 函数可以用来从字符串中替换某个指定的模式为另外的替换值， 替换字符串使用 `s` 前缀的字符串， 也可以用 `SubstitutionString()` 将一个字符串型表达式转换成替换字符串类型。

In [158]:
```
pat = r"([a-zA-Z0-9_.]+)@([a-zA-Z0-9_.]+)"
reppat = s"\1-nospam@\2"
replace("==jason@abc.com==", pat => reppat)
```

Out[158]:
```
"==jason-nospam@abc.com=="
```

其中包含 `s` 前缀的字符串是替换字符串，其中的 `\1` 等的反斜杠不需要重复。

可以用 `eachmatch()` 函数提供对对多处匹配的循环。如

In [159]:
```
pat = r"\w+"
for imatch in eachmatch(pat, "It is raining.")
    println("\"$(imatch.match)\"")
end
```

```
"It"
"is"
"raining"
```

文件输入输出

文本文件读写

对文本文件， `readlines(filename)` 函数根据输入的文件名读入文件的各行为字符串数组， 每个元素是一行， 缺省不包含换行符。 用 `read(filename, String)` 将整个文件读入为一个长字符串。

用 `fh = open(filename)` 打开指定的文件用于读取， 这里 `fh` 称为一个文件句柄。 读取如 `readline(fh)` 。 用 `close(fh)` 关闭 `fh` 对应的文件。

用 `fh = open(filename, "w")` 打开指定的文件用于输出， 写入如 `println(fh, "x = ", x)` 。 结束写入后用 `close(fh)` 关闭输出文件。

Julia 1.0中没有 `writelines()` 函数， 可以自定义如下的函数：

```
function writelines(lines::Array{AbstractString, 1}, filename::AbstractString)
    open(filename, "w") do io
        for line in lines
            println(io, line)
        end
    end
    return
end
```

文件和目录信息

文件保存在目录（directory）中。 用 `pwd()` 返回当前的工作目录， 不给定具体路径的文件名默认在工作目录中。 用 `cd(path)` 设定 `path` 为当前工作目录。

用 `abspath(filename)` 求一个文件的绝对路径。 用 `joinpath()` 将目录与文件连接成一个完整路径。

用 `ispath(filename)` 判断每个文件或者目录是否存在。 用 `isdir(filename)` 判断某个路径是否目录， 用 `isfile(filename)` 判断某个路径是否文件。

用 `readdir(path)` 返回指定目录的文件和子目录列表， 无 `path` 时对应当前工作目录。

程序控制结构

复合表达式

用 `begin ... end` 可以将多行的多个表达式组合起来当作一个表达式， 复合表达式的值是其中最后一个表达式的值。 如

In [160]:
```
z = begin
  x = 1
  y = 2
  x + y
end
z
```

Out[160]: 3

多个表达式也可以用分号分隔后写在圆括号中，作为一个复合表达式，如

In [161]:
```
z = (x = 1; y = 2; x + y)
z
```

Out[161]: 3

短路与运算以及分支结构

`&&` 是一种短路运算， 表达式 `cond && expr` 仅当 `cond` 为true时才计算（运行） `expr` ， 所以这种写法经常用作程序分支的简写： 条件 `cond` 为真时执行 `expr` ， 否则不执行。

比如，在计算x的平方根之前，先判断其非负：

In [162]:
```
x = -1.44
x < 0 && println("平方根计算：自变量定义域错误，x=", x)
```

平方根计算：自变量定义域错误，x=-1.44

短路或运算以及分支结构

`||` 是一种短路或运算，表达式 `cond || expr` 仅当cond为false时才计算（运行） `expr` ， 所以这种写法经常作为程序分支的缩写： 条件cond为假时才执行 `expr` ，否则不执行。

比如，求平方根时当自变量不为负时才计算平方根：

In [163]:
```
x < 0 || (y = sqrt(x))
```

Out[163]: true

if–end结构

可以用 `if cond ... end` 结构在条件 `cond` 成立时才执行某些语句，如

In [164]:
```
x = 1.44
if x >= 0
  y = sqrt(x)
  println("√", x, " = ", y)
end
```

```
√1.44 = 1.2
```

注意条件不需要用括号包围，结构以 `end` 语句结尾。

if–else–end结构

`if cond ... else ... end` 结构当条件成立时执行第一个分支中的语句， 当条件不成立时执行第二个分支中的语句。 如

In [165]:
```
x = -1.44
if x >= 0
  y = sqrt(x)
  println("√", x, " = ", y)
else
  y = sqrt(-x)
  println("√", x, " = ", y, "i")
end
```

```
√-1.44 = 1.2i
```

if-elseif-else-end结构

`if cond1 ... elseif cond2 ... else ... end` 可以有多个分支， 有多个条件 `cond1` ， `cond2` , ..., 依次判断各个条件，那个条件成立就执行对应分支的语句， 所有条件都不成立则执行 `else` 分支的语句。 条件 `cond2` 隐含条件 `cond1` 不成立， `cond3` 隐含 `cond1` 和 `cond2` 都不成立， 依此类推。 例如

In [166]:
```
age = 35
if age < 18
  println("未成年")
elseif age < 60
  println("中青年")
elseif age < 100
  println("老年")
else
  println("老寿星! ")
end
```

```
中青年
```

三元运算符

可以用 `cond ? expr1 : expr2` 表示比较简单的两分支选择， 当 `cond` 成立时结果为 `expr1` 的结果， 当 `cond` 不成立时结果为 `expr2` 的结果。 如

In [167]:
```
x = -1.44
y = x >= 0 ? sqrt(x) : sqrt(-x)
```

Out[167]: 1.2

for循环

for循环一般是沿着某个范围进行计算或处理，格式如下：

```
for loopvar = a:b
  expr1
  expr2
  ...
end
```

其中 loopvar 是自己命名的循环变量名， for结构块内的语句（表达式）先对 loopvar=a 运行， 再对 loopvar=a+1 运行, 最后对 loopvar=b 运行，然后结束。 如

In [168]:
```
for i=1:3
  y = i^3
  println(i, "^3 = ", y)
end
```

```
1^3 = 1
2^3 = 8
3^3 = 27
```

范围也可以是 `1:2:9` ， `0:0.1:1` 这样的带有跨度（增量）的， 可以是倒数的如 `3:-1:1` 表示3, 2, 1。

要小心的是，循环内的变量都是循环的局部变量， 比如上例中的 `i` 和 `y` 都是局部的， 退出循环后无法访问 `i` 和 `y` 的值。 循环内可以读取循环外部的变量值但是不能修改循环外部变量的值。

当循环在自定义函数内时， 可以读写循环外的变量， 但是循环变量以及循环内新定义的变量仍不能在退出循环后保留。

在自定义函数之外使用循环外的变量时， 应该在循环开始处用 `global` 关键字声明该变量。 如：

```
n = 5
p = 1
for i = 1:n
    global p
    p *= i
end
println(p)
## 120
```

Julia程序在Jupyter界面运行时， 可以用选项使得循环内的变量能够直接访问循环外的变量， 不需要 `global` 声明。 但是，为兼容性起见， 还是应该按语法要求加上 `global` 声明。

对向量元素循环

用 `in` 关键字，可以使得循环变量遍历某个向量的元素，如：

In [169]:
```
x = [2, 3, 5, 7, 11]
for i in x
  y = i^3
  println(i, "^3 = ", y)
end
```

```
2^3 = 8
3^3 = 27
5^3 = 125
7^3 = 343
11^3 = 1331
```

向量下标循环

设 x 是一个向量，上面的做法可以遍历 x 每个元素。 有时还需要按照向量的下标遍历，这时使用 eachindex(x) ，如

In [170]:
```
x = [2, 3, 5, 7, 11]
for i in eachindex(x)
  println("Prime No. ", i, " = ", x[i])
end
```

```
Prime No. 1 = 2
Prime No. 2 = 3
Prime No. 3 = 5
Prime No. 4 = 7
Prime No. 5 = 11
```

用 enumerate() 可以对下标与值同时循环，如

In [171]:
```
x = [2, 3, 5, 7, 11]
for (i, xi) in enumerate(x)
  println("Prime No. ", i, " = ", xi)
end
```

```
Prime No. 1 = 2
Prime No. 2 = 3
Prime No. 3 = 5
Prime No. 4 = 7
Prime No. 5 = 11
```

理解(Comprehension)

对向量的循环，经常可以表达成一种称为comprehension的语法。 例如，为了生成1, 2, 3的立方的向量，可以写成

In [172]:
```
xcube = [i^3 for i=1:3]
```

Out[172]:
```
3-element Array{Int64,1}:
  1
  8
 27
```

对前5个素数作立方，可以写成

In [173]:
```
x = [2, 3, 5, 7, 11]
y = [z^3 for z in x]
```

Out[173]:
```
5-element Array{Int64,1}:
   8
  27
 125
 343
1331
```

当然，这样的简单运算也可以用加点四则运算表示：

In [174]:
```
x = [2, 3, 5, 7, 11]
y = x .^ 3
```

Out[174]:
```
5-element Array{Int64,1}:
   8
  27
 125
 343
1331
```

两重for循环

for循环可以嵌套，如

In [175]:
```
for i=1:9
  for j=1:i
    print(j, "×", i, " = ", i*j, "   ")
  end
  println()
end
```

```
1×1 = 1
1×2 = 2  2×2 = 4
1×3 = 3  2×3 = 6  3×3 = 9
1×4 = 4  2×4 = 8  3×4 = 12  4×4 = 16
1×5 = 5  2×5 = 10  3×5 = 15  4×5 = 20  5×5 = 25
1×6 = 6  2×6 = 12  3×6 = 18  4×6 = 24  5×6 = 30  6×6 = 36
1×7 = 7  2×7 = 14  3×7 = 21  4×7 = 28  5×7 = 35  6×7 = 42  7×7 = 49
1×8 = 8  2×8 = 16  3×8 = 24  4×8 = 32  5×8 = 40  6×8 = 48  7×8 = 56  8×8 = 64
1×9 = 9  2×9 = 18  3×9 = 27  4×9 = 36  5×9 = 45  6×9 = 54  7×9 = 63  8×9 = 72  9×9 = 81
```

这种两重循环可以简写为一个for语句，外层循环先写， 内层循环后写，中间用逗号分隔。 如

In [176]:
```
for i=1:9, j=1:i
  print(j, "×", i, " = ", i*j, "   ")
  j==i && println()
end
```

```
1×1 = 1
1×2 = 2  2×2 = 4
1×3 = 3  2×3 = 6  3×3 = 9
1×4 = 4  2×4 = 8  3×4 = 12  4×4 = 16
1×5 = 5  2×5 = 10  3×5 = 15  4×5 = 20  5×5 = 25
1×6 = 6  2×6 = 12  3×6 = 18  4×6 = 24  5×6 = 30  6×6 = 36
1×7 = 7  2×7 = 14  3×7 = 21  4×7 = 28  5×7 = 35  6×7 = 42  7×7 = 49
1×8 = 8  2×8 = 16  3×8 = 24  4×8 = 32  5×8 = 40  6×8 = 48  7×8 = 56  8×8 = 64
1×9 = 9  2×9 = 18  3×9 = 27  4×9 = 36  5×9 = 45  6×9 = 54  7×9 = 63  8×9 = 72  9×9 = 81
```

矩阵元素遍历

矩阵元素按照行列下标遍历，可以写成两重循环的形式。 Julia的矩阵是按列存储的， 所以循环时先对第一列各个元素循环， 再对第二列各个元素循环， ……，按这样的次序遍历是效率较高的。 如

In [177]:
```
A1 = [1 2 3;
      4 5 6]
for j=1:3, i=1:2
  println("A1[", i, ", ", j, "] = ", A1[i,j])
end
```

```
A1[1, 1] = 1
A1[2, 1] = 4
A1[1, 2] = 2
A1[2, 2] = 5
A1[1, 3] = 3
A1[2, 3] = 6
```

for结构的两重循环中写在前面的是外层循环, 循环变量变化较慢， 写在后面的是内层循环，循环变量变化较快。 上例中列下标j写在前面，行下标i写在后面， 所以关于列的循环j是外层循环，关于行的循环i是内层循环， 这样的矩阵元素遍历方式是按列次序遍历。

矩阵的comprehension

在方括号内用两重的循环变量遍历可以定义矩阵。如

In [178]:
```
Ac = [i*100 + j for i=1:2, j=1:3]
```

Out[178]:
```
2×3 Array{Int64,2}:
 101  102  103
 201  202  203
```

这里写在前面的循环变量i对应于行下标， 写在后面的循环变量i对应于列下标。 执行时行下标i在内层循环，列下标j在外层循环。

这种格式也允许在方括号前写出要求的元素类型，如：

In [179]:
```
Float64[i*100 + j for i=1:2, j=1:3]
```

Out[179]:
```
2×3 Array{Float64,2}:
 101.0  102.0  103.0
 201.0  202.0  203.0
```

如下程序返回矩阵对角化的结果：

In [180]:
```
[(i==j ? Ac[i,i] : 0) for i=1:2, j=1:3]
```

Out[180]:
```
2×3 Array{Int64,2}:
 101    0  0
   0  202  0
```

while循环

for循环适用于对固定的元素或者下标的遍历， 在未预先知道具体循环次数时，需要使用当型循环循环或者直到型循环。

Julia中用 `while cond ... end` 表示当型循环， 在条件cond成立时执行结构内的语句， 直到cond不成立时不再循环。

这种循环一定要使得循环条件是在有限步内会成立的， 否则就产生无限循环，称为死循环，程序将无法终止。

与for循环类似， while循环内的变量也是局部的， 在非自定义函数内while循环访问外部变量时也要用 `global` 关键字声明。 如：

```
s = 1
i = 0
while i < 5
    global i, s
    i += 1
    s *= i
end
println(s)
## 120
```

例：平方根计算

求某个正数x的平方根， 相当于求解方程$f(u)=u^2-x=0$。 利用一阶泰勒展开式$f(u)=f(u_0)+f'(u_0)(u-u_0)+o(u-u_0)$， 其中$f'(u)=2u$，可以得到迭代公式

$$u_n=u_{n-1}-\frac{f(u_{n-1})}{f'(u_{n-1})}=u_{n-1}-\frac{u_{n-1}^2-x}{2u_{n-1}}=\frac{1}{2}\left(u_{n-1}+\frac{x}{u_{n-1}}\right)$$

给定一个初始值u_0， 迭代直到$|u_n-u_{n-1}<\epsilon$为止， ϵ是预先给定的精度如10^{-6}。

程序如下：

In [181]:
```
function mysqrt(x, eps=1E-6)
    u = 1.0
    u1 = 0.0
    while abs(u - u1) >= eps
        u1 = u
        u = 0.5*(u + x/u)
    end
    return u
end
mysqrt(2)
```

Out[181]:
```
1.414213562373095
```

例：随机数查找

设向量x中保存了许多[0,1]上的随机数， 依次查询直到找到第一个大于0.99的为止：

In [182]:
```
n = 1000
x = rand(n)
i = 1
while i <= n && x[i] <= 0.99
    global i
    i += 1
end
if i <= n
  println("i = ", i, " y = ", x[i])
else
  println("Not found!")
end
```

```
i = 10 y = 0.9969702204648303
```

直到型循环与break语句

当型循环每次进入循环之前判断循环条件是否成立， 成立才进入循环。

直到型循环每次先进入循环，在循环末尾判断循环退出条件是否满足， 满足退出条件时就不再循环。 Julia语言没有提供专门的直到型循环语法，可以用如下的方法制作直到型循环：

```
while true
  expr1
  expr2
  ...
  cond && break
end
```

其中 cond 是循环退出条件。 break 语句表示退出一重循环。

例如，用泰勒展开近似计算自然对数$\log(1+x)$：

$$\log(1+x) = x + \sum_{k=2}^{\infty} (-1)^{k-1} \frac{x^k}{k}$$

实际计算时不可能计算无穷次，所以指定一个精度如 eps=0.0001 ，当计算的通项小于此精度时停止计算。 程序用直到型型循环写成：

In [183]:
```
eps = 0.0001
x = 1.0
y = x; xk = x; sgn = 1; k = 1
while true
  global k, sgn, xk, y, eps
  k += 1; sgn *= -1; xk *= x
  item = xk / k
  y += sgn*item
  item < eps && break
end
println("eps = ", eps, " log(1+", x, ") = ", y,
  " Iterations: ", k)
```

```
eps = 0.0001 log(1+1.0) = 0.6931971730609582 Iterations: 10001
```

continue语句

break语句可以在循环内退出当前所在的循环， 去继续执行当前所在循环后面的程序。 continue语句不是退出整个循环， 而是中止当前一轮循环， 进入循环的下一轮。 在for循环和while循环中都可以使用break语句和continue语句。 如：

In [184]:
```
for i = 1:5
    println(i)
    if i==3
        continue
    end
    y = i*i
    println(y)
end
```

```
1
1
2
4
3
4
16
5
25
```

异常处理

只要是程序， 就难以避免会有出错的时候。 为了在程序出错的时候造成严重后果， 传统的办法是对程序增加许多判断， 确保输入和处理是处于合法和正常的状态。 这样的做法过于依赖于程序员的经验， 程序代码也过于繁复。

现代程序设计语言增加了“异常处理”功能。 程序出错时， 称为发生了**异常**(exception)， 编程时可以用专门的程序“捕获”这些异常， 并进行处理。 这并不能保证程序不出错， 而是出错时能得到及时的、不至于造成严重后果的处理。

Julia中捕获异常并处理的基本结构是：

```
try
  可能出错的程序
catch 异常类型变量名
  异常处理程序
end
```

如

```
x = [2, -2, "a"]
for xi in x
    try
        y = sqrt(xi)
        println("√", xi, " = ", y)
    catch ex
        if isa(ex, DomainError)
            println("√", xi, ": 平方根函数定义域异常")
        else
            print("√", xi, ": 平方根函数其它异常")
        end
    end
end
## √2 = 1.4142135623730951
## √-2: 平方根函数定义域异常
## √a: 平方根函数其它异常
```

异常处理代码会降低程序效率， 一些常见错误还是应该使用逻辑判断预先排除。

`throw()` 函数可以当场生成（称为“抛出”）一个异常对象， 如 `throw(DomainError())` 。 Julia中内建了一些异常类型， 详见Julia的手册。

`error(msg)` 可以直接抛出ErrorException对象， 使得程序停止， 并显示给出的 `msg` 字符串的内容。

自定义函数初步

Julia语言支持自定义函数。 在计算机程序语言中， 函数是代码模块化、代码复用的基础。 将常用的计算或者操作写成函数以后， 每次要进行这样的计算或者操作， 只要简单地调用函数即可。 将复杂的任务分解成一个个函数， 可以使得任务流程更加明晰， 任务的不同阶段之间减少互相干扰。

Julia用户需要了解的一点是， Julia利用一个叫做LLVM的即时编译(JIT)系统执行程序， Julia编译器先将程序编译为LLVM中间代码， LLVM再将中间代码转换成本地二进制代码执行， 这使得Julia程序运行效率可以与C++等强类型编译语言相比拟， 但这种做法使得Julia函数在第一次运行时产生额外的时间开销， 第二次运行时就没有额外开销了。 写成函数的Julia代码通常比没有写成函数的代码运行速度快很多， 因为函数的代码在编译时会进行许多优化， 而不写成函数的代码优化较少。

Julia程序中， 用 # 表示本行后面的内容是注释， 不参与执行。 可以用 #= 开始一段注释，用 =# 结束注释。

自定义函数的单行格式

类似于$f(x) = x^2 + 3x + 1$这样的简单函数， 可以用一行代码写成

In [185]:
```
f(x) = x^2 + 3*x + 1
```

Out[185]:
```
f (generic function with 1 method)
```

调用如

In [186]:
```
f(2)
```

Out[186]:
```
11
```

In [187]:
```
f(1.1)
```

Out[187]:
```
5.510000000000001
```

自定义函数的多行格式

需要用多行才能完成的计算或者操作， 就需要写成多行的形式。格式如

```
function funcname(x, y, z)
  ...
end
```

其中 `funcname` 是函数名称， `x, y, z` 等是自变量名， `...` 是函数内的语句或表达式， 定义以 `end` 关键字结尾， `end` 也是许多其它程序块的结尾标志。 函数体内的语句（表达式）一般缩进2到4个空格对齐， 但不像Python那样是必须的， 缩进主要是为了程序的可读性。 函数以最后一个表达式为返回值（结果）， 也可以用 `return` 关键字指定返回值。 不需要返回值的函数可以返回特殊的 `nothing` 值。 `nothing` 是类似R中的 `Null` 的值， 表示不存在。 Julia中用 `missing` 表示缺失值。

例如，写一个函数，以向量的形式输入一个变量的样本值， 计算样本标准差：

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

自定义函数如

In [188]:
```
# mysd: Input numeric vector x, output its sample standard deviation.
function mysd(x)
  n = length(x)
  mx = sum(x) / n
  s = 0.0
  for z in x
    s += (z - mx)^2
  end
  sqrt(s / (n-1))
end
```

Out[188]:
```
mysd (generic function with 1 method)
```

调用如

In [189]:
```
mysd([1, 2, 3, 4, 5])
```

Out[189]:
```
1.5811388300841898
```

事实上，上面的函数定义可以用向量化方法进行简化，如

In [190]:
```
function mysd_simple(x)
  n = length(x)
  mx = sum(x)/n
  sqrt( sum(x .- mx) / (n-1) )
end
```

Out[190]:
```
mysd_simple (generic function with 1 method)
```

第二个版本 `x .- mx` 表示向量 `x` 每个元素与一个标量 `mx` 相减。 这样的做法在Julia中运行效率并不比第一个版本直接循环的效率更好， 甚至可能还不如第一个版本。 Julia语言与R、Matlab等语言不同， 显式的循环一般有更高的执行效率， 向量化写法仅仅是可以使得程序比较简洁。

可选参数(Optional argument)和关键词参数(Keyword arguments)

函数的某些参数可以在定义时指定缺省值， 称为**可选参数**， 在调用时可以省略这些参数。 比如：

In [191]:
```
f_quad(x, a=1, b=0, c=0) = a*x^2 + b*x + c
```

Out[191]:
```
f_quad (generic function with 4 methods)
```

这样，调用 `f_quad(x)` 相当于 `f_quad(x, 1, 0, 0)`，`f_quad(x, 2)` 相当于 `f_quad(x, 2, 0, 0)`，`f_quad(x, 2, 4)` 相当于 `f_quad(x, 2, 4, 0)`。调用时总是假定最后的参数取缺省值而不能令中间的某个参数取缺省值而后续的参数取指定值，所以 `f_quad(x, 2, 4)` 相当于 `f_quad(x, 2, 4, 0)` 而不是 `f_quad(x, , 0, 4)`。不带缺省值的参数和带有缺省值的参数都称为**按位置对应的参数**(positional arguments)，参数（称为虚参）与调用时的值（称为实参）按位置次序对应，实参个数少于虚参个数时，缺少的必须是排在最后可选参数，按缺省值调用。

另外一种带有缺省值的参数称为**关键字参数**，在定义函数时这些参数必须写在分号的后面，调用时必须使用“ 参数名=参数值 ”的格式输入实参值，而不允许按位置对应。关键字参数一般是函数的一些选项，调用函数时仅当需要使用与缺省选择不同的选项时才输入关键字参数的值。例如：

In [192]:
```
function f_quad2(x, a=1, b=0, c=0; descending=true)
    if descending
        return a*x^2 + b*x + c
    else
        return c*x^2 + b*x + a
    end
end
```

Out[192]:
```
f_quad2 (generic function with 4 methods)
```

则调用 `f_quad(x)` 相当于 `f_quad(x, 1, 0, 0)` 即 `x^2`，调用 `f_quad(x, descending=false)` 相当于 `f_quad(x, 0,0,1)` 即 `1`。调用时关键字参数与位置参数之间可以用分号分隔也可以用逗号分隔，比如调用时 `f_quad(x, descending=false)` 和 `f_quad(x; descending=false)` 都可以。

可以定义有多个关键字参数的函数，关键字参数与位置参数之间用分号分隔。

注意可选参数与关键字参数表面相似，但是有本质差别：

- 关键字参数在定义时必须用分号与位置参数分隔；
- 关键字参数在调用时，必须使用“ 变量名=变量值 ”的写法，次序不重要；
- 所有位置参数，不论是有缺省值的还是没有缺省值的，都不允许写成“ 变量名=变量值 ”的写法，只能按位置对应，省略时只能从后向前省略；
- Julia函数允许同一函数名根据不同的参数个数和类型（signature）而进行不同的操作，每一个不同的参数组合称为一个“方法”，这样的特性称为“多重派发”（multiple dispatch），这样的参数组合只考虑位置参数而不考虑关键字参数。

可变个数参数与元组实参

在自定义函数的自变量名后面加上三个句点作为后缀，如 `args...`， 则此函数可以有可变个数的位置参数， `args` 为一个元组。

如：

In [193]:
```
function f_vara1(x, args...)
    println("x=", x)
    println("其它参数：", args)
end
f_vara1(11, 1, 2, 3)
```

```
x=11
其它参数：(1, 2, 3)
```

有时需要传递给一个多自变量函数的实参保存在了一个变量中， 比如， 函数 `max()` 求各个自变量中最大值， 如

In [194]:
```
max(1, 3, 1, 4)
```

Out[194]: 4

如果要求最大值的数已经在一个元组或数组中如何利用 `max()` 求最大值？ 可以用“展开”（scatter）的方法将一个变量中的多个值展开成函数的自变量， 方法是在作为实参的自变量名后面加三个句点后缀，如

In [195]:
```
x = [1, 3, 1, 4]
max(x...)
```

Out[195]: 4

多返回值

函数的最后一个表达式为函数的返回值， 也可以用 `return y` 这样的方法返回值。

如果需要返回多个值， 可以将多个值组成一个元组(tuple)返回， 通过这样的方式就可以返回多个结果。 给元组赋值可以从结果中拆分出多个结果。 如

In [196]:
```
function summ(x)
  xm = sum(x) / length(x)
  xs = sum(x .^2) / length(x)
  return (xm, xs)
end
res1, res2 = summ([1, 2, 3, 4, 5])
println(res1, ", ", res2)
```

```
3.0, 11.0
```

参数传递模式

Julia的参数传递是“共享传递”(pass by sharing)， 而不是按值传递，这样可以省去复制的开销。 如果参数是可变(mutable)的数据类型，如数组， 则函数内修改了这些参数的值，传入的参数也会被修改。

例如：

In [197]:
```
function double!(x)
    for i in eachindex(x)
        x[i] *= 2
    end
end
xx = [1, 2, 3]
double!(xx)
xx
```

Out[197]:
```
3-element Array{Int64,1}:
 2
 4
 6
```

Julia函数的命名习惯是， 如果函数会修改其第一个自变量的值， 将函数名末尾加上一个叹号后缀。

无名函数

Julia的函数也是所谓“第一类对象”（first class objects）， 可以将函数本身绑定在一个变量上， 函数名并非必须， 允许有无名函数。 无名函数在“函数式编程”(functional programming)范式中有重要作用。

无名函数格式是： 参数表 `->` 返回值表达式，其中参数表即自变量表， 没有自变量时参数表写 `()`， 只有一个自变量时可以不写圆括号而只写自变量名， 有多个自变量时将自变量表写成一个元组格式。

比如，函数$f(x) = x^2 + 1$又可以写成

In [198]:
```
x -> x^2 + 1
```

Out[198]:
```
#12 (generic function with 1 method)
```

无名函数的另一种写法如

In [199]:
```
function (x)
    x^2 + 1
end
```

Out[199]:
```
#14 (generic function with 1 method)
```

这样就产生了一个无名函数。

无名函数经常用在 `map()`，`filter()`， `reduce()` 这样的函数式编程函数中。

`map(f, x)` 将函数 `f` 应用到变量 `x` 的每个元素，如：

In [200]:
```
map(x -> x^2 + 1, [1,2,3])
```

Out[200]:
```
3-element Array{Int64,1}:
  2
  5
 10
```

在调用 `map()` 函数时， 如果对每个元素执行的操作需要用多行代码完成， 用无名函数就不太方便。 例如，对数组的每个元素， 负数映射到0， 大于100的数映射到100， 其它数值不变，用有名函数可以写成：

In [201]:
```
function fwins(x)
    if x < 0
        y = 0
    elseif x > 100
        y = 100
    else
        y = x
    end
    return y
end
fwins.([-1, 0, 80, 120])
```

Out[201]:
```
4-element Array{Int64,1}:
   0
   0
  80
 100
```

也可以写成 `map(fwins, [-1, 0, 80, 120])` 。 如果要用无名函数的格式， Julia还提供了map函数的一种do块格式，如

In [202]:
```
map([-1, 0, 80, 120]) do x
    if x < 0
        y = 0
    elseif x > 100
        y = 100
    else
        y = x
    end
    return y
end
```

Out[202]:
```
4-element Array{Int64,1}:
   0
   0
  80
 100
```

注意这样调用 map() 时圆括号中仅有要处理的数据， 要进行的操作写在 do 关键字后面， 操作写成了不带 -> 符号的无名函数格式。

函数 filter(f, x) 中 f 是返回布尔值的函数， x 是向量， filter(f, x) 的结果是将 f 作用在 x 的每个元素上， 输出 f 的结果为真值的那些元素组成的数组。如

In [203]:
```
filter(x -> x>0, [-2, 0, 1,2,3])
```

Out[203]:
```
3-element Array{Int64,1}:
 1
 2
 3
```

reduce(f, x) 中 f 是接受两个自变量的函数， 如加法、乘法，结果是将 x 中的元素用 f 反复按结合律计算结果。 比如， 求 x 的元素和， 除了 sum(x) 函数， 也可以写成 reduce(+, x) 。

map() 、 filter() 、 reduce() 对非数值元素的数组也是适用的。

闭包

可以在函数内定义内嵌函数并以此内嵌函数为函数返回值， 称这样的内嵌函数为闭包（closure）。 闭包的好处是可以保存定义时的局部变量作为一部分， 产生“有状态的函数”， 类似于面向对象语言中的方法， 而Julia并不支持传统的面向对象语言中那样的类。

例如， 某个函数希望能记住被调用的次数。 传统的方法无法解决问题，比如下面的版本是无效的：

In [204]:
```
function counter_old()
    n = 0
    n = n+1
    return n
end
println(counter_old(), ", ", counter_old())
```

```
1, 1
```

可以用如下的闭包做法：

In [205]:
```
function make_counter()
    n = 0
    function counter()
        n += 1
        return n
    end
end
my_counter = make_counter()
typeof(my_counter)
```

Out[205]:
```
var"#counter#22"
```

In [206]:
```
println(my_counter(), ", ", my_counter())
```

```
1, 2
```

递归调用

函数定义时允许调用自己， 这使得许多本质上是递推的计算程序变得很简单， 比如，$n! = n(n-1)!$， 用递归函数可以写成：

In [207]:
```
function myfact(n)
    if n==1
        return 1
    else
        return n*myfact(n-1)
    end
end
myfact(5)
```

Out[207]: 120

再比如Fibonacci序列， $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$， 前几个数是0, 1, 1, 2, 3, 5, 8, 13, 21, 34。 用递归调用写成

In [208]:
```
function myfib(n)
    if n==0
        return 0
    elseif n==1
        return 1
    else
        return myfib(n-1) + myfib(n-2)
    end
end
for n in 0:9
    println(n, ": ", myfib(n))
end
```

```
0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
8: 21
9: 34
```

但是，这样的递归在反复调用自身以后效率会很低， 因为对每个n都要多次调用自变量为$0, 1, 2, \ldots$的情形。 可以用闭包的方法将已有结果保存， 使得计算效率大大提高：

In [209]:
```
function makefib()
    saved = Dict(0=>0, 1=>1)
    function fib(n)
        if !(n in keys(saved))
            saved[n] = fib(n-1) + fib(n-2)
        end
        return saved[n]
    end
end
myfibnew = makefib()
for n in 0:9
    println(n, ": ", myfibnew(n))
end
```

```
0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
8: 21
9: 34
```

这种方法也可以用来解决判断素数的问题。 可以用字典类型保存一个已经确认的素数表， 然后为了判断某个数是否素数， 先在已保存素数表中查找， 查找不到再用除法判断。 之所以用字典类型而不是数组类型保存素数表， 是因为数组类型的查找是线性查找而字典类型则为杂凑表查找， 效率更高。

模块

前面的例子都是相当于在命令行直接运行的。 这样，变量和函数属于命令行对应的名字空间， 称为Main模块。 如果要写比较长的程序， 所有变量和函数都在同一个名字空间中就很容易发生名字冲突。

Julia使用模块来区分名字空间， 不同模块的同名变量、函数没有关系， 不会发生冲突。

在一个模块内， 可以有模块本身的全局变量， 不同模块的全局变量即使同名也没有关系。

在模块内，可以控制其它模块的哪些名字是通过导入变得可见的， 还可以规定本模块的哪些名字是通过导出变得可以被其它模块访问。

一个简单的模块定义如：

In [210]:
```
module MyStat

export mean, rmse

function mean(x)
  sum(x) / length(x)
end

function rmse(x)
  sqrt(sum(x .^2) / length(x))
end

end
```

Out[210]:
```
Main.MyStat
```

调用一个模块中的全局变量或者函数时， 需要将模块用 using 或者 import 关键字引入到当前的名字空间中。

using MyStat 使得当前模块可以直接使用 MyStat 模块中用 export 声明过的函数，如

```
using .MyStat
rmse([1,2,3,4])
```

这里的 .MyStat 是表示找到 MyStat 模块定义就在当前环境中。 using 整个模块是比较不安全的做法， 这样会引入多个用户自己不一定了解的函数和全局变量进入当前模块， 建议慎用。

比较安全的方法是用 using 指定将模块中的哪些函数名导入到当前名字空间，如

```
using .MyStat: mean, rmse
rmse([1,2,3,4])
```

注意上面的冒号要紧跟着 .MyStat ，不能有空格隔开。

可以用 import 声明导入单个的函数。如

```
import .MyStat.mean, .MyStat.rmse
rmse([1,2,3,4])
```

也可以用 import 仅导入模块名， 其中的函数需要用“模块名.函数名”的格式调用。如

```
import .MyStat
MyStat.rmse([1,2,3,4])
```

用 import 导入的函数都可以定义新方法， 但是如果仅导入了模块名， 则定义新方法时函数名还要用“模块名.函数名”的格式表示。

使用 using 导入的单个函数不能添加新的方法， 用 using 导入整个模块则可以为导入的的函数添加新的方法。

当一个模块的全局变量通过 using 或者 import 导入到当前模块后， 当前模块不允许存在同名的全局变量， 而且也只允许读取其它模块中的全局变量值而不允许对其进行修改。

一个模块可以存放在一个单独文件中； 一个文件也可以同时包含多个模块定义； 一个模块的多段代码也可以分别存放在不同文件中然后用 include() 函数载入到模块定义中。 模块定义一般存在于Julia扩展包中， 安装扩展包后，不需要用 .MyStat 这样的相对路径， 只需要写 import MyStat 这样的绝对路径就可以了。 .MyStat 中的 . 表示在当前名字空间中查找 MyStat 模块定义。

为了调用MyStat模块， 设源文件保存在 `mystat.jl` 中， 可以先 `include("mystat.jl")` 然后用 `MyStat.mean()` 的格式调用函数， 也可以用

作用域

变量的作用域是某个变量可见的范围。 同名的变量使得问题变得复杂， 变量作用域使得同名的不同变量能够区分开来。

变量作用域都是某些程序结构的范围内， 比如一个函数定义范围，而不是任意的一段程序行的范围。

有两种主要的作用域：

- 全局作用域
- 局部作用域，可以嵌套
  - 软局部作用域
  - 硬局部作用域

全局作用域适用于模块(module)内，baremodule内， 或者在命令行环境内。 每个模块有自己的全局变量， 命令行运行的程序相当于main模块。

for, while, 自省(comprehensions)，try-catch-finally, let结构内部构成软局部作用域。 而begin复合语句，if结构，不构成局部作用域。

硬局部作用域是函数定义，struct, macro中。

句法作用域

函数可以读取其外围环境中的变量的值。 这里“外围环境”的定义Julia规定为句法作用域(lexical scoping), 即一个自定义函数的作用域的外围环境是**定义**此函数的环境， 而**不是运行时**调用这个函数的环境。

例如：

In [211]:
```
module Amod
  x = 1
  foo() = x
end
import .Amod
x = -1
println(Amod.foo())
```

```
1
```

最后一个语句调用 `Amod.foo()` 时用到的是模块Amod的全局变量 `x`， 这是定义 `foo()` 时包含函数定义的代码中的 `x` 变量。 赋值为-1的变量 `x` 是命令行的全局变量， 这是调用 `foo()` 时所处环境中的变量。

要注意的是， 函数使用所处环境中的变量的当前值， 这不一定是定义该函数时的变量值。如

In [212]:
```
module Amod2
  x = 1
  foo() = x
  function setx!(xnew)
    global x = xnew
    return ()
  end
end
import .Amod2
Amod2.setx!(2)
x = -1
println(Amod2.foo())
```

```
2
```

以上程序中调用 `Amod2.foo()` 函数时， 其依赖的变量 `Amod.x` 的值已经被修改为2了， 所以 `Amod2.foo()` 返回值不是1而是2。

一个模块内的所有代码可以区分为不同的作用域。

- 不在任何函数内的变量是全局变量。
- 函数的自变量和函数内用local声明的变量是属于函数的自变量。
- `for`，`while`，`let` 环境内用local声明的变量是属于环境本身局部作用域的局部变量。

全局作用域

在命令行环境中定义的变量属于命令行环境的全局作用域，实际是Main模块的全局作用域。

每一个模块有自己的全局作用域，但是没有一个统一的全局作用域。 模块内在所有函数定义外部赋值的变量为全局变量。 模块内的任何位置用 `global` 关键字声明的变量为该模块的全局变量。

为了访问其它模块的全局变量，可以用using或者import引入， 也可以用“模块名.变量名”格式。 事实上，每个模块是一个名字空间。 只有同一模块的代码可以修改本模块的全局变量， 在模块外可以读取模块内全局变量的值但是不允许直接对模块内的全局变量赋值修改， 可以通过调用同一模块的函数间接地修改模块内的全局变量。

使用全局变量容易造成程序漏洞， 尤其是修改全局变量的值会造成难以察觉的错误。 在为全局变量赋值时用 `const` 前缀声明其为常数全局变量， 这样的全局变量不能重新绑定， 但如果其中保存mutable值的话还是可以修改保存的值的。 这种做法可以避免使用全局变量的一些错误以及性能缺陷。如

```
const GC = 9.8
```

局部作用域

许多代码块结构都会产生一个局部作用域，如函数定义、for循环等。 局部作用域都可以读写其定义环境所处作用域（称为父作用域）的变量， 但有例外：

- 如果局部作用域对变量的赋值会修改**全局**变量， 这时如果不在局部作用域中用global声明该变量， 程序出错；
- 如果在局部作用域中用local声明了变量， 则对该变量的修改不会影响父作用域的变量。

局部作用域不是名字空间， 所以内层可以访问外层变量， 但是外层无论如何不能访问内层作用域的局部变量。

按照对父作用域的变量如何继承来区分， 局部作用域分成硬局部作用域和软局部作用域。

下面举例说明这些作用域规则。 假设每个例子都是在重新启动REPL的命令行环境内执行， 这样没有其它全局变量的干扰。

例1

下面的例子说明， 局部作用域独有的变量在父作用域内无法访问。 例子中 `for` 循环内构成了一个局部作用域， 其中独有的变量 `z` 无法在其父作用域即 `f1()` 函数局部作用域内访问。

In [213]:
```
function f1()
  for i=1:5
    z = i
  end
  println(z) # 错误：z无定义
end
```

Out[213]:
```
f1 (generic function with 1 method)
```

如果调用 `f1()` ，会出错：

```
UndefVarError: z not defined
```

即在 `for` 循环外部变量 `z` 无定义。

例2

在下面的程序中， `f2()` 函数内构成一个局部作用域。 `f2()` 内定义的变量 `z` 不能在外部访问。

In [214]:
```
function f2()
  z = 1
  println("Inside f2(): z=$z")
end
```

Out[214]:
```
f2 (generic function with 1 method)
```

如果执行 `f2()` ，结果为

```
Inside f2(): z=1
```

如果执行 `println("Outside f2(): z=$z")` ，结果为

```
UndefVarError: z not defined
```

例3

下面的例子修改了父作用域中的变量， 变量不是全局变量也没有用 `local` 声明。 函数 `f3()` 中的 `for` 循环构成一个局部作用域， 其父作用域是 `f3()` 的局部作用域， `for()` 循环中可以直接读取并修改父作用域中非全局的变量 `z` 的值：

In [215]:
```
function f3()
  z = 0
  for i=1:5
    z += i
  end
  println(z) # 15
end
f3()
```

```
15
```

注意 for 作用域中 z 不是全局变量也没有用 local 声明。

例4

在 f3() 定义中，如果在 for 结构内用 local 声明变量 z ， 则程序会出错。

下面的例子说明 f4() 中的局部变量 z 与其中的 for 结构中的 z 是两个不同的变量， 因为在 for 结构中用 local 声明了该结构中的 z 是局部版本：

In [216]:
```
function f4()
  z = 0
  for i=1:3
    local z
    z = i
    println("i=$i z=$z")
  end
  println("Outside for loop: z=$z") # 0
end
f4()
```

```
i=1 z=1
i=2 z=2
i=3 z=3
Outside for loop: z=0
```

例5

下面的例子说明，如果局部作用域内的赋值会修改全局变量的值， 在没有在局部作用域内用 `global` 声明该变量的情况下程序出错：

```
z = 0
for i=1:5
  z += i
end
```

程序结果为：

```
ERROR: UndefVarError: z not defined
```

在Jupyter中运行时可能会因为修改了规则而不报错。

这个问题的正确做法是将程序像 `f3()` 那样写在一个函数中， 这就不会发生在局部作用域内修改全局变量的问题。 在局部作用域用 `global` 声明要修改的全局变量也可以：

In [217]:
```
z = 0
for i=1:5
  global z
  z += i
end
println(z) # 15
```

```
15
```

使用 `global` 关键字要慎重， 一旦在局部作用域中将某个变量用 `global` 声明为全局变量， 则此变量就不仅仅可以被其父作用域访问， 而是在模块内全局可访问。 如

In [218]:
```
function f5()
  for i=1:5
    global z=i
  end
  println("Outside for structure: z=$z")
end
f5()
println("Outside function f5(): z=$z")
```

```
Outside for structure: z=5
Outside function f5(): z=5
```

软局部作用域

软局部作用域内的变量默认是在其父作用域内的变量， 但是:

- 在软局部作用域内新定义的变量，作用域外仍不能访问;
- 软局部作用域内用local声明过的变量仅能在该作用域内访问而不会与其父作用域的同名变量冲突;
- 软局部作用域内试图为全局变量赋值而未在作用域内用 `global` 声明该变量会出错。

for循环，while循环，自省结构， try-catch-finally结构，let结构会引入软局部作用域。 软局部作用域，如for循环， 一般用来处理其父作用域（一般是函数内部）内的变量， 与其周围代码是不可分割的。 比如在用for循环做累加时， 累加结果变量一定是在for循环父作用域内而不能是局部的, 所以软作用域非以上三种特殊情况下可以读写其父作用域的变量。

for循环和自省结构中的循环变量都是结构内的局部变量， 即使在其父作用域中有同名变量也是如此， while循环没有语法上的循环变量所以不受此限制。 在结构中新定义的变量都是结构内的局部变量， 但如果父作用域是函数的局部作用域且父作用域内有同名变量， 则结构内的变量读写父作用域中的变量。

软局部作用域的这些规定与例外规定与其它程序语言存在较大差别， 初学者很容易出错。建议如下：

- 对软局部作用域，尽量不要在其中新定义变量;
- 如果新定义变量， 用local声明使其作用域变得明显可见就不会发生误读误判， 即使父作用域中没有同名变量也加上这个声明可以使得程序的意图更清楚;
- 对于函数内的软局部作用域，其父作用域是函数的局部作用域， 为了能够访问函数的局部变量， 软局部作用域内不需要也不应该使用global声明该变量， 因为其变量除了新定义的，都是函数的自变量和局部变量， 使用global声明的副作用是该变量成为全局变量， 而不仅仅是父作用域中可访问的变量。

硬局部作用域

函数定义，struct结构，宏定义内部为硬局部作用域。 其中函数定义可以嵌套在另一个函数定义中， 而struct结构和宏定义则只能在全局作用域中定义。

在硬局部作用域中，几乎所有变量都是从其父作用域内继承来的， 例外情况包括：

- 对父作用域内的变量赋值，会修改全局变量时， 这时赋值会产生一个局部副本而不修改全局变量值。 注意软局部作用域没有用 `global` 声明而修改全局变量会出错， 而不是默默地生成一个局部副本。
- 用local声明的变量，仅在此局部作用域内起作用， 即使父作用域中有同名变量或者有同名的全局变量。

在硬局部作用域中， 父作用域中的变量以及全局变量都可以不经声明直接读取值； 父作用域中的变量如果不是全局变量， 可以直接修改变量值。

硬局部作用域中不经声明不能修改全局变量值。 需要在局部作用域内用global声明该变量才能修改全局变量值。 为了能明显地反映程序意图， 在局部作用域内不论读或者写访问全局变量时， 都最好在局部作用域内用 `global` 关键字声明该变量。

例6

例如， 在函数内部读取全局变量的值， 可以不用 `global` 声明：

In [219]:
```
z = -1
function f6()
  println("函数内读取外部变量：z=$z")
end
f6()
```

```
函数内读取外部变量：z=-1
```

但是， 在函数内没有用 `global` 声明的全局变量， 不能修改， 意图修改全局变量的代码实际是建立了一个局部变量：

In [220]:
```
z = -1
function f7()
  z = 1
  println("函数内不用global声明修改全局变量，修改后：z=$z")
end
f7()
println("退出函数后全局变量：z=$z")
```

```
函数内不用global声明修改全局变量，修改后：z=1
退出函数后全局变量：z=-1
```

这说明函数 f7() 内并没有修改全局变量 z 的值， f7() 运行期间显示的 z 是一个局部变量。 函数内并没有能够修改外部的 z 变量值， 而且一旦函数内给 z 赋值，整个函数体内 z 都是局部变量， 这样在给 z 赋值之前 z 是无定义的， 而不是能访问外部的 z 值。 下面的程序在 z=1 赋值之前显示 z 的值， 这时的 z 已经是局部变量，所以程序会出错：

```
z = -1
function f7()
  println("函数内不用global声明修改全局变量，修改前：z=$z")
  z = 1
  println("函数内不用global声明修改全局变量，修改后：z=$z")
end
f7()
println("退出函数后全局变量：z=$z")
```

所以，函数内赋值的变量， 最好用local声明以避免误解。 嵌套定义的函数是一个例外， 嵌套定义的函数的父作用域是另一个函数的局部作用域， 不存在修改全局变量的问题。

在函数内用 global 声明变量， 就可以读写访问全局变量。 为了程序意图更清晰， 即使仅读取全局变量， 最好也用 global 声明。 如

In [221]:
```
z = -1
function f8()
  global z
  z = 1
  println("函数内用global声明修改全局变量：z=$z")
end
f8()
println("退出函数后全局变量：z=$z")
```

```
函数内用global声明修改全局变量：z=1
退出函数后全局变量：z=1
```

上述程序中所有的变量 `z` 都是全局变量 `z` 。

嵌套定义函数的作用域

嵌套地定义在函数内的函数， 其作用域与直接定义在全局作用域的函数不同：

- 直接定义在全局作用域的函数的父作用域是全局作用域， 父作用域的变量是全局变量， 按照规定， 函数内修改没有用 `global` 声明的变量只能生成一个同名局部变量；
- 嵌套地定义的函数， 其父作用域是另一个函数的局部作用域， 所以在嵌套定义内可以不需要声明直接读写父作用域中的变量， 实际上也不能用 `global` 声明父作用域中的变量。
- 对于struct结构和宏定义， 它们只能在全局作用域定义而不能嵌套定义， 所以不存在这个差别。

例7

In [222]:
```
x = "global.x" # 全局变量
function foo()
  local x = "foo.x" # 这是一个baz作用域内的局部变量
  function bar()
    println("在bar()开始时: x=$x") # baz.x, 读取父作用域的局部变量
    x = "bar.x" # 内嵌函数内, 允许读写访问其所在函数的局部变量
    println("在bar()修改后: x=$x") # bar.x, 读取父作用域的局部变量
  end
  bar()
  println("在bar()结束后: x=$x") # bar.x, 函数baz()的局部变量x被内嵌函数修改了
end
foo()
println("在foo()结束后: x=$x") # global.x, 内嵌函数没有修改全局变量
```

```
在bar()开始时: x=foo.x
在bar()修改后: x=bar.x
在bar()结束后: x=bar.x
在foo()结束后: x=global.x
```

可以看到， `baz()` 和 `bar()` 内的 `x` 都与全局的 `x="globa.x"` 没有关系。 嵌套定义的 `bar()` 内能读取其父作用域 `baz` 作用域的 `x` 的值， 也修改了其父作用域中的 `x` 的值， 这一点与非内嵌函数修改全局变量不同。

例8

之所以规定嵌套函数可以直接读写访问其父作用域中的变量， 是实现所谓“闭包”(closure)的要求。 闭包是带有状态的函数， 因为Julia不支持如Java、C++、Python这些语言的“类”(class)， 所以需要利用闭包来实现记忆状态的函数。 在这一点上Julia语言与R语言比较相近。

在下面的例子中， `f9()` 是一个局部作用域， 在 `f9()` 内嵌套地定义了无名函数并且将其作为函数 `f9()` 的返回值, 这时 `f9()` 的函数值是一个函数对象。 通过将 `f9()` 的函数值赋值给变量 `counter` , 变量 `counter` 就变成了一个函数， 按照句法作用域规则， `counter()` 函数可以完全读写访问其定义时的父作用域 `f9()` 中的变量 `state` , `state` 就变成了 `counter()` 函数的私有状态变量， 可以保存上次运行状态：

In [223]:
```
function f9()
  state = 0
  function ()
    state += 1
  end
end
counter = f9()
println(counter(), ", ", counter())
```

```
1, 2
```

硬作用域和软作用域的比较

软局部作用域，如for循环， 一般用来处理其父作用域（一般是函数局部作用域）内的变量， 与其周围代码是不可分割的。 比如在用for循环做累加时，累加结果变量一定是在for循环父作用域内而不能是局部的, 所以软作用域可以读写其父作用域的变量。

另一方面，硬作用域一般是独立地运行在不同的调用场合的， 与周围的代码的关系没有那么紧密，很可能会在别的模块中被调用。 所以硬作用域不允许修改全局变量值（除非使用global声明）。

数据类型

程序中的常量和变量都有类型，比如，常数108的类型为Int64。函数 `typeof()` 可以返回常量或变量的类型，如

In [224]:
```
typeof(108)
```

Out[224]:
```
Int64
```

变量的类型由其中保存的值的类型决定，Julia变量实际是“绑定”到了某个保存了值的地址。如

```
In [225]: x = 108; typeof(x)
```

```
Out[225]: Int64
```

函数 `typemax()` 可以求一个数值类型能保存的最大值， `typemin()` 求能保存的最小值。如

```
In [226]: show([typemin(Int8), typemax(Int8)])
```

```
Int8[-128, 127]
```

Julia的变量不必须声明类型， 但是必须初始化，未定义的变量用于计算会出错。 对许多用户来说， 不需要知道类型声明就可以完成大部分常见任务。

Julia是动态类型的。 动态类型的语言中的变量只有在运行时才能确定类型， 而静态类型的语言在编译阶段就确定了类型。

在Julia中，某个变量x一开始绑定了一个整数值， 在后续执行中可以重新绑定一个字符串值。 当然，这样的做法是不可取的。

Julia归类到动态类型语言， 是因为不必须声明变量类型。 其它的动态类型语言如Python不支持声明变量类型， Julia是允许不声明变量类型， 也可以声明变量类型， 声明变量类型往往可以改善程序效率， 尤其是函数的自变量类型。 同名的函数可以因为其自变量类型声明的不同而执行不同的操作， 这称为多重分派(multiple dispatch)。

在循环内部声明变量类型可以避免因为每次查询变量类型而引起的运行效率损失。

类型系统

用程序语言术语描述， Julia语言的类型系统是动态的，主格的(nominative), 参数化的。 通用类型可以带有类型参数。 类型的层次结构是显式声明的， 不是由兼容结构隐含的。 实体类型(concrete types)不能互相继承， 这与很多面向对象系统不同。 Julia的理念认为继承数据结构并不重要，继承行为才是最有用的。

Julia类型系统的其它特点：

- 对象值与非对象值没有明确的区别。 实际上，Julia中所有值都是对象，都有一个类型。 类型有一个唯一的类型系统， 所有类型都属于一个完全连接的图， 图中所有节点都是第一类对象可以采用的类型。
- 没有有意义的“编译时类型”。 仅在程序运行时一个值才有真正的类型。 在面向对象语言中这称为“运行时类型”。
- 只有值才有类型，变量本身没有类型， 变量只是绑定到值上的一个名称。 当然，对一般用户而言， 声明过类型的变量也可以认为是有类型的。
- 抽象类型和实类型都可以用其它类型参数化， 如果不需要引用参数或者限制参数取值时可以不写类型参数。

Julia的所有类型都可以表示在一个类型树中， 类型分为抽象类型与实体类型， 每个类型有且仅有一个父类型， 最高层的父类型是Any类型。 每个类型都属于Any类型。 实体类型不允许有子类， 一个类型不允许有多个直接的父类。

抽象类型仅用来作为其他类型的父类， 没有取类型的值。 例如，Julia的数值类型可以由如下的类型结构组成：

```
abstract type Number end
abstract type Real     <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer  <: Real end
abstract type Signed   <: Integer end
abstract type Unsigned <: Integer end
```

所有的数值都是Number， Number又分为实数(Real)和其它数， 实数包括Integer和AbstractFloat， Integer中有Signed和Unsigned， AbstractFloat中有Float16, Float32, Float64等， Signed中有Int8, Int16, Int32, Int64, Int128等， Unsigned中有UInt8, UInt16, UInt32, UInt64, UInt128等。 布尔类型Bool是Integer的子类。

程序中如1.2和1.2e-3这样的字面浮点数的类型是Float64， 单精度数(Float32)可以写成如1.2f0, 1.2f-3这样的格式。

BigInt是任意精度整数， BigFloat是任意精度浮点数。 实际上，BigFloat有一个用户可修改的的有效位数设置。

用 <: 运算符来判断子类关系是否成立，如

In [227]:
```
Int32 <: Number
```

Out[227]: true

In [228]:
```
Int32 <: AbstractFloat
```

Out[228]: false

复数类型

Julia内建了复数类型， 这是Number的子类。 关键字im用来表示复数虚部， 比如 `1.0 + 2.1im` 表示复数$1 + 2.1i$。

复数类型用实数类型分别保存实部和虚部， 其定义利用了Julia语言的“参数化类型”： 实部与虚部的实际类型可以是一个“类型参数”。 比如， `Complex{Float64}` 用Float64(即双精度实数)分别保存实部和虚部。

对复数类型可以应用函数 `abs()` , `exp()` , `sqrt()` , `real()` , `imag()` 等。 如

In [229]:
```
abs(1.0 + 2.1im)
```

Out[229]: 2.3259406699226015

In [230]:
```
sqrt(1.0^2 + 2.1^2)
```

Out[230]: 2.3259406699226015

有理数类型

Julia内建了有理数类型， 分别保存分子和分母， `Rational{Int64}` 用Int64类型分别保存分子和分母。 有理数是Number的子类， 支持四则运算， 也可以与其他数值类型混合进行四则运算。

用 `2 // 5` 表示有理数$\frac{2}{5}$， 其数据类型为 `Rational{Int64}` 。 可以用函数 `num()` 取出分子， 用 `denom()` 取出分母， 用 `float()` 转换为浮点实数。

函数内部以及某些控制结构如 `for` 循环结构内， 定义的变量是局部的， 全局变量在局部作用域内的访问有特殊规则。 函数内部可以嵌套定义函数， 嵌套在内部的函数的局部变量又有特殊的作用域规则。

类型转换与提升

设 `T` 为类型名， `T(x)` 在可以无损转换的情况下将自变量 `x` 转换为 `T` 类型返回， 但是不能无损转换时报错。如

In [231]:
```
Int64(1.0)
```

Out[231]:
```
1
```

In [232]:
```
Int64(1.5)
```

```
InexactError: Int64(1.5)

Stacktrace:
 [1] Int64(::Float64) at .\float.jl:710
 [2] top-level scope at In[232]:1
```

In [233]:
```
Float64(1)
```

Out[233]:
```
1.0
```

In [234]:
```
Int64(true)
```

Out[234]:
```
1
```

In [235]:
```
Bool(1)
```

Out[235]:
```
true
```

`string()` 可以将大多数类型转换为字符型， 也可以连接多项，如

```
In [236]: string(1+2)
```

```
Out[236]: "3"
```

```
In [237]: string("1+2=", 1+2)
```

```
Out[237]: "1+2=3"
```

反过来，为了将字符串中的数字转换成数值型， 用 `parse()` 函数， 如

```
In [238]: parse(Float64, "1.23") + 10
```

```
Out[238]: 11.23
```

通过多重派发定义各个运算符， 使得Julia的四则运算、比较运算等表达式中可以混合使用布尔型、整型、浮点型数， 参与运算的数据类型自动提升到更一般的数据型。 如

```
In [239]: 1 + 2*1.5 - false
```

```
Out[239]: 4.0
```

类型声明

Julia用 `::` 表示类型归属， 此运算符有两个含义：类型验证(assertion)与类型声明。

类型验证

在变量名和表达式末尾添加 `::` 然后可以要求类型验证。 如 `x::Float64` ， `(x+1)::Int64` 。 这样做的理由是

1. 作为一个额外的验证确保程序是按照预想的逻辑执行的；
2. 给编译器提供类型信息以改善程序效率。

::符号的意思是“is instance of”(是某某类的一个实例)， 即左边的表达式是右边的类的一个实例 如果类型是实类型，则左边必须确实是此类型； 如果类型是抽象类型，则左边应该是其子类。 类型声明不成立的时候会发生异常，否则结果是左边的值。

例如，

```
(1+2)::AbstractFloat
```

结果出错

```
ERROR: TypeError: in typeassert, expected AbstractFloat, got Int64
```

而

```
(1+2)::Int
```

类型验证无误，则验证不起作用， 结果为表达式的值 `3` 。

类型声明

当带有类型声明的变量被赋值时或用local声明时， 此变量被强制规定成此类型， 等号右边的值被转换成此类型再赋值。 这样的做法避免了类型被无意间改变而使得程序效率损失， 能够给编译器提供额外信息以产生高效可执行代码。 注意这种声明会作用到整个当前作用域，包括声明之前的程序。 在全局作用域如命令行还不支持这样的声明。 函数自变量也可以用如此方法声明， 这样的声明可以为一个函数针对不同输入类型规定不同的处理方法， 称为“多重派发”(multiple dispatch)。

局部变量类型声明例如

In [240]:
```
function foo()
  local x::Int8;
  x = 100;
  x
end
typeof(foo())
```

Out[240]:
```
Int8
```

函数的返回值也可以声明返回值类型，如 上例中函数总是返回Float64类型，即使是return 1的结果也会转换成Float64类型

In [241]:
```
function sqp(x)::Float64
    return sqrt(x+1)
end
[typeof(sqp(0)), typeof(sqp(1))]
```

Out[241]:
```
2-element Array{DataType,1}:
 Float64
 Float64
```

通过对函数返回类型的声明， 不论结果是1还是$\sqrt{2}$， 结果类型都是Float64。

抽象类型

抽象类型不能实例化， 只是作为实类型的父类， 好处是可以使得函数自变量类型取一类类型。 比如将函数自变量声明为Number， 则输入的值为Int8, Float64等都是允许的。 详见Julia手册。

初等类型

初等类型就是基础的二进制表示，如整数，浮点数。 Julia允许自定义初等类型，用primitive type命令。 Julia提供的初等类型也是用Julia定义的。 详见Julia手册。

复合类型

复合类型就是其它编程语言中的记录，结构，对象等。 复合类型是若干个有命名的域的集合， 此种类型的实例可以看成一个值。 复合类型是最常用的用户自定义类型。

很多语言中复合类型是与匹配的函数耦合在一起的，称为对象(objects)， 比如Java、C++、Python等。 在Julia中，所有的值都是对象，都是某个类的实例，但是对象没有耦合在一起的函数。 Julia的做法是调用函数时根据参数的类型不同而选择不同的操作方式， 称为多重派发(multiple dispatch)。 多重派发和一般的对象系统差别在于函数的操作选择依赖于所有参数的类型， 而不仅仅是第一个参数的类型， 但是函数一般没有对应的状态（属性）。

用struct定义的复合类型是不可修改的(immutable)，这样的好处是

- 更高效。有时可以高效地包装进数组中，有时甚至不需为其分配额外存储空间。
- 不能修改构造器提供的值。
- 不能修改的内容，也使得程序比较简单。

如果某个域是可修改类型(mutable)，如数组， 则该域保存的内容还是可以修改的， 这里不可修改是指这样的域不能再绑定到别的对象上。

用mutable struct定义可修改复合类型(Mutable composite types)， 其域的值可以修改。 这样的数据类型一般在堆上分配内存，有稳定的内存地址。 这是一个容器，不同运行时刻的容器内容可以变化， 但是其地址不变。 另一方面，不可变的类型是与各个域的值紧密联系的， 域的值就决定了对象的一切。

在赋值和函数参数传递时不可变类型复制传递， 可变类型按引用传递； 不可变复合类型的域不可修改 （不能绑定到其它值，但是如果域本身是可变类型还是可以修改内容的）。 典型的可变类型是数组， 而元组(tuple)是不可变类型。

新的复合类型定义以及实例化方法详见Julia手册。

参数化类型

参数化类型是类似于C++中模板(template)的类型。 一个类型可以将其元素的值类型作为类型参数， 一次性地定义一批类型。 参数化类型使得同一算法可以用来处理不同类型的数据。

数组(Array)就是参数化类型。 `Vector{Float64}` 或 `Array{Float64,1}` 就是元素为Float64的一维数组， 而 `Vector{String}` 或 `Array{String,1}` 就是字符串的一维数组。 `Array{Float64,2}` 是元素为Float64的二维数组， 也称为矩阵。 这里 `Float64` ， `String` 就是参数化类型的类型参数。

参数化类型涉及到Julia函数的多重派发应用． 技术比较复杂． 详见Julia手册。

方法

Julia的函数可以通过给自变量声明类型实现高效代码， 同时也可以使得程序意图更明显。 同一个函数可以有不同类型的自变量， 这其实是多个函数共用同一个函数名， 称这些函数为该函数名的“方法”（methods)。 用类似于C++模板的方法可以将函数自变量类型参数化， 这样同一算法可以适用于多个自变量类型。

这部分技术比较复杂， 不恰当地应用可以造成程序错误， 初学者可以暂时忽略， 自定义函数时先不声明自变量类型。 等发现某个函数造成了运行效率瓶颈， 或者Julia语言已经运用比较纯熟时， 再改进原来的函数， 使其对具体的不同自变量类型有不同的实现。

多重派发

函数可以将0到多个自变量（参数）集合映射为一个返回值， 类似的操作对于不同的参数类型，可能有不同的具体实现。 比如，两个整数相加，两个浮点数相加，一个整数加一个浮点数，是类似的运算， 但实际计算过程很不一样。 尽管如此，因为其概念上的相近似，Julia中将所有这些运算都归结为+函数。

对于这种类似的操作， Julia在定义函数时， 可以不是一下就完整定义， 而是针对不同参数类型逐步地完善函数定义。 函数针对一种参数类型组合所规定的操作叫做一个**方法**。 借助于参数类型的声明(annotation)， 以及参数个数， 可以对同一函数定义多个方法， 每一种参数类型组合定义一个方法。 调用一个函数时， 规定类型最详细、与输入的参数类型最匹配的方法被调用。 设计时考虑比较周到的话，虽然一个函数有多种特殊的处理方法， 其结果看起来还是可以比较一致的。

从一个函数的多种方法选择一个方法来执行的过程称为派发(dispatch)。 Julia在派发问题上与其它的面向对象语言由很大区别， Julia是根据参数个数不同以及所有参数类型的不同来选择方法的， 这称为多重派发(multiple dispatch)， 其它面向对象语言一般仅根据第一个参数派发， 而且第一个参数往往不写出来。 Julia的做法更合理，更灵活、更强大。

对于常见的数学计算， 一般用Float64来计算。 可以先定义Float64版本的函数， 然后对于一般的自变量， 可声明为 `Number` ， 转换为 `Float64` 后调用已有函数即可。

例如：

In [242]:
```
ff(x::Float64, y::Float64) = 2x + y
ff(x::Number, y::Number) = ff(Float64(x), Float64(y))
println(ff(1.0, 2.0))
println(ff(1, 2))
println(ff(1.0, 2))
```

```
4.0
4.0
4.0
```

上述函数如果还希望输入的 `x` 和 `y` 都是整数时返回整数值，只要再定义一个整数输入的方法：

In [243]:
```
ff(x::Integer, y::Integer) = 2*Int64(x) + Int64(y)
ff(1,2)
```

Out[243]:
```
4
```

注意，为了使得上述函数可以对向量使用， 不需要单独定义方法， 而只要用加点语法即可，如

In [244]:
```
ff.([1.0, 2.2], [-1.0, 3.3])
```

Out[244]:
```
2-element Array{Float64,1}:
 1.0
 7.7
```