symPy教程

（翻译至<http://docs.sympy.org/latest/tutorial/index.html>

20180725）

（非常欢迎大家改错、并提出宝贵意见。译者四级都没过，如有错误请见谅。）

# 导论

本教程假设读者已经知道了Python编程基础。如果你没有学过Python，[Python官方教程](https://docs.python.org/3/tutorial/index.html)是很好地。

本教程需要一个良好的数学背景。大多数的例子不需要微积分的知识，但是有一些需要微积分的知识，一些高级特性需要更多的数学。如果你在某一个小节遇到了你不熟悉的数学函数，你可以跳过它，或者用一个你熟悉的函数来替换它，再或者你可以查找一下维基百科学习一下新的知识。一些重要的单不属于常识的数学概念，在必要时也会有所介绍。

## Install

首先你要安装SymPy。参考[安装文档](http://docs.sympy.org/latest/install.html#installation)（译注：推荐使用Anaconda）。

你也可以使用SymPy Live Sphinx 扩展来在浏览器里运行代码。点击下面的绿色“在SymPy Live中运行代码块”按钮试一下。（译注：这个功能在翻译文档中没有，请勿再试了。☺）

**>>> from** **sympy** **import** \*

**>>>** x = symbols('x')

**>>>** a = Integral(cos(x)\*exp(x), x)

**>>>** Eq(a, a.doit())

Eq(Integral(exp(x)\*cos(x), x), exp(x)\*sin(x)/2 + exp(x)\*cos(x)/2)

|  |
| --- |
| **快速提示：**你不需要安装SymPy也可以使用SymPy。你可以在[http://live.sympy.org](http://live.sympy.org/)中使用在线shell（译注：在翻译时译者打不开这个链接。），或者在本文档的右下部有一个shell（译注：这个功能在原网站中才有。） |

SymPy Live shell在底部角落处弹出来，并执行（求值）代码块。你也可以点击一行代码来执行。

SymPy Live shell 是一个全交互式的Python shell。你可以在文本框中输入任何表达式来进行求值。在整个教程中会随时使用它。

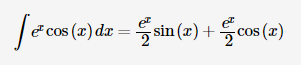
可以随时显示或者隐藏SymPy Live shell，只要点击屏幕右下方的绿色按钮即可。

SymPy Live shell默认使用LaTeX（译注：正确写法是，本文使用LaTeX来代替正确写法。）进行输出，如果你希望输出的结果类似于文档的输出，可以在设置中改变Output Format的值为Str或者Unicode。

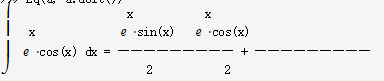
（译注：在修改之后礼记运行可能会报错。请稍等几分钟在运行程序。

展示：

LaTeX



Unicode



Str

Eq(Integral(exp(x)\*cos(x), x), exp(x)\*sin(x)/2 + exp(x)\*cos(x)/2)

）

如果你希望在计算之前修改例子代码。可以在设置中改变Evaluation Mode的值为“copy”。这样会在点击例子之后，复制例子的代码到SymPy Live shell，但是不会对代码求值。你可以修改之后再进行求值。你也可以在输入框中使用键盘上的向上/向下键来调出shell的历史记录。

你也可以在[http://live.sympy.org](http://live.sympy.org/)获取到SymPy Live shell 的一些额外特性，例如手机将强版和保存历史记录。

## 练习

这个教程是2013年在得克萨斯州奥斯汀举办的SciPy会议中的教程的基础。教程的站点在[这里](http://certik.github.io/scipy-2013-tutorial/html/index.html)（译注：失效）。这里有一些视频、原始材料、以及IPython notebook练习。IPython notebook练习强烈推荐给任何通过本教程学习的人。

## 关于本教程

本教程的目标是给没有使用过SymPy库的人一个关于SymPy的介绍。在教程中介绍了许多SymPy的特性，但是并不详细。事实上，教程中展示的许多函数都有更多的选项和功能。SymPy文档的其余部分，都在API文档中。在API文档中详细列出了，每个函数的所有特性和选项。

本教程的目标有：

* 对没有用过SymPy的人（但使用过Python并有必要的数学基础）一个适当的指导。
* 风格应该既简单有有趣，读起来像是一本书。
* 提供一些有意义（insightful）的例子和练习，用来帮助读者学习，并使这个过程轻松愉快一些。
* 按照逻辑顺序介绍一些概念。（To introduce concepts in a logical order.）
* 使用好的实践和习惯用法，避免一些反模式（antipatterns）。容易导致反模式的函数或方法是要尽力避免的。只对高级用户有用的一些特性在本文档中没有展示。
* 始终坚持的一点是，如果做一件事情有多种途径，只展示最好的那一种方式。
* 避免不必要的重复，假设读者都已经看过了本教程的之前部分。

对本教程和SymPy的反馈总是被欢迎的，我们的[邮件列表](https://groups.google.com/forum/?fromgroups#!forum/sympy)。

# 介绍

## 什么是符号计算

符号计算是用于计算符号化的数学对象。意思是说，数学对象是被精确地表示，而不是一个近似值。并且在符号表达式的左侧放置待计算的变量。

举个例子，假如说我们想使用Python内置函数来计算平方数，我们可能像这样做：

**>>> import** **math**

**>>>** math.sqrt(9)

3.0

9是一个完全平方数，所以我们得到了精确的答案：3。但是假如我们要计算的不是完全平方数呢？

**>>>** math.sqrt(8)

2.82842712475

这时我们得到了近似解。2.82842712475不是8的精确地平方根（事实上，是一个无理数，无法使用有限小数来表示。）。如果我们在意的小数表示形式，我们可以这样做。

但是假设我们想要得到更多。回忆一下，。我们很难从之前结果上推断出这个（实际值）。这就是为什么要用符号计算了。在SymPy一类的符号计算中，非完全平方数的平方根是默认不计算的。

**>>> import** **sympy**

**>>>** sympy.sqrt(3)

sqrt(3)

此外——诚如我们之前看到的那样，这才是符号计算的真正力量——符号化的结果是被简化的。

**>>>** sympy.sqrt(8)

2\*sqrt(2)

## 一个更有趣的例子

上面的例子展示了我们如何使用SymPy精确计算一个无理数。但是SymPy拥有更强大的力量。符号计算系统（顺便说一句，符号计算系统还经常被称之为计算机代数系统，或者CASs）还可以计算带变量的符号表达式。

我们继续往下看，在SymPy中，变量使用symbols来定义。和许多符号处理系统不同，在SymPy中变量必须在使用之前被定义（这样做的理由将在下一节讨论）

让我们定义一个符号表达式来表示这样一个数学表达式：

**>>> from** **sympy** **import** symbols

**>>>** x, y = symbols('x y')

**>>>** expr = x + 2\*y

**>>>** expr

x + 2\*y

注意：我们（一般）写就是说我们把x和y当做普通的Python的变量，（然后计算他的值）。但是在此处，他取代了要计算的值。表达式就只是，现在让我们查看一下：

**>>>** expr + 1

x + 2\*y + 1

**>>>** expr - x

2\*y

注意上面的例子，当我们输入expr – x，我们没有计算x + 2\*y – x，而是 2\*y。这里的x和-x自动的消除了。这和上面的sqrt(8)化简为2\*sqrt(2)类似。但是，在SymPy中并不总是如此：

**>>>** x\*expr

x\*(x + 2\*y)

这里我们希望能够转换为，但是正相反，我们看到的表达式是前者。这是SymPy的默认主题。除了类似于和较明显的简化，大部分简化并不是自动执行的。这是因为，我们更喜欢这种因式分解形式，或者这种扩展的形式。两种形式在不同的情况下都有用。在SymPy中，提供了从一种转变为另一种的功能。

**>>> from** **sympy** **import** expand, factor

**>>>** expanded\_expr = expand(x\*expr)

**>>>** expanded\_expr

x\*\*2 + 2\*x\*y

**>>>** factor(expanded\_expr)

x\*(x + 2\*y)

## 符号计算的威力

像SymPy这种符号计算系统的真正威力在于计算各种类型的符号计算的能力。SymPy可以简化表达式、计算导数、积分和极限，求解方程，矩阵处理以及其他所有的符号计算。SymPy还包括一些作图、打印（比如输出一个漂亮的2D数学公式，或者LaTeX）、代码生成、物理、统计学、组合数学、数论、几何学、逻辑，以及其他模块。这里只是举几类简单的SymPy符号计算的例子来刺激一下你的学习兴趣。

**>>> from** **sympy** **import** \*

**>>>** x, t, z, nu = symbols('x t z nu')

使用Unicode字符来打印更多的例子。

**>>>** init\_printing(use\_unicode=**True**)

计算的导数

**>>>** diff(sin(x)\*exp(x), x)

x x

ℯ ⋅sin(x) + ℯ ⋅cos(x)

（译注：在pycharm中显示是exp(x)\*sin(x) + exp(x)\*cos(x)，之后不再一一注释）

计算

**>>>**integrate(exp(x)\*sin(x) + exp(x)\*cos(x), x)

x

ℯ ⋅sin(x)

计算

**>>>** integrate(sin(x\*\*2), (x, -oo, oo))

√2⋅√π

─────

2

找到

**>>>** limit(sin(x)/x, x, 0)

1

求解

**>>>** solve(x\*\*2 - 2, x)

[-√2, √2]

求解微分方程

**>>>** y = Function('y')

**>>>** dsolve(Eq(y(t).diff(t, t) - y(t), exp(t)), y(t))

-t ⎛ t⎞ t

y(t) = C₂⋅ℯ + ⎜C₁ + ─⎟⋅ℯ

⎝ 2⎠

找出的特征值

**>>>** Matrix([[1, 2], [2, 2]]).eigenvals()

⎧3 √17 √17 3 ⎫

⎨─ + ───: 1, - ─── + ─: 1⎬

⎩2 2 2 2 ⎭



（译注：pycharm结果：{-sqrt(17)/2 + 3/2: 1, 3/2 + sqrt(17)/2: 1}）

用球贝塞尔函数来表示一般贝塞尔函数

**>>>** besselj(nu, z).rewrite(jn)

√2⋅√z⋅jn(ν - 1/2, z)

────────────────────

√π

（译注：pycharm结果：sqrt(2)\*sqrt(z)\*jn(nu - 1/2, z)/sqrt(pi)。

这里的变量有些混乱，nu就是下标v，函数jn就是球贝塞尔函数。**一般贝塞尔函数**是常微分方程的标准解函数。这个常微分方程一般称为贝塞尔方程（在圆柱坐标系下求解拉普拉斯方程所得）。解函数无法用初等函数来表示：。其中的和函数有多种表示，一般使用第一类贝塞尔函数和第二类贝塞尔函数，即。在球坐标系下求解三维亥姆霍兹方程得到另一种贝塞尔方程：。对该方程求解得到**球贝塞尔函数**和。。

以上数学资料我全不懂，参考（百度百科资料不全）：<https://zh.wikipedia.org/wiki/%E8%B4%9D%E5%A1%9E%E5%B0%94%E5%87%BD%E6%95%B0>）

使用LaTeX打印

**>>>** latex(Integral(cos(x)\*\*2, (x, 0, pi)))

\int\_{0}^{\pi} \cos^{2}{\left (x \right )}\, dx

## 为什么是SymPy？

有许多计算机代数系统，这篇[维基百科](https://en.wikipedia.org/wiki/List_of_computer_algebra_systems)的文章列出了其中的一些。为什么SymPy是一个好的选择呢？

首先，SymPy是完全免费的。SymPy是一个开源的系统，他遵守BSD许可协议，所以如果你愿意你可以修改源码甚至销售。比较而言，Maple或者Mathematics等其他的商业系统动辄需要数百美元的许可费用。

其次，SymPy是基于Python的。大多数计算机代数系统都发明了自己的语言，但SymPy没有这样做。SymPy完全使用Python写的，也是用Python来执行的，这意味着，如果你学会了Python你可以很容易的开始SymPy，因为你已经知道了语法（并且如果你不会Python，这也很容易学习。）我们已经知道Python是一个设计优良并通过测试的语言。SymPy的开发者对其开发数学软件的能力十分自信，但是编程语言的设计是完全不同的一件事。使用已经存在的语言，可以让我们只专注与把一件事情做好：数学。

其他的计算机代数系统，例如Sage也使用Python这个语言。但是Sage太大了，需要超过1G的空间。SymPy的优势就是轻量级。除了相对较小之外，她也不依赖任何其他的Python包，所以她几乎可以很方便的在任何地方使用。此外Sage的目标和SymPy的目标是不同的。Sage的目标是一个全面的数学系统，他是通过整合已有的开源数学系统来实现其功能的。当你调用某个Sage函数，比如：integrate，他实际调用的是其包含的某个开源包的函数。事实上，SymPy是被Sage包含在内的。相反，SymPy的目标是成为一个独立的系统，所有的特性都用其本身来实现。

SymPy最后一个重要特性是她可以被当做一个库来被使用。许多计算机代数系统专注于在一个交互的环境中被使用。但是如果你希望自动化或者扩展他们，这是很困难的。对于SymPy，你可以很容易的在交互式Python环境中使用她，或者把她导入到你自己的Python应用中。SymPy还提供了很多API，以供使用者方面的在自己定制的函数中扩展她。

# 难点

首先，我们要先明确SymPy的一些事情。和NumPy、Django以及Python标准库中的一些模块（sys 或者 re）一样，SymPy只不过是Python的一个库而已。这意味着，SymPy不会添加任何东西到Python中。Python中固有的一些局限性，也是SymPy的局限性（译注：好别扭。就是说Python做不到的SymPy也做不到）。这也意味着SymPy尽可能的使用Python风格，来使哪些熟悉Python编程的人更容易使用SymPy进行编程。举个简单的例子，SymPy使用Python语法来建立表达式。在Python中不允许隐式乘法（例如：3x或者3 x），因此在SymPy中也不被允许使用。要表达3乘以x，你必须写作3\*x（不能省略\*）。

## 符号

一个重要的结果是，SymPy可以在任何Python允许的环境下使用。我们要向其他库一样来导入她。

**>>> from** **sympy** **import** \*

这可以将SymPy中所有的函数和类都导入到我们交互式Python环境中，现在我们可以做计算了。

**>>>** x + 1

Traceback (most recent call last):

*...*

NameError: name 'x' is not defined

哦哦哦！发生了什么？我们试着使用变量x，但是她告诉我们x没有被定义。在Python中，变量在被定义之前没有任何意义。SymPy并没有什么不同。这和你之前使用过的许多符号处理系统不一样。在SymPy中，不会自动定义变量。如果要定义一个变量，我们必须使用symbols

**>>>** x = symbols('x')

**>>>** x + 1

x + 1

symbols可以接受一系列用空格或者逗号分隔开的变量名，并创建这些符号。我们可以分配这些变量名。之后，我们会展示一下解决这个问题的便捷做法。现在让我们来先定义最普通的变量名：x，y和z，以便接下来使用。

**>>>** x, y, z = symbols('x y z')

需要注意的是，我们标记的符号的名字和变量的名字，彼此之间不需要有任何联系。

**>>>** a, b = symbols('b a')

**>>>** a

b

**>>>** b

a

此处我们使用了一种非常混乱的方式，把符号a分配给变量b，符号b分配给变量a。现在Python变量名a指向了SymPy符号名b，反之亦然。（译注：a, b = symbols('b a')前面的a，b是Python变量名，后面括号中的b a是SymPy符号名）相当混乱。我们也可以这样做：

**>>>** crazy = symbols('unrelated')

**>>>** crazy + 1

unrelated + 1

这里展示了，如果我们需要，符号可以有一个长度大于一个字符的名字。

通常，Python变量名和符号名最好一致。尽管有一些例外：符号名字可以包含一些Python中不允许变量名使用的字符，或者只是希望避免使用太长的名字（分配一个长名字的符号给短名字的Python变量）。

为了避免混乱，本教程自始至终，符号的名字和Python变量的名字都是一致的。而且，“符号”这个词代表SymPy中的符号，“变量”这个词代表Python中的变量。

最后，让我们理解一下SymPy符号和Python有何区别。思考一下下面这个例子：

x = symbols('x')

expr = x + 1

x = 2

print(expr)

改变x的值为2，并没有影响到expr，这是因为x = 2改变的是Python中的变量x为2，而不影响SymPy符号x。我们创建expr时使用的是符号x。当我们创建expr时，Python变量x是符号x，在我们创建之后，我们改变了Python变量x的值为2。但是expr的仍然没有变。这个特性并不是SymPy所特有的。所有Python程序都是这样：如果变量改变了，表达式仍然是创建时的样子，不会自动改变。举个例子：

**>>>** x = 'abc'

**>>>** expr = x + 'def'

**>>>** expr

'abcdef'

**>>>** x = 'ABC'

**>>>** expr

'abcdef'

在这个例子中，如果我们想知道当x的值改变之后expr的值是什么，我们需要重新执行创建expr的代码，即expr = x + 1。如果使用几行代码来创建expr显得有些复杂。使用SymPy这种符号计算系统的一个优势就是我们可以给expr建立一个形象化的表达式，同时带入x。 在SymPy中，使用subs来处理这种情况是一个比较合适的方法。之后会有更详细的讨论。

**>>>** x = symbols('x')

**>>>** expr = x + 1

**>>>** expr.subs(x, 2)

3

|  |
| --- |
| **快速提示：**可以使用subs来改变表达式中的某个Symbol值。  **>>>** x = symbols('x')  **>>>** expr = x + 1  **>>>** expr.subs(x, 2)  3 |

## 等号

实际上，SymPy并没有扩展Python的语法。一个重要的表现就是，在Python中的=并不等价于SymPy中“相等”，相反他在Python中表示赋值。这是Python语言定死的，SymPy并没有尝试来改变他。

然而，你可能认为，在Python中用来判断相等的==，在SymPy中也是一样。这也是不对的。让我们看一下当我们使用了==发生了什么。

**>>>** x + 1 == 4

False

当我们执行x + 1 == 4，我们仅仅得到了False。在SymPy中，==用于判断结构上的精确相等。这意味着，a == b的意思是我们要知道a是否等于b。==的结果一定是一个bool类型。

（译注：可以这样理解==，它用于判断两个变量是否在外观上是同一个表达式。结构精确相等下面还会再解释的。

**>>>** x = symbols('x')  
**>>>** expr1 = (x + 1)\*\*2  
**>>>** expr2 = (x + 1)\*\*2  
**>>>** expr3 = x\*\*2 + 2\*x + 1  
**>>>** expr1 == expr2

True

**>>>** expr1 == expr3

False

**>>>** expr1.equals(expr3)

True

）

创建符号等式，要使用另一个对象叫做Eq。

**>>>** Eq(x + 1, 4)

Eq(x + 1, 4)

（译注：Eq用来表示这样一个表达式：x + 1 == 4）

关于==还有一点需要说明，假设我们想知道是否和是否相等。我们可以这样做：

**>>>** (x + 1)\*\*2 == x\*\*2 + 2\*x + 1

False

我们又一次得到了False。但是确实等于。这里到底发生了什么？我们是不是发现了SymPy的一个bug啊？或者我们只是学识浅薄，还不懂这个基本代数常识？

回忆一下我们上面讲到的，==用来判断两个表达式的结构精确相等。“精确”在这里的意思是，表达式在结构上完全相等。这里和在外形上并不相同。一个是两个数据项和的平方，另一个是三个数据项之和。

事实证明，当使用SymPy库时，用==判断结构上的精确相等比用它表示符号相等或者判断数学相等更有用。（译注：结构相等：看起来两个表达式一样；符号相等：显示的等号；数学相等：两个表达式的数学意义是一样的。）但是作为新手，你可能更在意后两者。我们有一个可选的符号用于标识符号相等：Eq。如果要判断两个事物相等，做好是利用一个基本原则：如果a=b，则a-b=0。因此，检查a是否等于b最好的办法就是计算a-b，并简化结果，看一看它是否为0。我们之后会学到一个叫做simplify的函数专门用于做这件事。这个方法并不是绝对正确的，事实上，他可以被证明这个函数可能判断两个符号表达式是否相等。但是他只是对常见的情况表现良好。

**>>>** a = (x + 1)\*\*2

**>>>** b = x\*\*2 + 2\*x + 1

**>>>** simplify(a - b)

0

**>>>** c = x\*\*2 - 2\*x + 1

**>>>** simplify(a - c)

4\*x

还有一个叫做equals的方法，通过若干个随机点的数值来判断两个表达式是否相等

**>>>** a = cos(x)\*\*2 - sin(x)\*\*2

**>>>** b = cos(2\*x)

**>>>** a.equals(b)

True

## 最后两点：^ 和/

你可能已经注意到了，我们使用\*\*，来代替标准的^进行求幂运算。那是因为SymPy继承了Python的惯例。^代表逻辑运算：异或。SymPy也继承了Python的这个惯例

**>>> True** ^ **False**

True

**>>> True** ^ **True**

False

**>>>** x^y

Xor(x, y)

最后一个小小的技巧问题： SymPy是如何有序工作的？当你敲出一些类似于x + 1 时，SymPy符号x被加上了一个Python整数1。Python的操作规则是允许SymPy告诉Python：SymPy对象知道如何和Python整数相加，然后1自动的转为SymPy的整型对象。

这种操作自动的在后台完成，你一般不需要知道究竟发生了什么。但是有一个特例，当你结合两个SymPy对象，或者结合一个SymPy对象和一个Python对象时，你会得到一个SymPy对象，但无论如何你结合两个Python对象，SymPy都不会运行，你只能得到一个Python对象。

**>>>** type(Integer(1) + 1)

<class 'sympy.core.numbers.Integer'>

**>>>** type(1 + 1)

<... 'int'>

注意：在Sympy Live中运行上面的例子，（1+1）是被Integer包围的，所以不会输出正确结果。

（译注：不是很理解上面提示的问题。

在Pycharm中测试

print(type(Integer(1) + 1))  
print(type(1 + 1))  
print(type(Integer(1)))  
print(Integer(1) + 1)  
print(1+1)

结果

<class 'sympy.core.numbers.Integer'>

<class 'int'>

<class 'sympy.core.numbers.One'>

2

2

）

这通常不是一个太大的问题，Python的整数和SymPy的Integers基本一致，但是有一个重要的特例：除法。在SymPy中，两个整数相除得到的是一个有理数。

**>>>** Integer(1)/Integer(3)

1/3

**>>>** type(Integer(1)/Integer(3))

<class 'sympy.core.numbers.Rational'>

但是在Python中，/表示两个整数或浮点数相除。结果取决于你是用的是Python2还是Python3（译注：在Python2中1/2的结果是0，在Python3中结果是0.5），以及你是否运行了from \_\_future\_\_ import division。

**>>> from** **\_\_future\_\_** **import** division

**>>>** 1/2

0.5

注意：在Sympy Live中运行上面的例子，（1+1）是被Integer包围的，所以不会输出正确结果。

为了避免这个问题，我们可以显式构造一个有理数对象。

**>>>** Rational(1, 2)

1/2

这个问题也会导致另一个问题：当我们都使用一个包含整数/整数的表达式时，例如：

**>>>** x + 1/2

x + 0.5

注意：在Sympy Live中运行上面的例子，（1+1）是被Integer包围的，所以不会输出正确结果。

发生这个问题的原因是Python先计算1/2的值为0.5，然后在和x相加之前，将结果转成SymPy类型。同样，我们也可以显式创建一个有理数：

**>>>** x + Rational(1, 2)

x + 1/2

在[难点与陷阱](http://docs.sympy.org/latest/gotchas.html" \l "gotchas)文档中有一些小技巧来避免这些问题。

## 扩展阅读

关于这一节的更多主题参考[难点与陷阱](http://docs.sympy.org/latest/gotchas.html#gotchas)。

# 基本操作

现在我们开始讨论一些在SymPy中处理表达式所需的一些基本操作。一些更高级的操作之后将会在[高级表达式处理](http://docs.sympy.org/latest/tutorial/manipulation.html" \l "tutorial-manipulation)小节讨论。（译注：最后一节，更新链接）

**>>> from** **sympy** **import** \*

**>>>** x, y, z = symbols("x y z")

## 置换

在数学表达式处理中，你可能做常用的操作就是置换了。置换就是用其他的内容来替代表达式里面的实例。可以使用subs函数来完成置换操作。例如

**>>>** expr = cos(x) + 1

**>>>** expr.subs(x, y)

cos(y) + 1

通常有两种情况需要做置换：

1、计算表达式在某个点的值。例如我们表达式是cos(x) + 1，我们想要计算他在x = 0 点出的值，即我们计算cos(0) + 1，值是2。

**>>>** expr.subs(x, 0)

2

2、用一个子表达式代替另一个子表达式。我们这样做有两个理由，第一个理由是，当我们尝试建立一个嵌套表达式，例如：。（译注：原文一共四层x，但是不符合下面的说明，所以改为3层。）建立这个表达式，我们从x\*\*y开始，用x\*\*y替换x，我们得到x\*\*(x\*\*y)。如果我们在用一个新的表达式x\*\*x来代替y，我们就得到了x\*\*(x\*\*(x\*\*x))，这就是目标表达式。

**>>>** expr = x\*\*y

**>>>** expr

x\*\*y

**>>>** expr = expr.subs(y, x\*\*y)

**>>>** expr

x\*\*(x\*\*y)

**>>>** expr = expr.subs(y, x\*\*x)

**>>>** expr

x\*\*(x\*\*(x\*\*x))

第二个理由是：我们可能要做一些化简操作，或者可能出现不化简就无法操作的情况。例如，我们有这样的表达式：sin(2x)+cos(2x)，我们希望用2sin(x)cos(x)来代替sin(2x)，我们之后会学到expand\_trig函数就是用来完成这个的。但是这个函数会同时展开cos(2x)，这个的结果并不是我们想要的。我们会在[高级表达式处理](http://docs.sympy.org/latest/tutorial/manipulation.html" \l "tutorial-manipulation)一节学习到一些精确化简的方法。（译注：最后一节，更新链接）最简单的办法就是使用2sin(x)cos(x)来代替sin(2x)

**>>>** expr = sin(2\*x) + cos(2\*x)

**>>>** expand\_trig(expr)

2\*sin(x)\*cos(x) + 2\*cos(x)\*\*2 - 1

**>>>** expr.subs(sin(2\*x), 2\*sin(x)\*cos(x))

2\*sin(x)\*cos(x) + cos(2\*x)

关于subs有两个需要注意的地方。意思他返回了一个新的表达式。SymPy对象是不可变的，这意味着，subs没有改变原始表达式，例如：

**>>>** expr = cos(x)

**>>>** expr.subs(x, 0)

1

**>>>** expr

cos(x)

**>>>** x

x

当我们看到执行expr.subs(x, 0)时，expr并没改变。事实上，因为SymPy表达式是不可变的，所以没有函数可以改变他们，所有的函数都是返回一个新的表达式。

**快速提示：**SymPy表达式是不可变的，没有函数可以改变原始表达式。

同时处理多个替换，可以使用（旧元素，新元素）这样的列表来处理

**>>>** expr = x\*\*3 + 4\*x\*y - z

**>>>** expr.subs([(x, 2), (y, 4), (z, 0)])

40

一般我们可以这样结合使用：用list包含一些带替换的变量，进行批量操作。例如：我们有一个表达式：，我们希望替换其中的一些x变为y：

**>>>** expr = x\*\*4 - 4\*x\*\*3 + 4\*x\*\*2 - 2\*x + 3

**>>>** replacements = [(x\*\*i, y\*\*i) **for** i **in** range(5) **if** i % 2 == 0]

**>>>** expr.subs(replacements)

-4\*x\*\*3 - 2\*x + y\*\*4 + 4\*y\*\*2 + 3

## 字符串转换为SymPy表达式

sympify函数（不要和simplify函数混淆）用于将字符串转为SymPy表达式。

例如：

**>>>** str\_expr = "x\*\*2 + 3\*x - 1/2"

**>>>** expr = sympify(str\_expr)

**>>>** expr

x\*\*2 + 3\*x - 1/2

**>>>** expr.subs(x, 2)

19/2

警告：sympify使用了eval，不要给他输入错误的值。

（译注：eval将字符串str当成有效的表达式来求值并返回计算结果。）

## evalf

用evalf计算一个表达式的值，返回浮点数。

**>>>** expr = sqrt(8)

**>>>** expr.evalf()

2.82842712474619

SymPy可以计算一个浮点数到任意经度。默认精度是15位数字（译注：包括整数部分）。但是你可以给evalf一个数字参数。现在计算一下100位长度的吧。

**>>>** pi.evalf(100)

3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117068

如果要计算一个符号表达式在某个点的数值，我们应该在先使用subs再使用evalf，但是更高效稳定的方法是使用带字典的subs参数替换evalf内的值。

**>>>** expr = cos(2\*x)

**>>>** expr.evalf(subs={x: 2.4})

0.0874989834394464

有时候，计算之后的舍入误差，小于希望的精度。用户可以通过设置chop的值自由决定后面的数字是否被舍弃。

**>>>** one = cos(1)\*\*2 + sin(1)\*\*2

**>>>** (one - 1).evalf()

-0.e-124

**>>>** (one - 1).evalf(chop=**True**)

0

## lambdify

如果你希望做一些简单的计算subs和evalf都很好用。但是如果你要计算有许多值的表达式。那么有更高效的方法。例如：如果你希望计算一个表达式有1000个值。使用SymPy可能会比你期望的要慢，特别是你比较在意精度的时候。你可以使用[NumPy](http://www.numpy.org/)和[SciPy](https://www.scipy.org/)来代替她。

最简单的办法就是用lambdify函数来吧SymPy表达式转换为一个可计算的表达式。lambdify在外形上看起来和lambda函数很像。除了他可以把SymPy名字转换为一个数值化库的名字（通常是NumPy）（except it converts the SymPy names to the names of the given numerical library, usually NumPy.）举个例子。

**>>> import** **numpy**

**>>>** a = numpy.arange(10)

**>>>** expr = sin(x)

**>>>** f = lambdify(x, expr, "numpy")

**>>>** f(a)

[ 0. 0.84147098 0.90929743 0.14112001 -0.7568025 -0.95892427

-0.2794155 0.6569866 0.98935825 0.41211849]

警告：sympify使用了eval，不要给他输入错误的值。

你可以使用其他的库来代替NumPy。例如使用标准的库函数math。

**>>>** f = lambdify(x, expr, "math")

**>>>** f(0.1)

0.0998334166468

**>>> def** mysin(x):

**...**  *"""*

**...**  *My sine. Note that this is only accurate for small x.*

**...**  *"""*

**...**  **return** x

**>>>** f = lambdify(x, expr, {"sin":mysin})

**>>>** f(0.1)

0.1

To use lambdify with numerical libraries that it does not know about, pass a dictionary of sympy\_name:numerical\_function pairs.

通过{SymPy 名字,数值函数名}这样的字典对，可以使lambdify来处理他不知道的数值函数库。例如

**>>> def** mysin(x):

**...**  *"""*

**...**  *My sine. Note that this is only accurate for small x.*

**...**  *"""*

**...**  **return** x

**>>>** f = lambdify(x, expr, {"sin":mysin})

**>>>** f(0.1)

0.1

# 打印

就像我们看到的，SymPy使用Unicode字符集可以漂亮的打印数据。这里简短介绍一下在SymPy中的一些打印参数。

## 打印模式（printers）

在SymPy中有一些可用的打印模式，常用的有：

* str
* srepr
* ASCII pretty printer
* Unicode pretty printer
* LaTeX
* MathML
* Dot

除了这些，还有一些方式可以将SymPy对象输出为代码，例如：C、Fortran、JavaScript、Theano和Python，这些不在本教程的讨论之内。

## 设置漂亮的打印

如果你希望有一个最漂亮的打印效果，请使用init\_printing()函数，它将自动根据你的环境设置最好的打印模式。

**>>> from** **sympy** **import** init\_printing

**>>>** init\_printing()

如果你想在交互式环境下工作，init\_session()函数将自动导入SymPy的一切：创建一些常用的符号，设置绘图，运行init\_printing()。

**快速提示：**你也可以使用SymPy Live 来改变设置模式，只要高边设置中的“输出格式”即可。

**>>> from** **sympy** **import** init\_session

**>>>** init\_session()

Python console **for** SymPy 0.7.3 (Python 2.7.5-64-bit) (ground types: gmpy)

These commands were executed:

>>> **from** **\_\_future\_\_** **import** division

>>> **from** **sympy** **import** \*

>>> x, y, z, t = symbols('x y z t')

>>> k, m, n = symbols('k m n', integer=**True**)

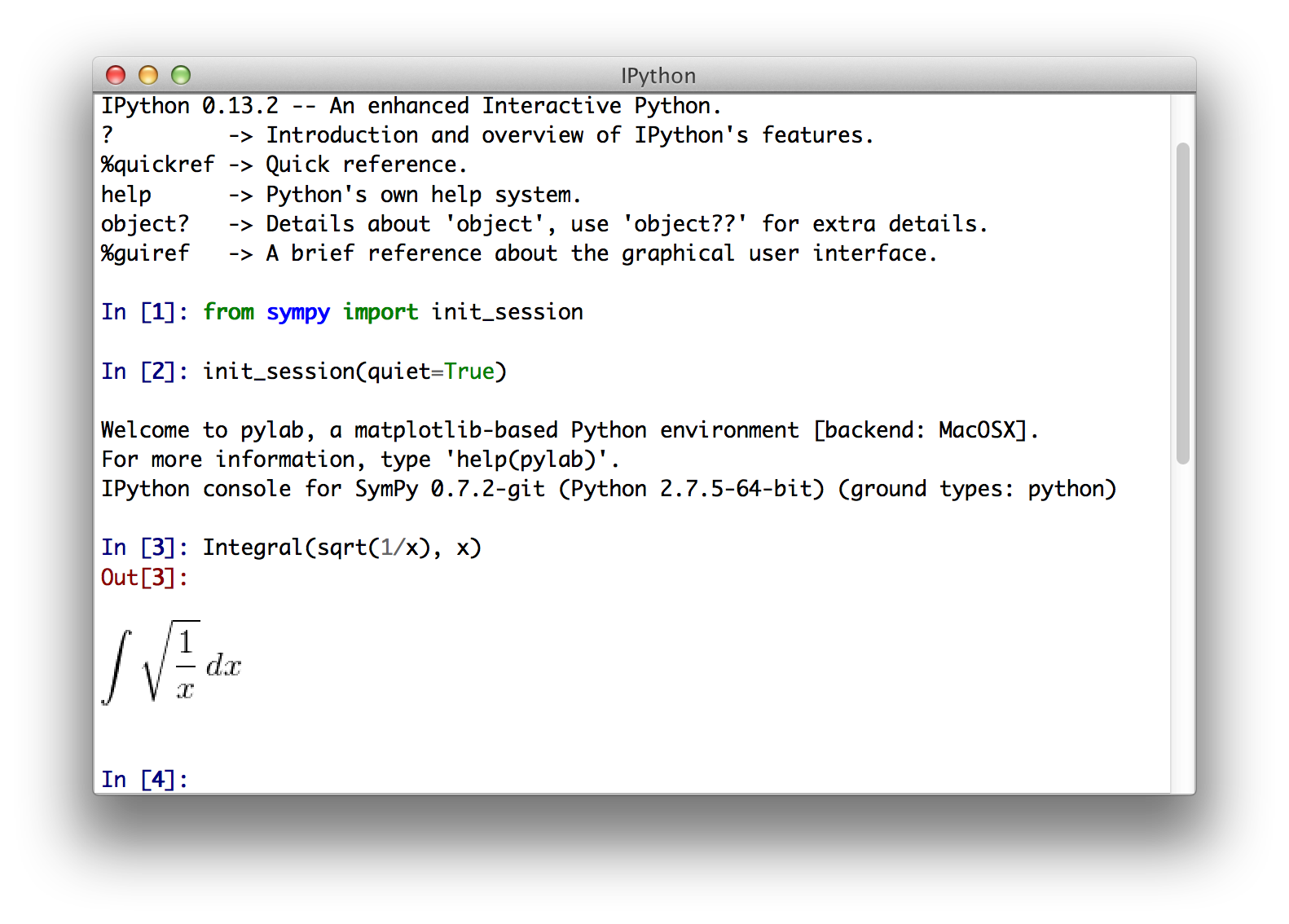
>>> f, g, h = symbols('f g h', cls=Function)

>>> init\_printing() *# doctest: +SKIP*

Documentation can be found at http://www.sympy.org

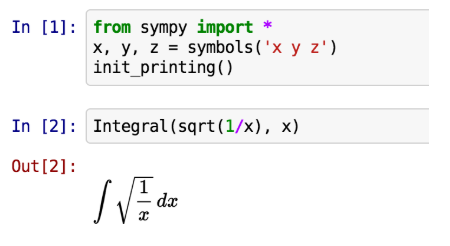
在任何情况下都会发生以下事情：

* 在IPython QTConsole中，如果LaTeX被按章，他将使用这种模式。

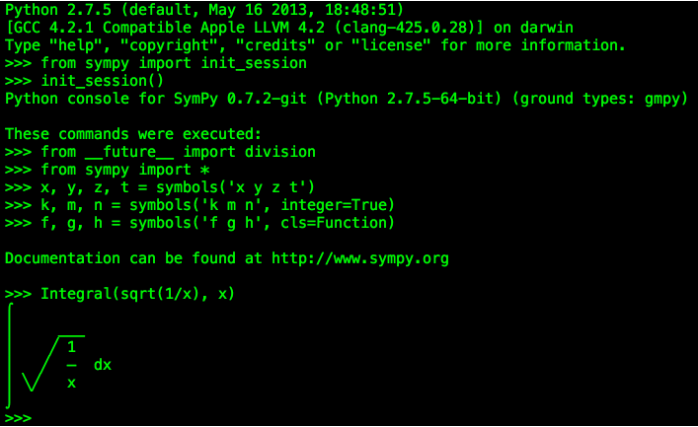


如果没有装LaTeX，但是装了Matplotlib，会使用Matplotlib的渲染引擎，如果Matplotlib也没有装，使用Unicode pretty printer。

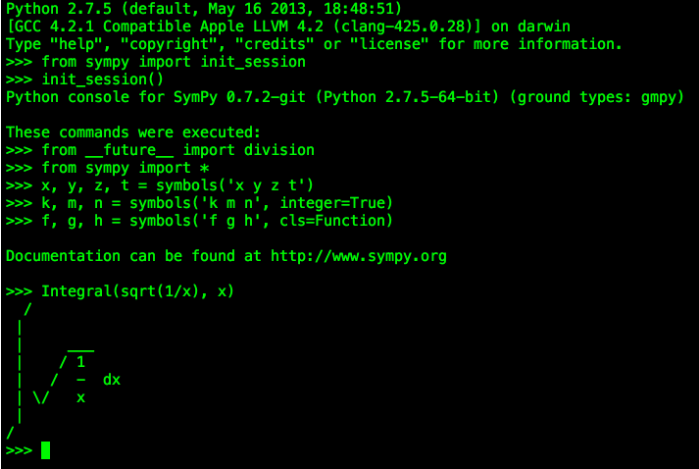
* 在IPython notebook中，使用MathJax来渲染LaTeX。



在IPython控制台中，或者一个规律的Python会话中，如果控制台支持Unicode，将使用Unicode pretty printer



如果不支持Unicode，则使用ASCII pretty printer。



## 打印函数

除了自动的打印之外，你可以用合适的函数来明确的设置一种打印模式。

### str

使用str(expr)可以将一个表达式转换为字符格式。print(expr)也可以产生同样的效果。字符样式是一种非常易读的样式。而且这种样式是符合Python语法的，所以他可以方便的进行复制和粘贴。str()生成的表达式和你输入的表达式看起来是一样的。

**>>> from** **sympy** **import** \*

**>>>** x, y, z = symbols('x y z')

**>>>** str(Integral(sqrt(1/x), x))

'Integral(sqrt(1/x), x)'

**>>>** print(Integral(sqrt(1/x), x))

Integral(sqrt(1/x), x)

### srepr

srepr样式的表达式是用来显示一个表达式的具体形式。在[高级表达式处理](http://docs.sympy.org/latest/tutorial/manipulation.html#tutorial-manipulation)一节中会有进一步的探讨。现在使用它就好[[1]](#footnote-1)。

**>>>** srepr(Integral(sqrt(1/x), x))

"Integral(Pow(Pow(Symbol('x'), Integer(-1)), Rational(1, 2)), Tuple(Symbol('x')))"

Srepr样式主要是用于了解如何构建一个表达式。

### ASCII Pretty Printer

ASCII pretty printer样式是通过pprint()来进行操作的。如果终端不支持Unicode，ASCII将是默认样式。否则你必须设置use\_unicode=False

**>>>** pprint(Integral(sqrt(1/x), x), use\_unicode=**False**)

/

|

| \_\_\_

| / 1

| / - dx

| \/ x

|

/

pprint()将输出打印到屏幕上，如果你希望输出string样式，请使用pretty()。

**>>>** pretty(Integral(sqrt(1/x), x), use\_unicode=**False**)

' / \n | \n | \_\_\_ \n | / 1 \n | / - dx\n | \\/ x \n | \n/ '

**>>>** print(pretty(Integral(sqrt(1/x), x), use\_unicode=**False**))

/

|

| \_\_\_

| / 1

| / - dx

| \/ x

|

/

### Unicode Pretty Printer

Unicode pretty printer也是通过pprint() 和pretty()来实现的。如果终端支持Unicode，则它是默认样式，如果监测不到pprint()则说明终端不支持Unicode。你可以使用use\_unicode=True来设置使用Unicode。

**>>>** pprint(Integral(sqrt(1/x), x), use\_unicode=**True**)

⌠

⎮ \_\_\_

⎮ ╱ 1

⎮ ╱ ─ dx

⎮ ╲╱ x

⌡

### LaTeX

使用latex()函数来使表达式用LaTeX方式展示。

**>>>** print(latex(Integral(sqrt(1/x), x)))

\int \sqrt{\frac{1}{x}}\, dx

latex()有很多的选项来设置不同的样式，具体参考[他的文档](http://docs.sympy.org/latest/modules/printing.html#sympy.printing.latex.latex)。

### MathML

还有一种MathML样式，用print\_mathml()来调用。它必须使用sympy.printing.mathml来导入。

**>>> from** **sympy.printing.mathml** **import** print\_mathml

**>>>** print\_mathml(Integral(sqrt(1/x), x))

<apply>

<int/>

<bvar>

<ci>x</ci>

</bvar>

<apply>

<root/>

<apply>

<power/>

<ci>x</ci>

<cn>-1</cn>

</apply>

</apply>

</apply>

print\_mathml()用于打印输出，如果你希望输出字符，请使用mathml()。

### Dot

sympy.printing.dot中的dotprint()用于dot样式的输出。他可以使用Graphviz渲染。在[高级表达式处理](http://docs.sympy.org/latest/tutorial/manipulation.html#tutorial-manipulation)一节，有很多输出的例子。

**>>> from** **sympy.printing.dot** **import** dotprint

**>>> from** **sympy.abc** **import** x

**>>>** print(dotprint(x+2))

digraph{

# Graph style

"ordering"="out"

"rankdir"="TD"

#########

# Nodes #

#########

"Add(Integer(2), Symbol(x))\_()" ["color"="black", "label"="Add", "shape"="ellipse"];

"Integer(2)\_(0,)" ["color"="black", "label"="2", "shape"="ellipse"];

"Symbol(x)\_(1,)" ["color"="black", "label"="x", "shape"="ellipse"];

#########

# Edges #

#########

"Add(Integer(2), Symbol(x))\_()" -> "Integer(2)\_(0,)";

"Add(Integer(2), Symbol(x))\_()" -> "Symbol(x)\_(1,)";

}

# 化简

为了使文档的可读性更高，我们更换了打印模式。

**>>> from** **sympy** **import** \*

**>>>** x, y, z = symbols('x y z')

**>>>** init\_printing(use\_unicode=**True**)

## 化简

现在我们来看一些有意思的数学知识。化简表达式是符号计算系统最有用的特点之一。SymPy有很多函数用来做各种化简。最常见的就是simplify()用来将各种数学函数智能的转化为一个最简单的表达式。举个例子：

**>>>** simplify(sin(x)\*\*2 + cos(x)\*\*2)

1

**>>>** simplify((x\*\*3 + x\*\*2 - x - 1)/(x\*\*2 + 2\*x + 1))

x - 1

**>>>** simplify(gamma(x)/gamma(x - 2))

(x - 2)⋅(x - 1)

这里的gamma(x)表示，参考[gamma函数](https://en.wikipedia.org/wiki/Gamma_function)。我们看到simplify()可以处理很多类型的表达式。

但是simplify()是有缺陷的。他适用于大部分的SymPy的化简操作。可以用他来尝试化简并决定是否应该化简。此外， “最简”并没有一个确定的定义。例如我们希望化简：为（x+1）2

**>>>** simplify(x\*\*2 + 2\*x + 1)

2

x + 2⋅x + 1

这不是我们想要的结果。之后会探讨的一个函数factor()，用于完成我们想要的功能。

simplify() 的另一个弱点是速度慢，因为他要做出各种样式的化简，然后在其中挑出一个最好的。如果你已经明确知道了你需要的是哪一中样式，最好选择一个特殊的化简函数用于你想要的化简。

1. SymPy没有使用Python内置的repr()函数来做repr样式的输出。因为在Python中的str(list)每个元素都调用了repr()函数。并且有一些SymPy函数返回一个list（如solve()）。srepr()函数很繁琐，大家都不希望默认调用他。 [↑](#footnote-ref-1)