

目录

1 背景知识	4
2 Matplotlib 简介	6
3 Pyplot 的基本用法	8
3.1 Pyplot 绘图的基本方法	8
3.1.1 线图	8
3.1.2 点图	12
3.1.3 条形图	14
3.1.4 直方图	16
3.2 Pyplot 设置坐标轴的基本方法	18
3.2.1 xlim(), ylim() 函数	19
3.2.2 xticks, yticks 函数	19
3.2.3 xlabel, ylabel 函数	19
3.2.4 grid 函数	19
3.2.5 title 函数	19
3.2.6 axis 函数	20
3.3 在图中添加文本	22
3.3.1 text 函数	22
3.3.2 subtitle 函数以及它和 title 的区别	24
3.3.3 annotate 函数	24
4 Axes 的基本用法	26
4.1 创建 Axes 对象	26
4.2 简述 Pyplot 和 Axes 的区别	31
4.3 用 Axes 进行坐标轴的设置	31
4.3.1 反转坐标轴方向	32
4.3.2 控制刻度显示	32
4.3.3 再谈网格线 grid	32
4.3.4 操控四条边框	33
4.3.5 其他	35
5 极坐标系	36
5.1 极坐标系简介	36
5.2 极坐标系的复杂设置	37
6 3D 图像	42

6.1	3D 图总览	42
6.2	浅析 3D 图	44
6.2.1	3D 空间的创建	44
6.2.2	3D 图像再划分——何时要 meshgrid	44
6.3	各类 3D 图的方法	48
6.3.1	线图与点图	48
6.3.2	曲面图	49
6.3.3	线框图	51
6.3.4	等高线图	52
7	事件处理与交互	54
7.1	什么是事件处理与交互——事件机制概述	54
7.2	一些需要阐明的和解释的	56
7.2.1	fig 与 canvas	56
7.2.2	artist 艺术家?	56
7.3	事件交互的实战应用	57
7.3.1	窗口大小改变	58
7.3.2	拖拽事件	59
7.3.3	鼠标、键盘、滚轮大杂烩	61
7.3.4	拾取事件	70
8	详述	71
8.1	如何利用 Python 解方程	71
8.1.1	SciPy-fsolve 法简介	71
8.1.2	SciPy-fsolve 法的深入思考与改良	72
8.2	numpy 数组与列表, 元组	77
8.2.1	区别与联系	77
8.2.2	相互转换	78
8.3	图例的高级设置	79
8.3.1	位置	79
8.3.2	边框与背景	79
8.3.3	标题与字体大小	80
8.3.4	列数	80
8.4	scatter 函数的高级设置	81
8.4.1	点的形状	81
8.4.2	点的透明度	81
8.4.3	边缘颜色	81
8.4.4	边缘宽度	81

8.4.5 颜色映射	81
8.5 简写问题	84
8.5.1 哪些参数可以简写	84
8.5.2 如何简写	84
8.5.3 简写的使用规则	85
8.6 Pyplot 绘图的其他方法	87
8.6.1 箱型图	87
8.6.2 阶梯图	89
8.6.3 茎叶图	90
8.6.4 饼图	91
8.6.5 两曲线间的颜色填充	93
8.6.6 堆积面积图	95
8.6.7 imshow() 函数	97
8.7 颜色映射	99
8.7.1 颜色映射的选取	99
8.7.2 哪些地方可以 cmap	100
8.7.3 norm 的作用	100
8.8 projection 参数: 投影类型的指定	105
8.9 颜色的另类定义	106
8.9.1 RGB 模式	106
8.9.2 RGBA 模式	106
8.9.3 十六进制模式	106
8.10 isinstance() 函数	108

1 背景知识

在正文开始之前，我想先阐明一些与实际作图关系不大但是又十分关键的概念。Metplotlib 是一个基于 Python 的作图库，因此我们需要提前对 Python 语法有基本的了解。

类和对象

“对象”是现实世界中的实例，包含属性（数据）和方法（行为）两个部分，而“类”则是对对象的定义，是对对象的属性和方法的抽象与概括，是对象的模板或蓝图，。总之，类是创建对象的基础，对象是类的实例。

希腊字母和数学符号

表示希腊字母（比如单纯将它们打印出来）的方法主要有两个：一是使用 Unicode 字符，二是使用 LaTex 语法。但是后者需要特定的环境支持，在普通的 Python 脚本中，我们最好，或者说，只能使用前者。

使用 Unicode 的方法是：在前面加上\u，后面接一个四位十六进制数，每个希腊字母以及一些特殊字符对应的这个十六进制数都可以在 Unicode 表中查到。一下是一些常用符号的对应编码：

π	\u03C0
α	\u03B1
β	\u03B2
γ	\u03B3
δ	\u03B4
ϵ	\u03B5
λ	\u03BB
σ	\u03C3
Σ	\u03A3
Π	\u03A0
ψ	\u03C8
Ω	\u03A9
ω	\u03C9
∞	\u221E

数学运算

如果我们需要一些希腊字母参与数学运算，或者计算三角函数，对数，指数，常见的作法是使用 math 库和 numpy 库。首先导入所需库：*import math* 以及 *import numpy as np* 我们可以用 *math.pi*, *math.e* 来表示圆周率和自然对数的值，但是不推荐这么做（因为有时候会莫名其妙报错），我更推荐的做法是用 *np.pi* 和 *np.e*，在后面的绘图中，我们会越来越认识到 Numpy 库的强大。

(这里简单解释一下，math 库是一个 Python 自带的标准库，其有一定的局限性，只能用于单个数值的数学计算，就是输入一个数值返回一个数值的那种运算，因此它对付指数对数、开根绝对值、正弦余弦等等都没问题，但是处理不了一个数组或者是矩阵运算；而 numpy 是一个第三方库，其包含的功能更为丰富，既能处理单个数值的运算，又能处理多数值，而且主要针对的就是多数值的运算，如处理数组、矩阵等)

尽管如此，math 库也拥有一些 Numpy 所不具备的功能，比如 $\text{math.factorial}(x)$ (计算阶乘)， $\text{math.gcd}(a,b)$ (计算最大公约数) 等等。因此在实际问题中需要二者结合使用。

对于一般的单个数值的数学运算，numpy 和 math 均可，以 numpy 为例参考下表：

运算	单个数值	数组
绝对值	<code>np.abs()</code>	<code>np.abs([])</code>
平方根	<code>np.sqrt()</code>	<code>np.sqrt([])</code>
指数 (e 为底)	<code>np.exp()</code>	<code>np.exp([])</code>
自然对数	<code>np.log()</code>	<code>np.log([])</code>
正弦	<code>np.sin()</code>	<code>np.sin([])</code>
余弦	<code>np.cos()</code>	<code>np.cos([])</code>
正切	<code>np.tan()</code>	<code>np.tan([])</code>
数组的和	\	<code>np.sum([])</code>
数组的平均值	\	<code>np.mean([])</code>
数组的中位数	\	<code>np.median([])</code>
数组的方差	\	<code>np.var([])</code>
数组的标准差	\	<code>np.std([])</code>
数组的最大值	\	<code>np.max([])</code>
数组的最小值	\	<code>np.min([])</code>

注意以下几点：

1. 对数组进行单个数值的运算是指以该数值每个元素为单个输入，返回一个由所有输入元素对应的输出值组成的数组
2. 对于对数运算上表只展示了自然对数，若要改变指数，numpy 和 math 是不一样的，比如以 10 为底的对数，在 math 中表示为：`math.log(x,10)`，在 numpy 中表示为：`np.log10(x)`。
3. 至于其他底数的指数运算，请结合一点点数学知识自行转换。

2 Matplotlib 简介

首先，我们需要明确两个 matplotlib 中的基本概念，一个是图形对象，一个是子图。

简单来说，图形对象是创建出来的用来承载图像的那个窗口，如下图：

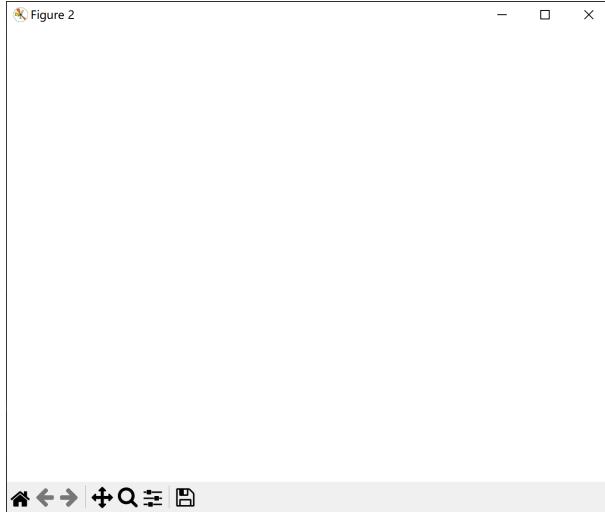


图 1: 2.1.1

而子图则是生成的坐标轴和绘制出的精美的图形，如下图：

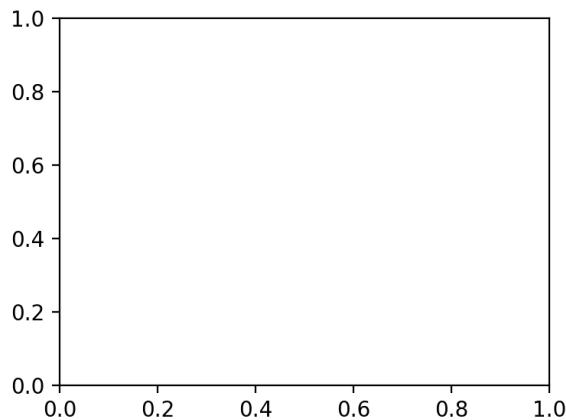


图 2: 2.1.2

其实，我们也可以把子图理解为坐标轴，而绘图的过程是对坐标轴实施的操作。

明晰了这两个概念后，我们就可以把 Matplotlib 绘图的过程视为创建图形对象——创建子图——对子图进行操作的过程。

如何创建图形对象和坐标轴呢？有以下方法：

1. `fig = plt.figure()`

这是通过 plt.figure() 创建了一个图形对象，并赋值给 fig，从而后面我们可以通过“fig”来对这个图形对象进行操作。

2. 如果后面在绘图时使用 plt.plot() 命令，那么我们是不需要创建图形对象的，因为当你调用 plt.plot 时，如果当前没有活跃的坐标轴（连一个图形对象都没有，当然不会有活跃的坐标轴），matplotlib 会自动创建一个新的图形和一个默认的坐标轴。

上述两种方法分别对应了对 Axes 和 pyplot 这两个 Matplotlib 库中用于绘图的主要接口的使用。如果使用 Axes 绘图，我们需要提前创建一个图像对象，而使用 pyplot 则不需要，这使得其使用更加简便，易于上手。除此之外，Axes 和 pyplot 的区别还有：

Pyplot:

1. pyplot 允许链式调用，即在一行代码中连续调用多个方法，例如：

python

```
plt.plot(x, y).title('My Plot').xlabel('X-axis').ylabel('Y-axis').show()
```

图 3: 2.2.1

2. 还有就是我们刚才说的，使用 pyplot 不需要提前创建图形对象，这是因为 pyplot 具有自动管理功能，它能够自动为你管理图形和轴对象，你不需要显式地创建它们。当你调用 plot、scatter 等方法时，pyplot 会在后台为你创建或更新当前的图形和轴。

Axes:

1. 更多的控制：Axes 提供了更多的方法来自定义和控制图表的各个方面，比如设置轴的刻度、添加注解、调整轴的显示范围等。
2. 面向对象：使用 Axes 时，你可以直接与图表的轴对象交互，这使得你可以更精确地控制图表的布局和外观。
3. 多个子图：当你需要在一个图形窗口中创建多个子图时，Axes 提供了更好的支持。你可以创建多个 Axes 实例，并将它们放置在不同的位置上。
4. 高级功能：Axes 支持一些高级功能，比如共享轴、对数轴、极坐标轴等，这些在 pyplot 中可能不那么直观。

接下来我会分别介绍 pyplot 和 Axes 的一些基本用法。

3 Pyplot 的基本用法

在 matplotlib 绘图程序的开头通常有一个导入: `import matplotlib.pyplot as plt`. 这里所做的就是导入了 matplotlib 库里的 pyplot 类, 并将其简称为 plt。后面程序中出现以 plt. 为前缀的命令就是在调用 pyplot 的方法。

3.1 Pyplot 绘图的基本方法

3.1.1 线图

线图, 顾名思义, 就是将所有离散的点连成线的图像。绘制线图有两种基本操作:

方法 1: 指定 x 的取点范围和密度, 然后指定 $x \rightarrow y$ 的映射关系 (写出函数表达式), 最后 `plt.plot(x,y)`。

方法 2: 直接指定所有图像中确定经过的点的 x,y 坐标, 放在 `plt.plot` 指令中即可实现。

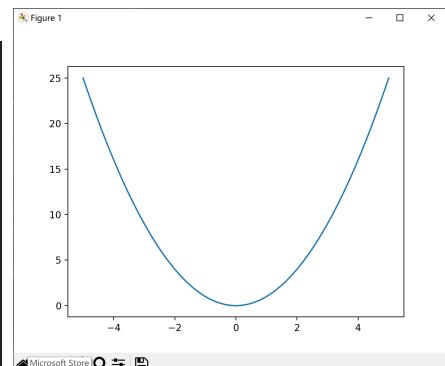
注意 `plot` 作为线图绘制的方法函数, 其本质是将所有离散的点以直线相连, 这一点在运用方法 2 的时候尤为明显, 而方法 1 由于可以指定取点密度, 只要我们将点密度设得足够大, 最后呈现的效果就是平滑的曲线 (方法 2 你总不可能手动输入 100 个点的坐标吧)。

下面我们以一个简单的函数 $y = x^2$ 为例, 展示这两个方法如何运用。

方法 1:

```
1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 x=np.linspace(-5, stop: 5, num: 100)
5 y=x**2
6 plt.plot(*args: x, y)
7 plt.show()
```

(a) 使用 `np.linspace`



(b) Gragh of $y = x^2$

`np.linspace(a,b,n)` 函数能够返回一个 numpy 数组, 其中 a 和 b 分别是这个数组的起始点与终止点, n 则是在这个区间内的取点数量, 通常取点数减一与区间长度的商如果是有限小数的话, 它的取点是等分的。(但我们大量取点拟合曲线的时候, 这件事并不重要)

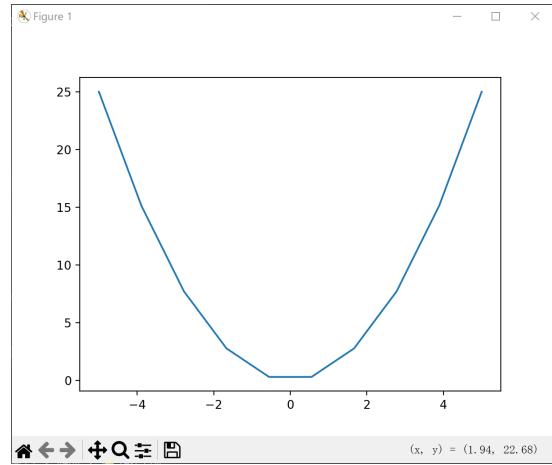
如果我们少取一些点，比如只取 10 个点，就可以明显看出 plot 实质上是以直线拟合曲线：

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 x=np.linspace(-5, stop: 5, num: 10)
5 y=x**2
6 plt.plot(*args: x,y)
7 plt.show()

```

(a) 使用 np.linspace，但是只有 10 个点



(b) Graph

plot 只能实现以直线连接各点，如果我们想要让各点以平滑曲线相连，而不采用暴力增加点数的方法呢？方法也是有的，那就是用多项式拟合，其中涉及到对 scipy.fsolve 等函数的使用，我们放在“详述” 8.1 中细说。

在这里要注意区分 numpy 数组和一般的 Python 列表，它们有许多区别和联系，对此的完整介绍我放在“详述” 8.2 部分，这里只说一点，就是 numpy 数组支持向量化计算，而列表不支持。向量化计算就是在背景知识——数学运算中提到的对一个数组进行单个数值形式的运算，输出一个由所有元素计算结果组成的新数组。简单来说，在 Python 中，array1+array2 是合法的，但是 list1+list2 是不合法的，同样的规则适用于其它单个数值基本运算。如图 3.1.5 所示，这段代码会报错。因为我们最后是通过当数值的运算构建由 x 到 y 的映射关系的，所以如果 x 是一个列表（或者元组），我们就无法对它（整个 x）直接进行运算操作，解决办法有两个，一是使用 for 循环把元素一一提取出来映射到 y，如图 3.1.6 所示，plot 函数似乎既可以接受 array 也可以接受 list 作为输入。

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 x=[-3,-2,-1,0,1,2,3]
5 y=x**2
6 plt.plot(*args: x,y)
7 plt.show()

```

(a) 3.1.5 错误示范

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 x=[-3,-2,-1,0,1,2,3]
5 y=[]
6 for i in x:
7     j = i ** 2
8     y.append(j)
9 plt.plot(*args: x,y)
10 plt.show()

```

(b) 3.1.6 正确示范

另一个办法是将列表转换为 numpy 数组形式，元组和列表均可以由以下方式转换：
`x=np.array([]) 或 x=np.array(())`。示例如下 3.1.7:

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 x=[-3,-2,-1,0,1,2,3]
5 x=np.array(x)
6 y=x**2
7 plt.plot(*args: x,y)
8 plt.show()

```

图 7: 3.1.7

方法 2

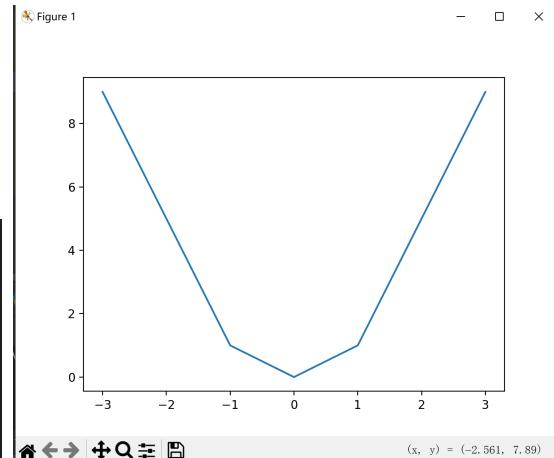
我们直接取几个整点，列出它们的 x,y 坐标，x,y 各列成一个数组的形式，然后直接丢给 plot 函数，plot 就可以把所以点以直线相连形成线图。如 3.1.8 和 3.1.9 所示：

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 plt.plot(*args: [-3,-1,0,1,3],[9,1,0,1,9])
5 plt.show()

```

(a) 3.1.8



(b) 3.1.9

除了接受点数据绘制折线图外，plot 函数还支持对线图的许多属性进行设定，包括颜色，粗细，实虚样式以及图例等等。

`plt.plot(x, y, color=' ', linewidth= , linestyle=' ', label=' ')`

举个例子，我如果希望把函数 $y = x^2$ 的图像设置为红色，线宽为 1.5，虚线，代码与效果如图所示：

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 x=np.linspace(-5, stop=5, num=100)
5 y=x**2
6 plt.plot(*args: x,y,color='red', linewidth=1.5, linestyle='--', label='$f(x)=x^2$')
7 plt.legend(loc='best')
8 plt.show()

```

图 9: 3.1.10

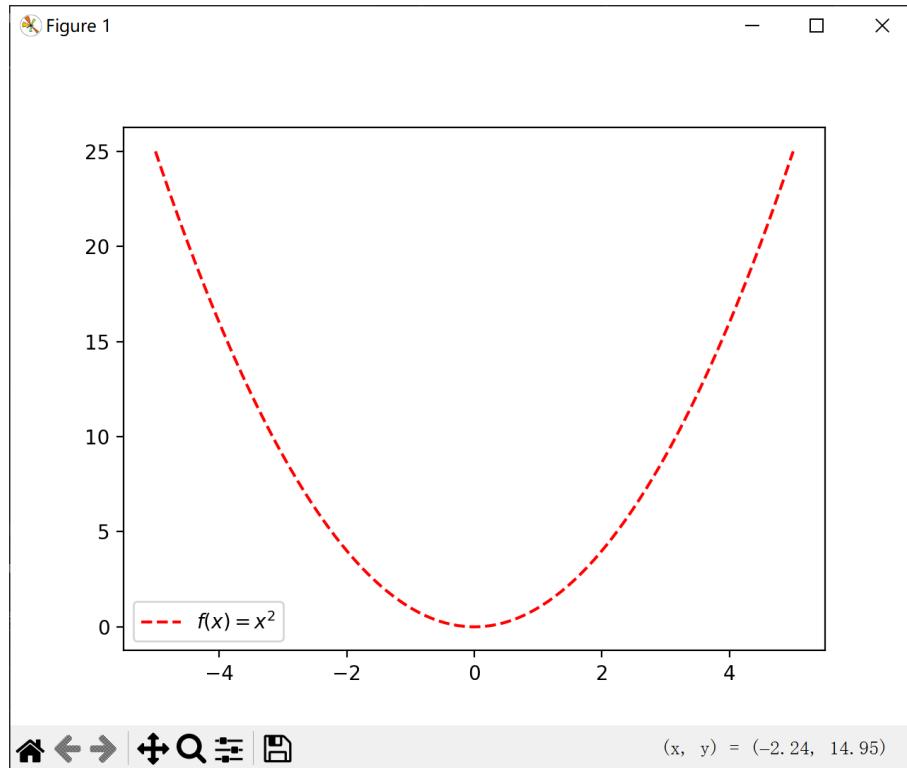


图 10: 3.1.11

对于 linestyle，尽管不同线形有相应的单词代称，比如实线是“solid”，但是多数时候我们更习惯用标点来指代线形，如下表所示：

实线	'solid'	,
虚线	'dashed'	,,
点线	'dotted'	.,
点划线	'dashdot'	-.,
无线条（仅显示标记）	'None'	,

对于图例，图例上显示的名字是在 plot 中的 label 设定的那个名字，然后调用图例函数 plt.legend() 即可生成图例，在图例函数中，可以通过 loc=' ' 来指定图例在图中的摆放

位置，比如 loc='best' 指将其摆放到系统认为最合适的位置，即图像最稀疏（对图像影响最小）的地方。legend 还可以进行其它的设定，放在“[8.3里讲](#)”。

3.1.2 点图

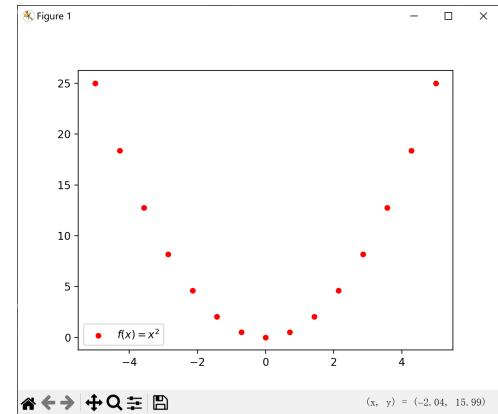
点图就是离散的点组成的图像，使用 scatter 函数，把在线图举的例子中的 plot 函数换成 scatter 函数，就可以生成一张 $y = x^2$ 点图。

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 x=np.linspace(-5, stop: 5, num: 15)
5 y=x**2
6 plt.scatter(x,y,color='red',s=20,label='$f(x)=x^2$')
7 plt.legend(loc='best')
8 plt.show()

```

(a) 3.2.1



(b) 3.2.2

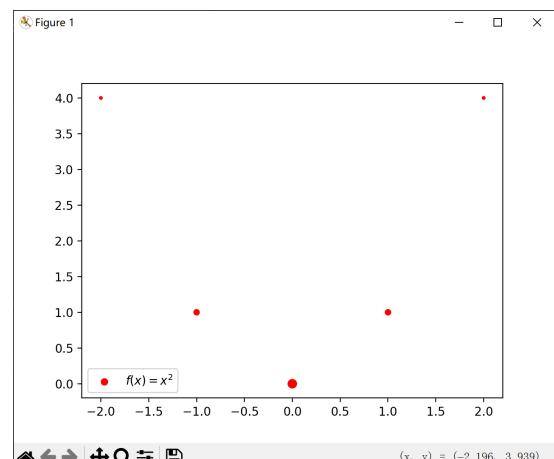
与 plot 相同的是，x,y 处都可以接受数组，列表，都可以设置颜色。不同的是，scatter 的 x,y 可以仅接受一个点坐标，而 plot 遇到这种情形则会报错。还有一个与线图属性的设置不同的地方，是点图没有所谓的线宽参数 linewidth，取而代之的是点宽参数 s，上图代码中的 $s = 20$ 就是将点宽设成了 20。（特别注意一点，这里的 s 不是 size 的简写!!! 如果在这里写 size 反而会报错，关于简写使用的问题在详述中会详细讨论 [8.5](#)），s 是一个数值时，这个数值指定了所有点的点宽；此外 s 还可以是一个数组或列表或元组，其中每个元素指定对应的点的大小，要注意的是它的元素数必须与 x,y 的元素数一样，否则会报错。

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 x=np.linspace(-2, stop: 2, num: 5)
5 y=x**2
6 size=(5,20,50,20,5)
7 plt.scatter(x,y,color='red',s=size,label='$f(x)=x^2$')
8 plt.legend(loc='best')
9 plt.show()

```

(a) 3.2.3



(b) 3.2.4

还有一个重要的不同，scatter 函数支持颜色，宽度，透明度作为数组传递给相应的参数，但是 plot 函数不支持，像下面这么写，在 scatter 里面没问题，但是在 plot 里面就会报错。

```
1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 import matplotlib.colors as ccolors
5
6 x=np.linspace(-5, stop=5, num=10)
7 y=x**2
8 plt.scatter(x,y,c=np.array(['r','m','k','g','b','r','m','k','g','b']),
9             alpha=[1,0.8,0.6,0.4,0.2,0.2,0.4,0.6,0.8,1],s=[5,15,25,40,60,60,45,25,15,5])
10 plt.show()
```

图 13: 3.2.5

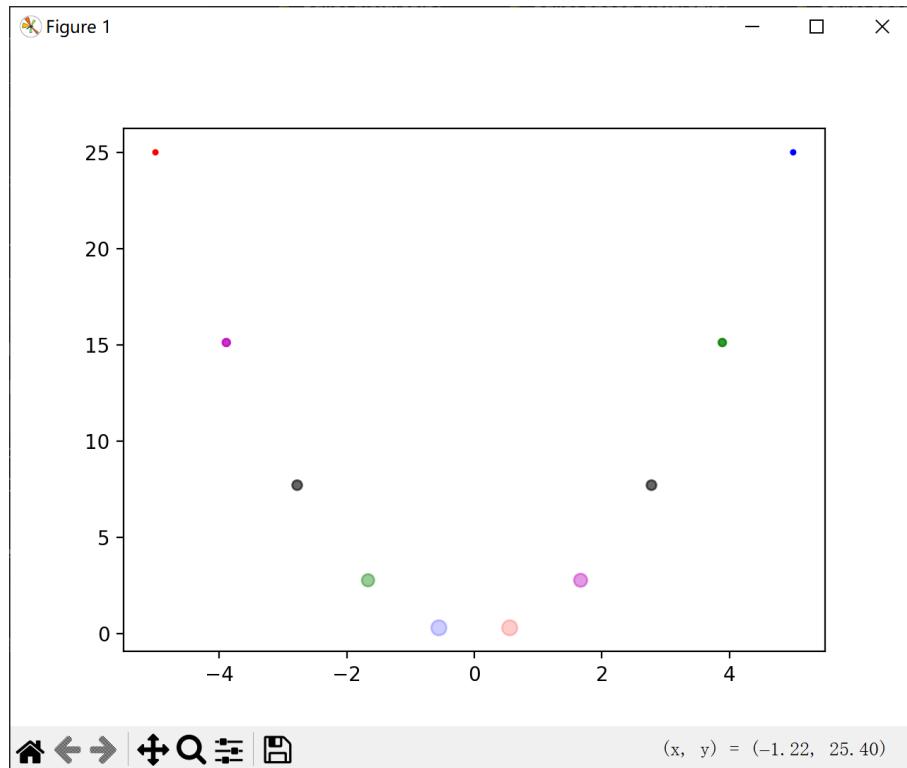


图 14: 3.2.6

(还要注意一件事，当我们向颜色传递数组类参数时，必须使用简写形式”`c=''`，“`c=`”，传递单个参数时则不需要，不要问我为什么。总而言之，color 参数用于设置统一的颜色，而 c 参数既用于根据数值数组来映射颜色，也用于在设置统一的颜色时，作为 color 的简写。)

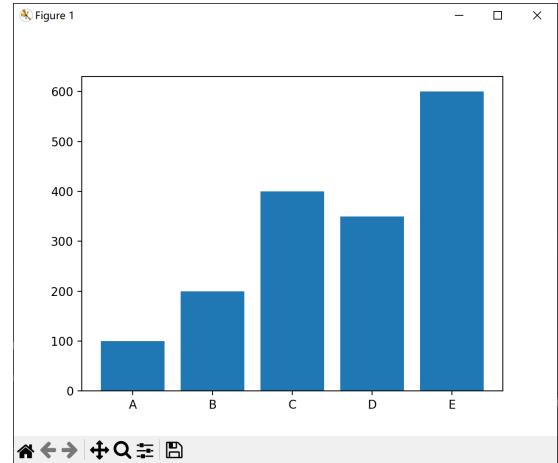
总之，scatter 的常用设置包括颜色，点宽，图例，其他高级的设置我们放到”详述“ 8.4。

3.1.3 条形图

条形图的核心由两要素组成，一是类别（x 轴），二是数值（y 轴），每个类别对应一个数值。因此绘制一张基本的条形图需要两个列表，分别存储类别和数值，然后类似 plot 函数的做法，将这两个列表传递给条形图函数 bar()。以下是一个示例：

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 categories=['A','B','C','D','E']
4 data=[100,200,400,350,600]
5 plt.bar(categories,data)
6 plt.show()
```

(a) 3.1.2.1



(b) 3.1.2.2

注意一件事，x 轴上的类别其实是有绝对宽度的，每两个类别的间距比如上图的 A-B 的间距是 1，而每个类别的条宽默认是 0.8，这个条宽是可以调整的，方法是 width=，数值应该在 (0, 1]，width=1 时所有条是紧挨的。

除了条宽，其它设置包括颜色、边缘颜色、透明度、图例、对齐方式、数值标签。颜色是 color，边缘颜色是 edgecolor，透明度是 alpha，图例是 label 加 legend 函数，这些都跟 plot 和 scatter 一样，不再赘述。下面重点介绍对齐方式和数值标签。

对齐方式

对齐方式指的是每个条形与 x 轴上的类别的相对位置，方法是 align=，align='center' 时是图 3.1.2.2 所示的中心对齐，因此这其实是默认的状态，我们还可以令 align='edge'，此时是边缘对齐，如图 3.1.2.3 所示：

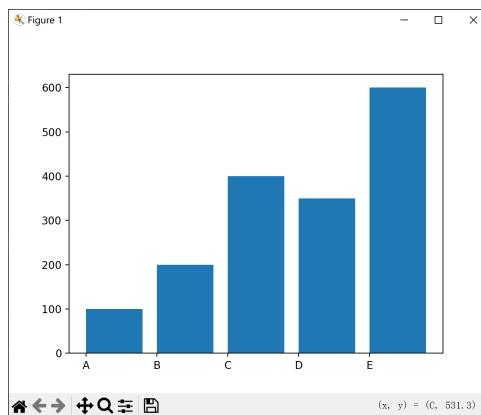


图 16: 3.1.2.3

(按理来说应该可以用 left 和 right 指定左右对齐的，但是不知道为什么在 Pycharm 里好像不行)

数值标签添加

有时我们需要把每一类的数值标在条形上方，这时我们需要 enumerate，它是一个很有用的 Python 内置函数，能同时返回可迭代对象中元素的索引值和数据值。其基本语法为 enumerate(可迭代对象, 起始索引)，其中起始索引不指定的话默认是 0，此时 enumerate 会从头开始遍历整个对象。enumerate 会返回一个枚举对象，这个对象可以被用来在循环中同时获取索引和值。这个对象是一个迭代器，可以被迭代。

下面是一个简单的示例，利用 enumerate 打印廿九高三五班前五位同学 + 一位班主任的学号和姓名字母：

```
1 A=['sy','yjl','fxc','wlr','lyl','wy']
2 for i,j in enumerate(A):
3     print('学号: {},姓名: {}'.format(*args: i,j))
4     # print(f'学号: {i},姓名: {j}')
```

(a) 3.1.2.3

```
C:\Users\金御风\PycharmPro
学号: 0,姓名: sy
学号: 1,姓名: yjl
学号: 2,姓名: fxc
学号: 3,姓名: wlr
... - . . . - -
```

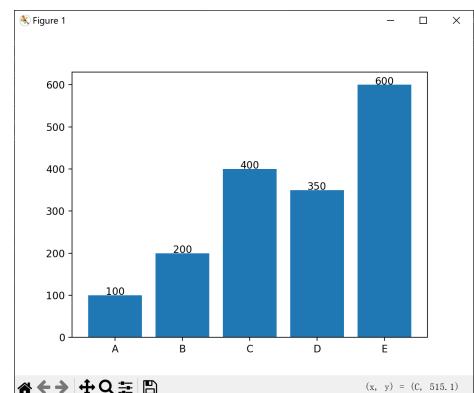
(b) 3.1.2.4

注意注释掉的 print 也是对的，这是两种格式化字符串的方法，详见[8.5.3](#)。第一个 print 中学号和姓名必须放在一个引号内，不能两个拼接，否则 format 只会对应后一个引号内的，以及注意 format() 前是”不是’,’!

下面来看如何利用 enumerate 添加数值标签。

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 categories=['A','B','C','D','E']
4 data=[100,200,400,350,600]
5 plt.bar(categories,data,width=0.8, align='center')
6 for i, v in enumerate(data):
7     plt.text(i, v + 0.5, str(v), ha='center')
8 plt.show()
```

(a) 3.1.2.5



(b) 3.1.2.6

哎呀差点忘了说了，我们还要用到 plt.text() 函数，它的前两个参数是 x,y 坐标，我们令 x=i 是因为不同类间距恰好默认为 1，而且第一类又是从 x=0 开始的（**这里要特别注意！！一般条形图原点是第一类标记处，不是图上看上去的横纵轴交点！**），这样每一

类的标签正好可以落在类的正上方（以及条形的中央，因为 `align='center'`），令 `v+0.5` 是为了让标签不与条形干涉，出现在稍靠上位置。第三个参数就是标签写些什么，这里是以数值本身作为标签，`ha` 是 `horizontalalignment` 的简称，设定了标签文本在前面指定坐标的什么相对位置出现，可以是`'left'`、`'center'`、`'right'`。

这个 `plt.text` 函数可以在图像的指定位置添加文本，是一个很有用的 `piplot` 函数，后面我们还要细讲它。[3.3.1](#)

水平条形图

将 `bar()` 改为 `barh()`，竖直条形图就变成了水平条形图，只需要做少许改动，首先是如果要设定宽度，参数是 `height` 而不是 `width`，还要就是使用 `plt.text` 函数添加数值标签时 `x,y` 和竖直是反着的，别忘了交换。

3.1.4 直方图

直方图和条形图很像，区别是条形图是明确知道分为几类，每一类对应的数值是多少；而直方图则是已知一堆杂乱的数据，通过指定划分为多少类，分类范围，将这些数据按照数值大小分类，统计落在每一类的数据个数（或比例），比如下图 3.1.4.1 就是将 60 颗子弹的出膛速度绘制成了 15 类的直方图，其程序如图 3.1.4.2。

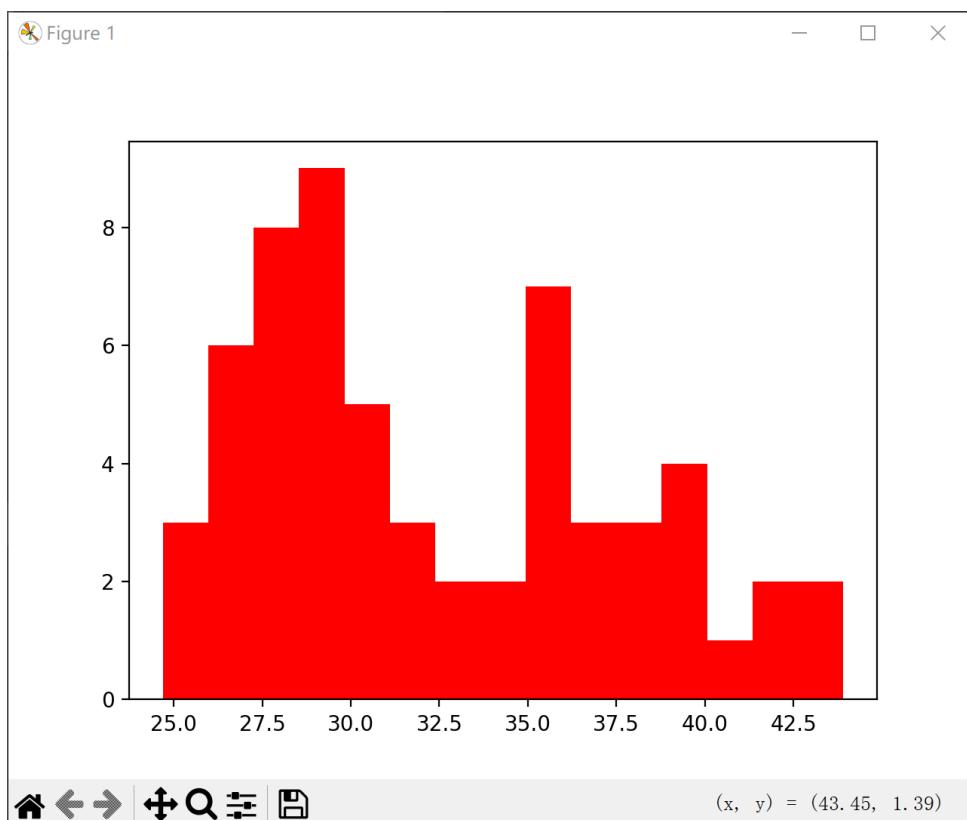


图 19: 3.1.4.1

```

import matplotlib.pyplot as plt
import numpy as np
speeds = [39.6, 28.6, 42.2, 35.4, 30.7, 28.7, 30.6, 29.4, 31.0, 28.5, 35.6, 37.0, 36.5, 24.7, 27.1,
          31.7, 27.4, 34.5, 36.0, 28.9, 27.6, 25.9, 39.4, 37.0, 27.8, 43.9, 38.6, 33.1, 30.7, 31.1,
          35.2, 39.6, 38.9, 29.7, 37.5, 26.3, 27.0, 25.2, 33.2, 28.2, 34.3, 40.8, 26.4, 36.1, 43.1,
          35.8, 26.5, 29.1, 37.7, 42.2, 27.2, 35.7, 28.9, 32, 27.3, 27.6, 30.0, 29.6, 29.7, 28.5]
plt.hist(speeds, bins=15, color='r')
plt.show()

```

图 20: 3.1.4.2

`plt.hist()` 函数接受的两个核心参数就是包含所有数据的列表（或一维 numpy 数组）和分的类数 `bins`。此外就是我们熟悉的一下基本设置：`color`, `edgecolor`, `alpha`, 不用多说了吧。

当然 `plt.hist()` 函数也有一些我们之前没提过的设置：

1. `range`: 数据的最小值和最大值，决定了直方图的横轴范围。
2. `density`: 如果为 `True`，则直方图的纵轴显示概率密度（就是频率啦）。如果为 `False`，则显示频数。通常默认为 `False`。
3. `histtype`: 直方图的类型，可以是'`bar`'、'`step`'、'`stepfilled`' 等。个人感觉 `bar` 和 `stepfilled` 没什么区别，而 `bar` 又是默认的状态，因此这里就展示一下图 3.1.4.1 以 `step` 的形式呈现是什么样子。

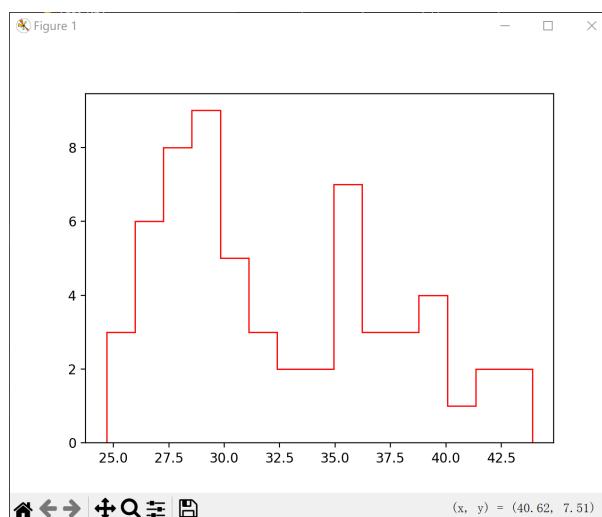


图 21: 3.1.4.3

4.`cumulative`: 为 `True` 时，显示的是每个 bin 加上所有之前 bin 的计数，相当于一个数据的逐渐累计，到最后一个 bin，其计数应是数据总数。相应的，如果 `cumulative=True` 并且 `density=True`，那么最后一个 bin 应该等于 1。这种方法可以用来观察数据的累积分布情况。比如图 3.1.4.1 以 `cumulative=1` 的形式呈现如下：

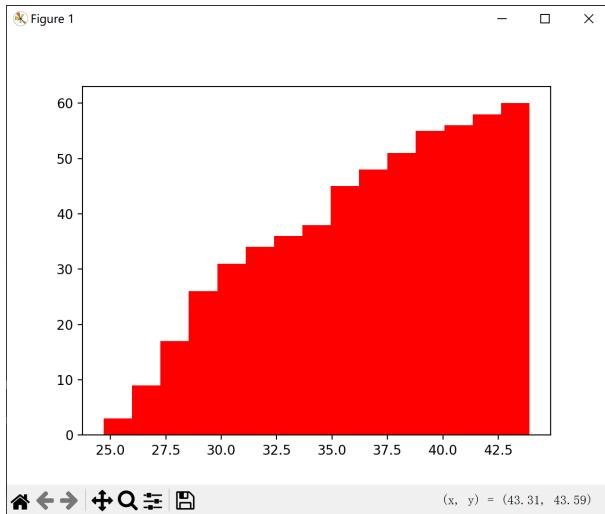


图 22: 3.1.4.4

除此之外，还有一些特殊的直方图，比如：

- **多组数据直方图:** 可以同时绘制多组数据的直方图，并设置不同的颜色和标签。

python

```
data1 = np.random.randn(1000)
data2 = np.random.randn(1000)
plt.hist([data1, data2], bins=30, label=['Group 1', 'Group 2'], color=['blue', 'green'])
plt.legend()
plt.show()
```

图 23: 3.1.4.5

如果使用 `stacked=True` 参数还可以将多组数据的直方图堆叠在一起，就形成了堆叠直方图。总之，可以向 `hist()` 传递不止一个列表，让多组数据在一张图中呈现。

Pyplot 绘图的基本方法到这里就结束了，其他特殊的 Pyplot 绘图方法放在[8.6](#) 中简要介绍。

3.2 Pyplot 设置坐标轴的基本方法

除了绘图，Pyplot 还有一些设置坐标轴的方法，相比 Axes 设置坐标轴的方法，Pyplot 的方法更简单，更基础，但是实现的功能也较为有限。

3.2.1 xlim(), ylim() 函数

如果我们令 plt.xlim((-1,2)), 那么图像的定义域就会被限定在 (-1, 2) 区间内, ylim 同理, 通常我们用元组形式表示这个限定区间 (所以出现了双括号), 但是其实 plt.xlim(-1,2) 也是可以的。

3.2.2 xticks, yticks 函数

这是一个很有用的函数, 能对坐标轴进行很多方面的设置。以 xticks 为例, 其可接受的参数有:

plt.xticks(ticks, labels, minor, kwargs)

其中 ticks 是对刻度的设置, 注意它与 xlim() 的区别! ticks 对坐标轴的限制仅是对有刻度区域的限制, 以及对刻度形式的设置, 其数据类型可以是 numpy 数组, 列表及元组。比方说令 ticks=[-2,-1,0,1,2], 那么 x 轴虽然还是 xlim() 或者 x=np.linspace 规定的范围, 但是只有 -2, -1, 0, 1, 2 这五个地方会有刻度。(我们也有时用 np.linspace 生成 ticks) labels 应当是一个长度和 ticks 相同的列表或什么别的, 它指定了那些标出来的刻度叫什么名字, 因而需要与 ticks 中元素一一对应。

minor 暂时没有发现有什么有用的功能, 其作用似乎是: minor=True 时, 之前设置的 ticks 与 label 会被忽略, minor=False 时则会正常显现设置效果。

这个 kwargs 其实是一些额外参数的统称, 常用的包括 color, fontsize (字体大小), rotation (字体旋转角度) 等等。

3.2.3 xlabel, ylabel 函数

用法就是 plt.xlabel(' '), plt.ylabel(' '), 用来指定 x,y 坐标轴的名字。

3.2.4 grid 函数

用法是 plt.grid(True) 或者 plt.grid(False), True 的时候会根据刻度划分生成网格。

3.2.5 title 函数

这是用来设置图表标题的函数, 其用法如下:

plt.title(label, fontdict, loc, pad)

label 用于指定图表的标题内容, 是必须提供的参数。

fontdict 用于设置标题的字体样式, 如字体大小、字体颜色、字体类型等。它是一个字典, 因此形如: fontdict={'fontsize': 14, 'color': 'red', 'fontweight': 'bold'} 可以设置标题

的字体大小为 14，颜色为红色，字体加粗。'red' 这里可以简写，然后'fontweight' 的可选值有：

1. 'light' : 较细的字体
2. 'normal' : 正常粗细的字体，也是默认值
3. 'medium' : 中等粗细的字体
4. 'semibold' : 半粗体
5. 'bold' : 粗体
6. 'heavy' : 较粗的字体
7. 'black' : 最粗的字体

loc 是标题的位置，可选值有'center'（居中，默认值）、'left'（左对齐）、'right'（右对齐），用于控制标题在图表顶部的水平位置。

pad 是标题与图表顶部的距离，单位为点数（1 点等于 1/72 英寸）。（个人觉得这个参数没什么用）

pyplot 还有一个功能与之很相似的函数 plt.subtitle()，它们的区别我们放在[3.3.2](#)里讲。

3.2.6 axis 函数

plt.axis() 虽然有很多参数，但是最好一个函数只接受一个参数，所以得分成多个 plt.axis() 来进行设置。其包含的参数有：

1. plt.axis([xmin, xmax, ymin, ymax]) : 直接指定 x 轴和 y 轴的显示范围，其中 xmin 和 xmax 分别表示 x 轴的最小值和最大值，ymin 和 ymax 分别表示 y 轴的最小值和最大值。
2. plt.axis('off') : 关闭坐标轴的显示，使图表中不显示坐标轴。其默认状态是 on，一旦被设成 'off'，与坐标轴有关的其他设置也将不复存在。效果如下图：

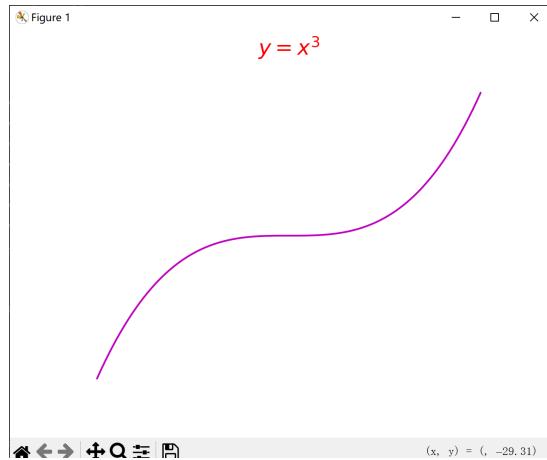


图 24: 3.2.6.1

3. plt.axis('equal') : 设置坐标轴的刻度间隔相等。
4. plt.axis('scaled') : 设置坐标轴的刻度间隔相等，并根据图表大小自动调整范围。
5. plt.axis('tight') : 自动调整坐标轴范围，使数据紧密显示。

下面我们看一个融合了以上方法的示例：

```
import matplotlib.pyplot as plt
import numpy as np
fig,ax=plt.subplots()
x=np.linspace(-3, 3, num=100)
t=x**3
plt.plot(x,t,c='m')
plt.xticks(ticks=[-3,-1,0,1,2], labels=['t1','t2','t3','t4','t5'], minor=0,
           color='blue', fontsize=12, rotation=45)
plt.xlabel('axis x')
plt.ylabel('axis y')
plt.grid(True)
plt.axis([-3,1,5,-6,6])
plt.title(label='$y=x^3$', fontdict={'fontsize':18,'color':'r',
                                         'fontweight':'bold'}, loc='center', pad=20)
plt.show()
```

图 25: 3.2.2.1

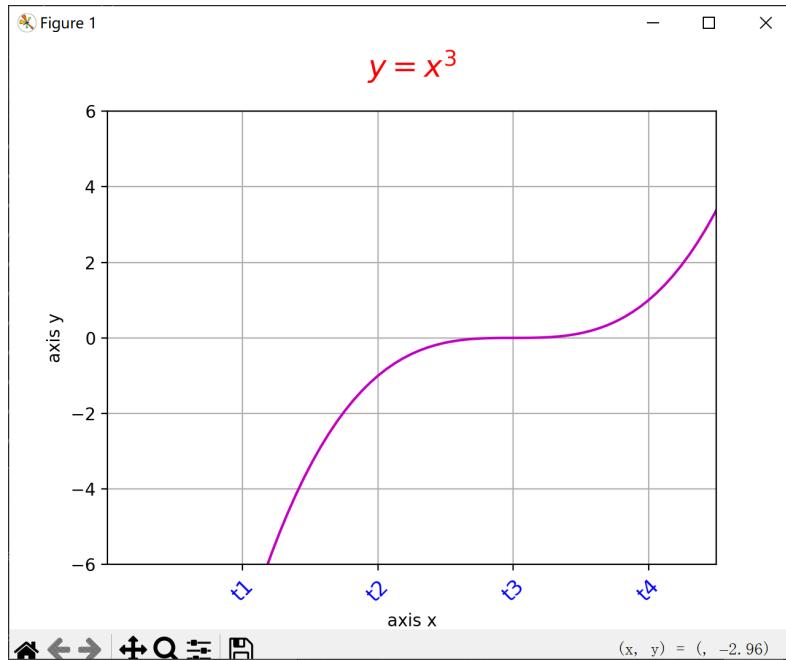


图 26: 3.2.2.2

3.3 在图中添加文本

Pyplot 中添加文本的函数主要有 `text`, `xlabel` 和 `ylabel`,`annotate`,`title` 和 `subtitle`, 其中 `title` 和 `xlabel`、`ylabel` 之前已经讲过, 这里重点介绍另外几个。

3.3.1 `text` 函数

`text` 的基本参数就是 x,y 坐标和内容, 这我们之前就提到过, 下面介绍其更复杂的设置。位置设置

`text` 函数设置文本位置的方法不止 x,y 坐标一种, 我们来看一个不用坐标的例子:

```
plt.text(0.5, 0.5, 'Center of Axes', transform=ax.transAxes)
```

在这个例子中出现了一个新的参数 `transform`, 它用来指定设置位置所用的坐标系, 或者说采用的参考方式, 常用的坐标系有:

1. `ax.transData` : 数据坐标系 (默认值), 就是在子图中的 x,y 坐标。
2. `ax.transAxes` : 轴坐标系, 范围是 0 到 1, 其中 $(0,0)$ 是轴的左下角, $(1,1)$ 是轴的右上角, 就是在子图中的相对位置, 或者说分别相对于 x,y 坐标轴的位置。
3. `fig.transFigure` : 图形坐标系, 范围也是 0 到 1, 是相对于整个图形窗口的位置。

因此, 上述例子中是将文本放置在了两轴的中心位置。

(注意前面有 ax. 或 fig., 所以别忘了提前创建坐标轴或图形对象)

文本内容设置

内容其实没什么好说的，就说两点：一是如果文本不止一行，可以使用换行符\n，例如 plt.text(1, 2, 'Line 1\nLine 2') 还有一点就是 text 以及 matplotlib 其它添加文本的方式一般都是添加英文文本的，如果要添加中文的话，最简单的做法是在前面添上两行代码：

```
plt.rcParams['font.sans-serif'] = ['SimHei']
```

```
plt.rcParams['axes.unicode_minus'] = False
```

(不用导入别的什么包，看来 pyplot 还是太全面了)

字体和样式设置

1. fontsize: 设置文本的字体大小。可以是数值，也可以是预定义的大小名称，如'small'、'medium'、'large' 等。
2. fontweight: 设置字体的粗细。常见的值有'normal'、'bold'、'light' 等。
3. 例如， plt.text(1, 2, 'Bold Text', fontweight='bold')。
4. fontstyle: 设置字体的风格。常见的值有'normal'、'italic'、'oblique' 等。
5. color: 设置文本的颜色。可以是颜色名称（如'red'、'blue' 等），也可以是十六进制颜色代码（如'#FF0000'）。
6. family: 设置字体家族。常见的值有'serif'、'sans-serif'、'cursive'、'fantasy'、'monospace' 等。

这个字体家族控制的是文本的总体风格，可能有些难以理解，下面用以下代码展示不同字体家族的效果：

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3], [4, 5, 6])
# Serif 字体
plt.text(1, 4, 'Serif Font', family='serif', fontsize=14, color='red')
# Sans-serif 字体
plt.text(1, 5, 'Sans-serif Font', family='sans-serif', fontsize=14, color='blue')
# Cursive 字体
plt.text(1, 6, 'Cursive Font', family='cursive', fontsize=14, color='green')
# Fantasy 字体
plt.text(2, 4, 'Fantasy Font', family='fantasy', fontsize=14, color='purple')
# Monospace 字体
plt.text(2, 5, 'Monospace Font', family='monospace', fontsize=14, color='orange')

plt.show()
```

图 27: 3.3.1.1

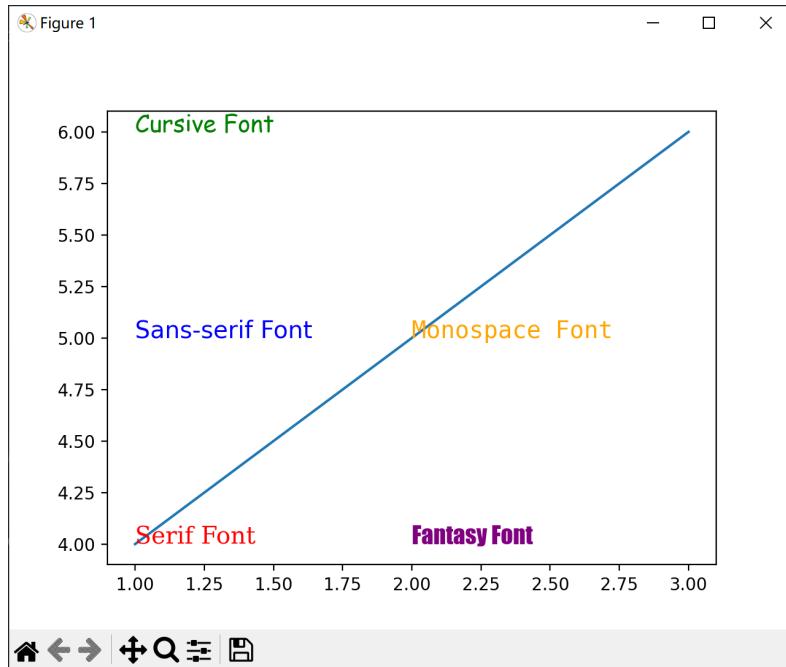


图 28: 3.3.1.2

3.3.2 subtitle 函数以及它和 title 的区别

二者最本质的区别是 title 用于设置单个子图（axes）的标题，而 subtitle 用于设置整个图形对象（fig）的总标题。对于多子图的 fig，title 设置的标题仅对一个子图有效，不会影响其他子图；而无论图形中有多少个子图，suptitle() 添加的标题都只有一行，它覆盖整个图形的顶部区域。

二者的设置参数都一样，参考[3.2.5](#)，只不过因为是总标题，subtitle 字体的默认大小会大一些。

3.3.3 annotate 函数

annotate 函数的功能是在图中添加带箭头的注释文本，其用法和主要参数如下：

```
plt.annotate(' ', xy = (), xytext = (), arrowprops = {'arrowstyle' : ' ', 'color' : ' ', 'lw' : })
```

第一个引号内容为注释文本，xy 接受的是箭头指向的坐标元组，xytext 接受的是注释文本所在位置的坐标（默认似乎是左对齐），arrowprops 接受一个与箭头属性有关的字典，我们可以在这个字典中设置箭头的样式、颜色、宽度等。颜色和宽度就不用多说了，样式可以有以下选项：

1. '-'：没有箭头，仅一条线。
2. '->'：简单的箭头，箭头指向右侧。

3. '<-': 简单的箭头，箭头指向左侧。
4. '<->': 双向箭头。
5. '-[': 方括号箭头，箭头指向右侧。
6. ']-': 方括号箭头，箭头指向左侧。
7. '-|>': 带有垂直线的箭头，箭头指向右侧。
8. '<-|>': 带有垂直线的双向箭头。
9. 'fancy': 花哨的箭头，箭头指向右侧。
- 10 'simple': 简单的箭头，箭头指向右侧。
11. 'wedge': 楔形箭头，箭头指向右侧。

我们用如下程序展示这 11 种箭头：

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 theta=360/11
4 plt.xlim( *args: -0.75,0.75)
5 plt.ylim( *args: -0.75,0.75)
6 r1=0.5
7 r2=0.1
8 plt.scatter( x: 0, y: 0,s=15,c='g')
9 plt.text( x: 0, y: 0, s: '(0,0)')
10 arrow_types_list=[ '->', '<->', '<->[ ]', '-|>', '<-|>', 'fancy', 'simple', 'wedge']
11 for i in range(11):
12     plt.annotate( text: f'{arrow_types_list[i]}',xy=(r2*np.cos(theta*i),r2*np.sin(theta*i)),xytext=
13     (r1*np.cos(theta*i),r1*np.sin(theta*i)),arrowprops={'arrowstyle':arrow_types_list[i],'color':'blue'})
14 plt.show()

```

图 29: 3.3.3.1

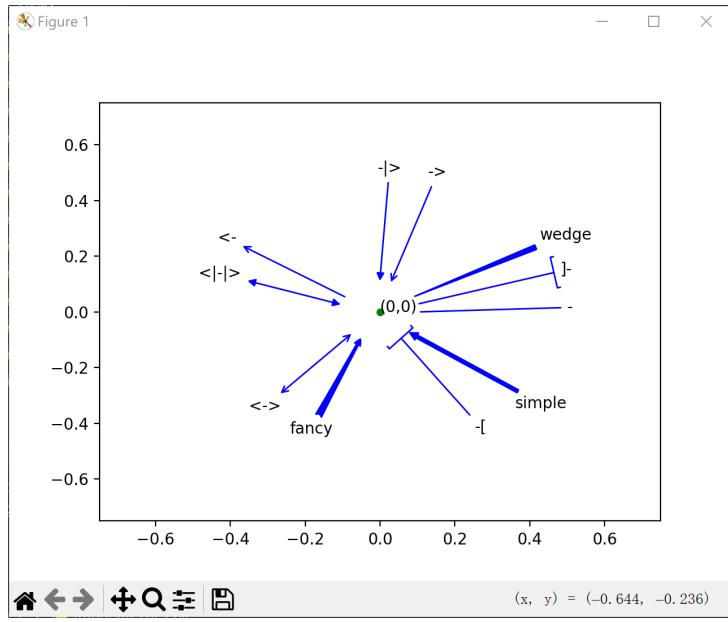


图 30: 3.3.3.2

4 Axes 的基本用法

4.1 创建 Axes 对象

创建 Axes 对象（活跃坐标轴）的方法有一下几种，其中有一些还是 pyplot 的方法，我们会以在一个图形窗口绘制四张子图，分别是对数函数、指数函数、正弦函数，余弦函数为例，展示如何使用直线方法：

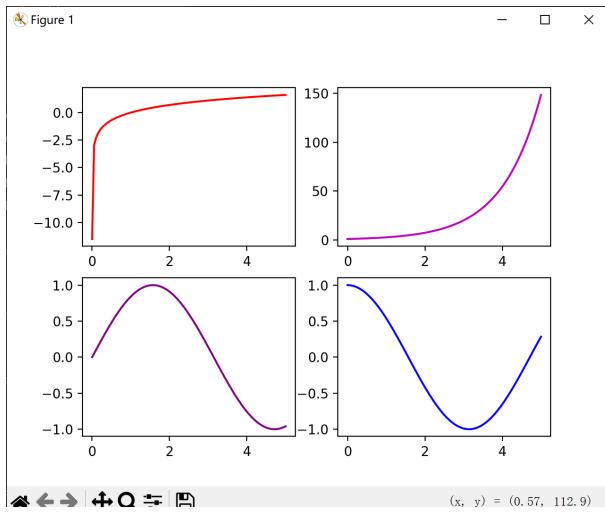


图 31: 4.1.1

1. plt.subplot:

```
plt.subplot(nrows, ncols, index)
```

nrows, ncols 是指这个方法将图形窗口分割成 nrows 行 ncols 列的网格，index 则是对这个网格的索引，在 index 指定的位置创建一个子图。它的返回值是 ax (一个子图)。

如果我们要在一个图形窗口绘制四张子图，分别是对数函数、指数函数、正弦函数，余弦函数，用 plt.subplot 方法是这样的：

```
import matplotlib.pyplot as plt
import numpy as np
axe1=plt.subplot(*args: 2,2,1)
axe2=plt.subplot(*args: 2,2,2)
axe3=plt.subplot(*args: 2,2,3)
axe4=plt.subplot(*args: 2,2,4)
x=np.linspace(start: 0.00001, stop: 5, num: 100)
y1=np.log(x)
y2=np.exp(x)
y3=np.sin(x)
y4=np.cos(x)
axe1.plot(*args: x,y1,c='r')
axe2.plot(*args: x,y2,c='m')
axe3.plot(*args: x,y3,c='purple')
axe4.plot(*args: x,y4,c='b')
plt.show()
```

图 32: 4.1.2

你也许会觉得这段代码很冗长，但是借助列表与遍历，我们可以对其进行简化：

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(start: 0.00001, stop: 5, num: 100)
y1=np.log(x)
y2=np.exp(x)
y3=np.sin(x)
y4=np.cos(x)
y_list=[y1,y2,y3,y4]
color=['r','m','purple','b']
for i in range(1,5):
    plt.subplot(*args: 2,2,i).plot(*args: x,y_list[i-1],c=color[i-1])
plt.show()
```

图 33: 4.1.3

在 `plt.subplot(2,2,i).plot(x,y_list[i-1],c=color[i-1])`，我直接对刚刚创建出来的子图进行绘图操作，这种做法的可行，也佐证了我们之前提到的，pyplot 方法支持链式调用。

2. plt.subplots:

```
fig, ax = plt.subplots(nrows, ncols)
```

和 plt.subplot 不同的是, (注意! 它们只有一个 s 之差!!) subplots 不指定索引, 因为它的返回值是图形对象 fig 和四个子图的集合, 这个集合是一个二维数组 (当 nrows 和 ncols 均大于一时), 我们可以用 [i,j] 的形式来索引它, 从而指定在不同位置的子图应该画什么图像。如果我们只要画一幅子图, 我们也可以直接令 fig, ax = plt.subplots(), 这样默认只创建应该子图, 非常简便。

在这个示例 (4.1.4) 中, 我作了两幅内容一样的图, 区别是第二段代码明显更为简约, 同时在第二段代码中我令 sharex 和 sharey 均等于 True, 指的是令 x 和 y 轴都共享, 这样的效果如下图所示:

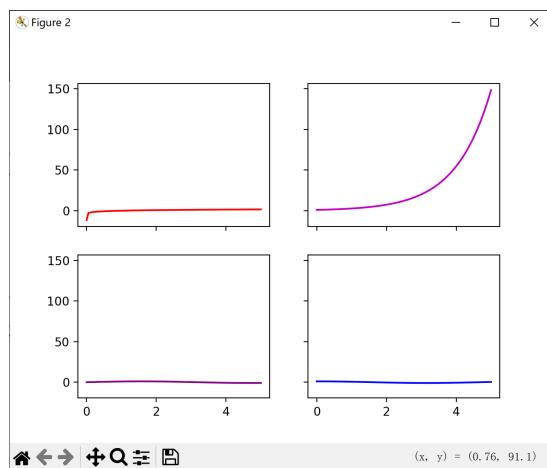


图 34: 4.1.5

是不是感觉很难看? 所以首先我们一般不会一下共享两个坐标轴, 其次共享坐标轴是为了方便比较不同子图中数据的大小关系, 一般不会用到它。

```

import matplotlib.pyplot as plt
import numpy as np
fig1, axes=plt.subplots( nrows= 2, ncols= 2, sharex=False, sharey=False)
x=np.linspace( start= 0.00001, stop= 5, num= 100)
y1=np.log(x)
y2=np.exp(x)
y3=np.sin(x)
y4=np.cos(x)
axes[0,0].plot(x,y1,c='r')
axes[0,1].plot(x,y2,c='m')
axes[1,0].plot(x,y3,c='purple')
axes[1,1].plot(x,y4,c='b')
# plt.show()
fig2, axes=plt.subplots( nrows= 2, ncols= 2, sharex=True, sharey=True)
y_list=[y1,y2,y3,y4]
color_list=['r','m','purple','b']
for i in range(4):
    axes.flatten()[i].plot(x,y_list[i],color_list[i])
plt.show()

```

图 35: 4.1.4

(还有一个很无聊的，你可注意可不注意的点，我在中间注释掉了一个 plt.show()，这时两幅图是一起出现的，分别为 Figure1 和 Figure2，但如果不注释掉，它会先出现 Figure1，当你又掉它后（专业一点说，终止掉 Figure1 的运行），Figure2 才会出现（开始执行）)

3. add_subplot:

fig.add_subplot(nrows, ncols, index)

这个方法是先创建图形对象 fig，然后在这个图形对象基础上在 index 指定位置创建子图。要论它与方法 1 plt.subplot() 的区别，就是方法 1 更便捷快速，通常用于快速创建单个子图，而 fig.add_subplot 允许你已经在已经创建的图形对象上添加子图。

4. add_axes:

fig.add_axes([left, bottom, width, height])

与 add_subplot 相似的是，它也是先创建图形对象 fig，然后在指定位置添加子图，不同的是，这个位置是通过 left, bottom, width, height 来指定的。add_axes() 方法的参数是一个包含四个值的列表或元组，分别表示 Axes 对象的左下角的 x 坐标、y 坐标以及宽度和高度，这些值都是相对于图形窗口的大小的比例值（范围在 0 到 1 之间）。也就是说，它允许你指定 Axes 对象的确切位置和大小。

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace( start: 0.00001, stop: 5, num: 100)
y1=np.log(x)
y2=np.exp(x)
y3=np.sin(x)
y4=np.cos(x)
fig=plt.figure()
fig.add_axes([0,0.5,0.5,0.5]).plot(*args: x,y1)
fig.add_axes([0.5,0.5,0.5,0.5]).plot(*args: x,y2)
fig.add_axes([0.5,0,0.5,0.5]).plot(*args: x,y3)
fig.add_axes([0,0,0.5,0.5]).plot(*args: x,y4)
plt.show()
```

图 36: 4.1.5

我们之前说过，创建图形对象最简单的方式是 `plt.figure()`，但是使用 `axes` 方法基本上都是 `ax.+` 函数的模式，就是说，你得先有函数可以作用的 `ax`，也就是子图（坐标轴），这就需要用到上面介绍的四种创建子图的方法。比较这四种方法，我们发现：方法 3 和方法 4 都需要提前创建图形对象 `fig`，而方法 1 和方法 2 都不需要，其中方法 2 是将图形对象 `fig` 和子图 `ax` 一起创建了，而方法 1 是完全不需要创建图形对象（毕竟人家是 `plt` 的方法嘛，在无图形对象的情况下创建子图，系统会自动生成一个图形对象）。总之我想说的是，如果需要用到 `axes` 的方法，我们需要结合实际需要，选择合适的创建子图的方法。

下面谈谈我自己的一些思考：我认为方法 3 是可以首先舍弃的，因为其功能和方法 1 其实完全一致，而没有方法 1 简便，因为它有依赖于图形对象的局限性。你也许想问，这个图形对象重要吗，一定要创建它吗；well，可以说在大多数时候，它是没什么用的，整段代码根本不需要它的存在；但是在少数时候它是必要的，比如说需要调整图形对象的长宽（就是生成出来的图像的大小），还有后面会将的事件处理与交互。所以，如果不着创建图形对象，我们可以放心大胆地用方法 1，如果子图又只有一张，那甚至可以效仿方法 2 的做法，直接简写为 `ax=plt.subplot()`，很简单吧。如果需要图形对象的话，我们通常可以采用方法 2，将图形对象 `fig` 和子图 `ax` 一起创建了，即 `fig,ax=plt.subplots()`。方法 4 较为复杂，而且同样需要先创建 `figure`，我认为其唯一用武之地就是需要专门设置不同子图在一张 `figure` 中的位置和大小的情况，这一点其他 3 个方法都代替不了。

差点忘了说了，跳过 `figure` 直接创建 `ax` 其实还有一种方法，`ax=plt.gca()`。（怎么样够简单吧）

4.2 简述 Pyplot 和 Axes 的区别

Pyplot 和 Axes 的方法有许多是重合的，其唯一区别就是把 plt 换成了 ax(或者别的名字，看你怎么定义的)，如下表所示：

类型	Pyplot	Axes	功能
常规绘图	plot(x, y, ...)	plot(x, y, ...)	线图
	scatter(x, y, ...)	scatter(x, y, ...)	点图
	bar(x, height, ...)	bar(x, height, ...)	垂直条形图
	barh(y, width, ...)	barh(y, width, ...)	水平条形图
	hist(x, bins, ...)	hist(x, bins, ...)	直方图
	boxplot(x, ...)	boxplot(x, ...)	箱型图
	fill_between(x, y1, y2, ...)	fill_between(x, y1, y2, ...)	两曲线间填充颜色
	step(x, y, ...)	step(x, y, ...)	阶梯图
	stem(x, y, ...)	stem(x, y, ...)	茎叶图
	pie(x, ...)	pie(x, ...)	饼图
文本和注解	text(x, y, s, ...)	text(x, y, s, ...)	在图表中添加文本
	annotate(xy, xytext, ...)	annotate(xy, xytext, ...)	添加注解
轴比例和旋转	tick_params(axis, ...)	tick_params(axis, ...)	刻度参数, 如旋转角度
图例和框架	legend()	legend()	添加图例
	grid()	grid()	添加网格线

其不同之处可参考下表：

类型	Pyplot	Axes	功能
轴和刻度	xlim(left, right)	set_xlim(left, right)	设置 x 轴的范围
	ylim(bottom, top)	set_ylim(bottom, top)	设置 y 轴的范围
	xticks(ticks, ...)	set_xticks(ticks)	设置 x 轴的刻度位置
		set_xticklabels(labels)	设置 x 轴的刻度标签
	yticks(ticks, ...)	set_yticks(ticks)	设置 y 轴的刻度位置
		set_yticklabels(labels)	设置 y 轴的刻度标签
轴标签和标题	xlabel(label)	set_xlabel(label)	设置 x 轴的标签
	ylabel(label)	set_ylabel(label)	设置 y 轴的标签
	title(label)	set_title(label)	设置图表的标题
轴比例和旋转	axis('equal'/'scaled')	set_aspect(aspect)	设置坐标轴的纵横比
	\	axis('on'/'off')	显示或隐藏轴

除了以上这些，就是 pyplot 主要用于静态绘图，不支持复杂的事件处理。而 axes 可以通过连接事件来实现交互式绘图。

4.3 用 Axes 进行坐标轴的设置

有一些功能与 Pyplot 重合的方法我就不再细说了，这里主要介绍一些 Axes 能做，而 Pyplot 做不了的精细设置。

4.3.1 反转坐标轴方向

`ax.invert_xaxis()`：反转 x 轴的方向。例如，x 轴原本是从左到右递增，调用此方法后变为从左到右递减。同样，`ax.invert_yaxis()` 可以反转 y 轴的方向。

4.3.2 控制刻度显示

通过 `ax.minorticks_on()` 命令可以在轴上显示较小的刻度，效果如下图，默认较小刻度是不显示的，即 `minorticks_off()` 状态。

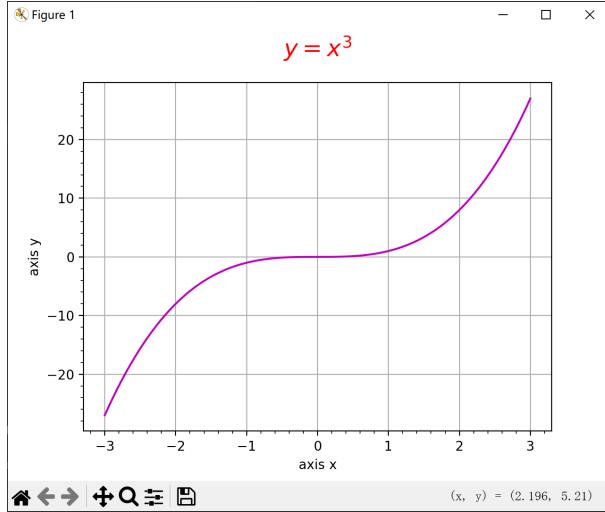


图 37: 4.3.2.1

4.3.3 再谈网格线 grid

我们知道，网格线是基于坐标轴上的刻度的，那么如何让添加的较小刻度也形成网格线呢？这就涉及到之前没有说的 `grid` 函数的更多设置。其主要设置如下：

`grid(True, which=' ', axis=' ')`

其中 `which` 的可选值有 `major`, `minor` 和 `both`, `major` 代表仅主要刻度形成网格线，`minor` 表示仅添加的较小刻度形成网格线，`both` 表示所有刻度均形成网格线。

`axis` 的可选值包括 `x,y` 和 `both`。`x` 表示仅 `x` 轴刻度形成网格线，即栅栏形（?），`y` 同理，而 `both` 就是默认的状态，`x,y` 均形成网格线。举个例子，`grid(True, which='both', axis='x')` 的效果如下图：

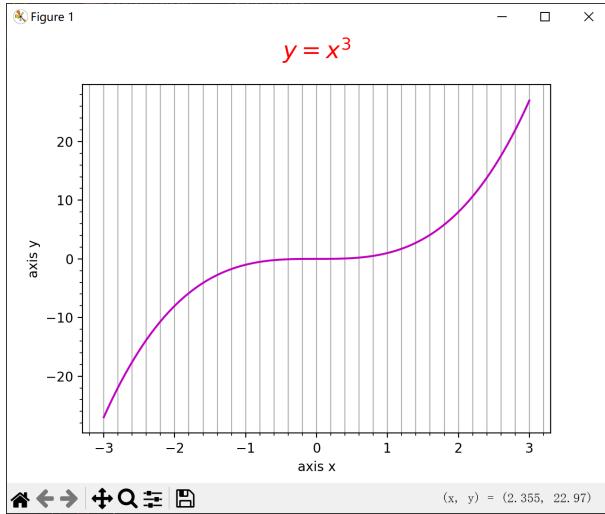


图 38: 4.3.3.1

4.3.4 操控四条边框

当我们已经把四条边框赋值给 `ax`, (比如通过 `ax = plt.gca()`) 我们就可以通过 `ax.spines['top']/'bottom'/right'/left']` 指定其中的任何一条边框进行操作。`'top'`, `'bottom'`, `'right'`, `'left'` 分别指上下左右四条边框。

我们可以以此设定边框的颜色,通过 `set_color`,比如 `ax.spines['right'].set_color('red')` 就是把右边框设为红色。还可以设置 x,y 轴相交的位置,通过 `set_position`,比如 `ax.spines['bottom'].set_position(0)` 即可将交界点设为在 $y=a$ 的位置, x 轴同理。最后我们甚至还可以设置边框宽度,通过 `set_linewidth()`。(要注意的是 `Axes` 方法不能链式调用,就是说 `ax.spines[]` 后面只能跟一个命令,别的命令就得另起一段了)

(顺带一提: 如果 `set_color('None')`, 那么颜色就是透明,也就是让边框不显示。)

如果我们要一次改变四条边框的某个参数比如宽度,一一指定未免太过麻烦,我们可以使用这样一个循环:

```
for spine in ax.spines.values():
    spine.set_linewidth(2)
```

`ax.spines` 是一个字典,包含四条边框,而 `ax.spines.values()` 是一个能返回使用边框的函数,因此我们可以用它来遍历所有边框。

在默认情况下,下边框和左边框分别是 x,y 轴,如果我们要设置其他边框为坐标轴,比如令上边框为 x 轴,右边框为 y 轴,可以使用如下命令:

```
ax.xaxis.set_ticks_position('top')
```

```
ax.yaxis.set_ticks_position('right')
```

此外，x,y 轴的交界处也是可以设置的，如果要让 x 轴位于 y=a 位置，也就是交界点为 (0, a) , 可以使用如下命令：

```
ax.spines['bottom'].set_position(('data', a))
```

看到这行命令的你可能会有诸多疑问，比如这个双括号是什么，'data' 又是什么。首先，`set_position()` 函数接受一个元组作为输入，因此出现了这样的“双括号”；这个元组的第一个元素'data' 是一个字符串参数，它指定了接下来以什么形式指定坐标轴的位置，这个位置可以接受的字符串参数有：（以下统一称边框为“脊柱”）

1. 'outward' : 将脊柱向外移动一定的距离。默认情况下，距离为 0，即脊柱紧贴坐标轴。例如 `ax.spines['right'].set_position(('outward', 10))` 表示将右侧脊柱向外移动 10 个单位。可以通过设置 `set_smart_bounds` 方法来调整距离。

2. 'axes' : 将脊柱的位置设置为坐标轴的相对位置，取值范围为 0 到 1。例如，('axes', 0.5) 表示脊柱位于坐标轴的中间。

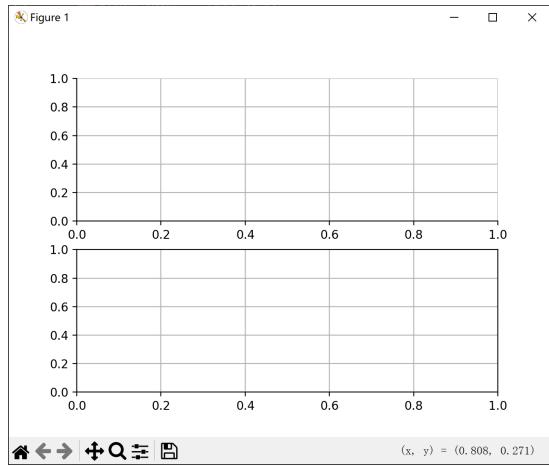
3. 'data' : 将脊柱的位置设置为数据坐标系中的某个位置。例如，('data', 3) 表示脊柱位于数据坐标系中的 x=3 或 y=3 处，如果指定的脊柱是左脊柱或右脊柱，则是 x=3，反之则是 y=3。

个人认为在大多数时候，'data' 法是最好用的，因此接下来重点介绍它。

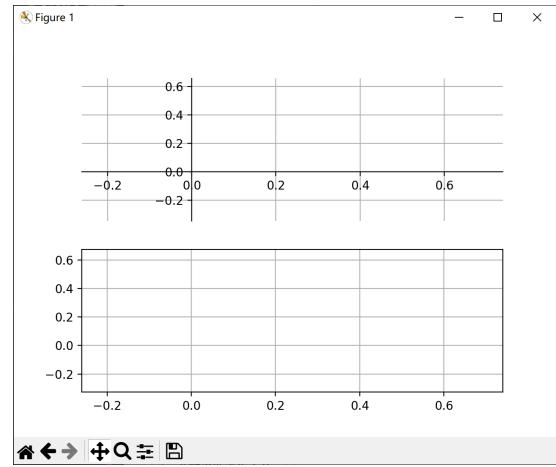
讲到这里，必须要澄清一个很容易误会的点，Python 在将子图绘制出来的时候，代表 x,y 的两脊柱相交在 (0, 0) 处，但我们并不能说默认两脊柱相交在原点处，事实上，在你人为设定交点前，不存在任何脊柱的交点，如果你点击下方十字箭头标志，然后拖动图像，你会发现交界点固定在原地，不会随原点移动，但是如果我们将增加这样一个交界点的设置，情况就正好相反，我们用如下程序进行演示：

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 ax1=plt.subplot(211)
4 ax2=plt.subplot(212)
5 ax1.grid(True)
6 ax2.grid(1)
7 ax1.spines['bottom'].set_position(('data', 0))
8 ax1.spines['left'].set_position(('data', 0))
9 ax1.spines['top'].set_color('None')
10 ax1.spines['right'].set_color('None')
11 plt.show()
```

图 39: 4.3.4.1



(a) 4.3.4.2 拖动前



(b) 4.3.4.3 拖动后

4.3.5 其他

还有一些零碎的，没那么重要的设置，这里简单提一下：

设置轴矩形补丁：

`set_frame_on(True/False)` 设置是否绘制轴矩形补丁，默认是 True。这个轴矩形补丁其实就是边框，如果为 False，四条边框就会被隐藏。

设置轴刻度线和网格线位置：

`set_axisbelow(True/False)` 设置轴刻度线和网格线是在图上方还是下方, True 则将刻度线和网格线设置在图下方，反之则在上方。

5 极坐标系

5.1 极坐标系简介

```
import matplotlib.pyplot as plt
import numpy as np
theta=np.linspace( start: 0, np.pi*2, num: 100)
r=1-np.sin(theta)
plt.polar( *args: theta, r, c='r')
plt.show()
```

图 41: 5.1.1.1

极坐标系的函数是 plt.polar(), 与 plot 函数相似, polar 先接受两个确定二维平面内点位置的数据, 然后接受其他参数的设置, 我们知道, 在极坐标系中, 确定位置的两个量是角度和距离, 我们通常设定为 theta 和 r。然后, 类似在直角坐标系中的做法, 我们利用 np.linspace 对自变量 theta 进行指定区间内的大量取点, 在示例中就是取满 0~360 度, 然后列出 r 关于 theta 的关系式就可以了, 是不是跟直角坐标系中的 plot 函数非常相似呢? (要注意的是传递给 polar 函数时是角度在前, 距离在后, 这也不难理解, 因为角度是自变量嘛)

上面这段代码绘制出的, 就是著名的“笛卡尔心形线”了, 效果如下:

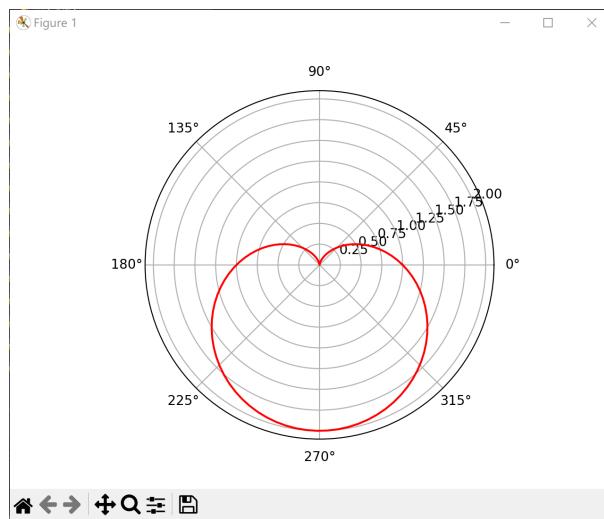


图 42: 5.1.1.2

就这张图, 讲讲极坐标系图的几个基本概念。周围一圈角度值叫角度刻度, 内部一环环上标的数值叫半径刻度 (我有时候喜欢称之为“圆周刻度”), 你会发现, 极坐标系的角度和半径可以分别对应直角坐标系的 x 和 y 轴, 事实上, matplotlib 也是这样设定的,

后面会讲到的用于设置极坐标系刻度的函数分别是 `set_xticklabels()`,`set_yticklabels()`, 这就是把 `xticklabels` 视为角度刻度, 把 `yticklabels` 视为半径刻度。

5.2 极坐标系的复杂设置

下面介绍 `polar` 函数一些更复杂的设置:

标记点样式相关参数:

1. `marker`: 标记点样式, 如' o ' 表示圆形标记点, '*' 表示星号标记点等。
2. `markersize(ms)`: 标记点的大小, 数值越大, 标记点越大。
3. `markerfacecolor(mfc)`: 标记点的填充颜色, 可以是颜色名称字符串 (如' red ')、十六进制颜色代码 (如'#FF0000') 等。
4. `markeredgecolor(mec)`: 标记点的边界颜色, 设置方法与 `markerfacecolor` 类似。

线条样式相关参数:

老生常谈的 `color`, `linestyle`, `linewidth(lw)`, `label` 和 `alpha`。

坐标轴相关:

极坐标系的坐标轴方法有一些和直角坐标系是相同的, 比如创建子图的方法; 还有 `plt.grid()` (或者 `ax.grid()`) 可以用 `Ture`, `False` 控制网格的有无, 唯一的区别是在极坐标系中, 网格是默认为“有”的; 最后 `plt.title()` 依然可以添加标题或其他文字。

其他的方法则稍有区别, 比如在直角坐标系中设置坐标轴刻度范围和位置的函数是 `xticks` 和 `yticks`, 在极坐标系中则是 `set_thetagrids()` 设置角度, `set_rgrids()` 设置半径 (注意这两是 `Axes` 方法啊! 不是 `plt`), 通常做法是向它们传递一个确定始末点和步长的 `numpy` 数组, 然后再传递一个长度相同的列表 `labels=[]` 确定这些刻度的名称。

如果要改变所有刻度的某个参数或依次指定, 该怎么办呢? 以角度刻度和颜色参数为例, 我们可以用 `get_xticklabels` 函数获取所有的角度标签 (刻度), 赋值给一个变量:

```
labels = ax1.get_xticklabels()
```

然后利用 `zip`, 在一个循环中改变所有的角度刻度:

```
for label, color in zip(labels, colors):
```

```
    label.set_color(color)
```

`colors` 是一个颜色列表, 其长度一个等于 `labels` 长度, 通过这两个列表元素的一一对应,

我们实现了对每个刻度颜色的指定，当然如果我希望所有刻度但是一个颜色，我们可以无视对 colors 的遍历，直接指定 set_color 中的 color 是什么颜色。如果 color 被指定为 ‘None’，那么所有刻度就被抹除了。

但是要注意的是虽然我无视了这个颜色列表，这个列表依然必须存在。如果希望彻底摆脱它，我们可以省去 colors 和 zip 函数，直接对 labels 遍历。

结合这些方法和我们前面所学的方法，我们可以在“笛卡尔心形线”基础上整出非常多的活：

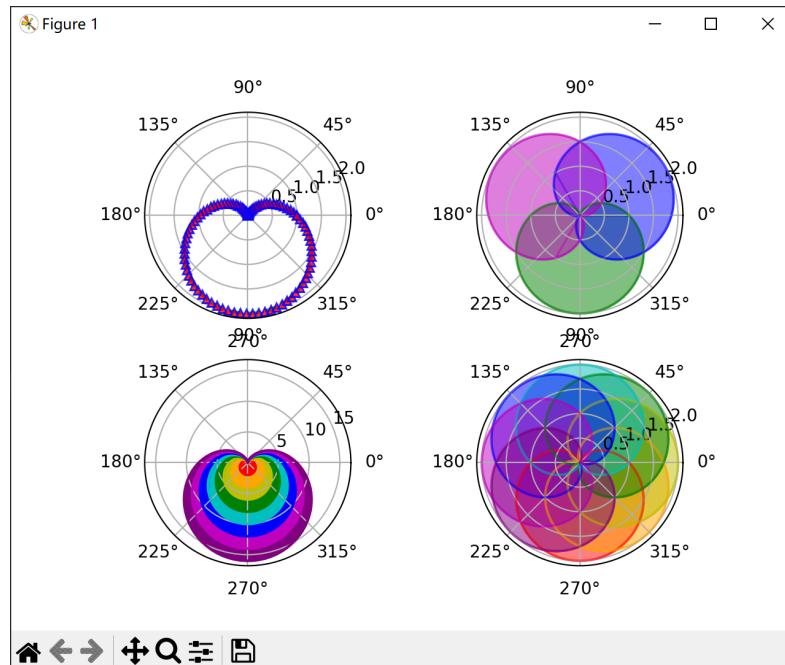


图 43: 5.2.1.1

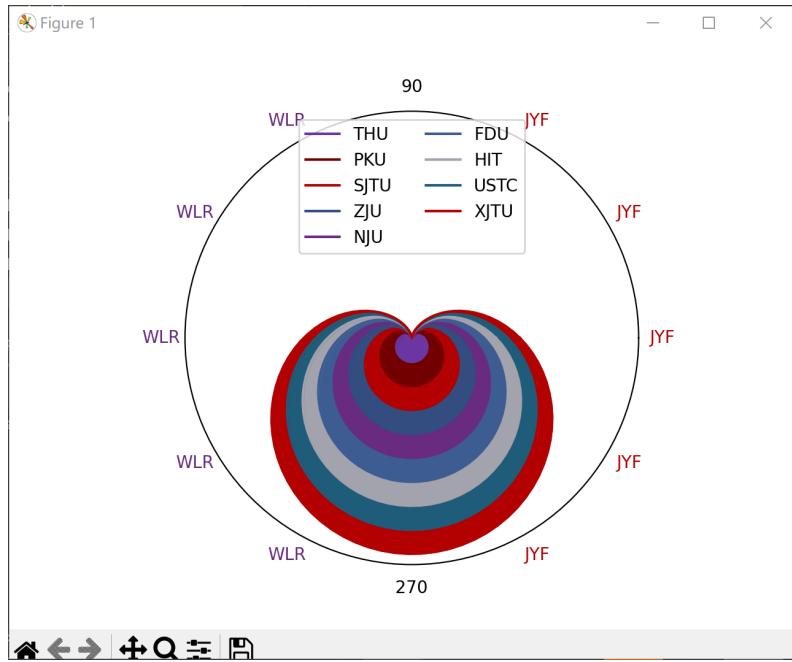


图 44: 5.2.1.2

请读者思考一下如何实现这些效果。

图 5.2.1.1 代码：

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 theta=np.linspace( start: 0,np.pi*2, num: 100)
4 r1=1-np.sin(theta)
5 r2=0
6 ax1=plt.subplot( *args: 221,projection='polar')
7 ax2=plt.subplot( *args: 222,projection='polar')
8 ax3=plt.subplot( *args: 223,projection='polar')
9 ax4=plt.subplot( *args: 224,projection='polar') # 分别创建四个子图
10 ax1.plot( *args: theta, r1, marker='^', ms=5, mfc='red',
11             mec='blue', color='green', linestyle='--', linewidth=2,
12             label='示例曲线', alpha=0.8) # 子图1
13 colorlist_1=['g','b','m']
14 for i in np.linspace( start: 0, stop: 2, num: 3):
15     ax2.plot( *args: theta+i*np.pi*2/3, r1, c=colorlist_1[int(i)],alpha=0.5)
16     ax2.fill_between(theta+i*np.pi*2/3,r1,r2, color=colorlist_1[int(i)],alpha=0.5) # 子图2
17 colorlist_2=['r','orange','y','g','c','b','m','purple']
18 for i in np.linspace( start: 0, stop: 7, num: 8):
19     ax3.plot( *args: theta, r1*(i+1), c=colorlist_2[int(i)],alpha=1,label='whatsername')
20     ax3.fill_between(theta,r1*i,r1*(i+1), color=colorlist_2[int(i)],alpha=1) # 子图3
21 for i in np.linspace( start: 0, stop: 7, num: 8):
22     ax4.plot( *args: theta+i*np.pi/4, r1, c=colorlist_2[int(i)],alpha=0.5,label='whatsername')
23     ax4.fill_between(theta+i*np.pi/4,r1,r2, color=colorlist_2[int(i)],alpha=0.5) # 子图4
24 plt.show()
```

图 45: 5.2.1.3

这里需要解释一个之前没有提到的点，如果我们需要创建多子图，就必须要使用 Axes 方法进行绘图，因为我们需要指定各个 ax 绘制不同的图像。但是 Axes 方法并没有极坐标系函数，就是说没有 ax.polar() 这个东西。那么我们就需要用另一种方法建立极坐标系，在 plt.subplot() 中指定 projection='polar'。这个 projection 参数我们在 3D 图绘制还会遇到它，有关它的详细介绍我们放在详述8.8。

图 5.2.1.2 代码：

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 thu_color=(0.42,0.21,0.63)
4 nju_color=(0.41,0.17,0.50)
5 pku_color=(0.45,0,0)
6 sjtu_color=(0.7,0,0)
7 ustc_color=(0.12,0.36,0.48)
8 zju_color=(0.2,0.3,0.5)
9 hit_color=(0.64,0.64,0.68)
10 fdu_color=(0.24,0.36,0.57)
11 xjtu_color=(0.7,0,0) # c9高校颜色一览
12 theta=np.linspace( start: 0,np.pi*2, num: 100)
13 r1=1-np.sin(theta)
14 r2=0
15 ax1=plt.subplot(*args: 111,projection='polar')
16 label_list=['THU','PKU','SJTU','ZJU','NJUSTU','FDU','HIT','USTC','XJTU']
17 colorlist_2=[thu_color,pku_color,sjtu_color,zju_color,nju_color,
18             fdu_color,hit_color,ustc_color,xjtu_color]
19 for i in np.linspace( start: 0, stop: 8, num: 9):
20     ax1.plot(*args: theta, r1*(i+1), c=colorlist_2[int(i)],alpha=1,label=label_list[int(i)])
21     ax1.fill_between(theta,r1*i,r1*(i+1), color=colorlist_2[int(i)],alpha=1)
```

图 46: 5.2.1.4

```
22 ax1.set_thetagrids(np.degrees(np.linspace( start: 0, 2*np.pi, num: 12, endpoint=False)),
23                     labels=[ 'JYF','JYF','JYF','90','WLR','WLR','WLR','WLR','270','JYF','JYF'])
24 # 获取所有角度标签
25 labels = ax1.get_xticklabels()
26 colors = [sjtu_color,sjtu_color,sjtu_color,'k',nju_color,nju_color,nju_color,nju_color,
27             nju_color,'k',sjtu_color,sjtu_color]
28 for label, color in zip(labels, colors):
29     label.set_color(color)
30 labels2 = ax1.get_yticklabels()
31 for label, color in zip(labels2, colors):
32     label.set_color('None') # 去掉圆周标签
33 ax1.grid(False) # 去掉网格
34 plt.legend(loc='upper center',ncol=2)
35 plt.show()
```

图 47: 5.2.1.4 续

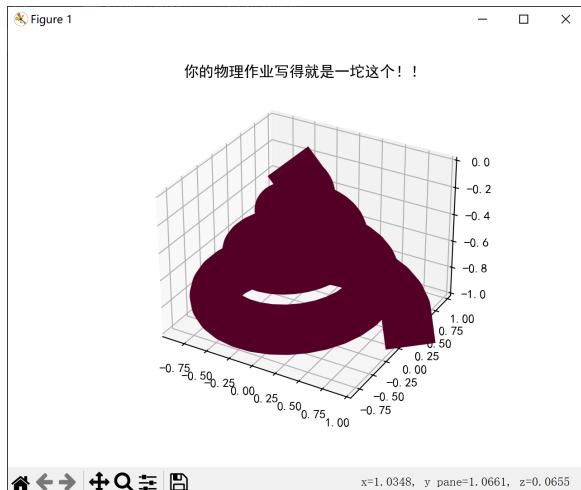
(顺带一提，图 5.2.1.2 所示代码中的颜色元组是下一个 section 要说的 RGB 模式自定义颜色，其数值是本人试验了一个晚上，与各高校校徽颜色反复比对出来的最佳配比，只有 HIT 是依据个人喜好设定的。手动 [doge])

6 3D 图像

6.1 3D 图总览

3D 图像，本质上就是比平面图多了一个变量，多了一个维度，根据 matplotlib 所能呈现出的效果，我们可以大致将 3D 图像分为 3D 线图、3D 散点图、3D 曲面图、3D 线框图、3D 等高线图这几类。下面我们分别展示这几类图是什么效果：

3D 线图

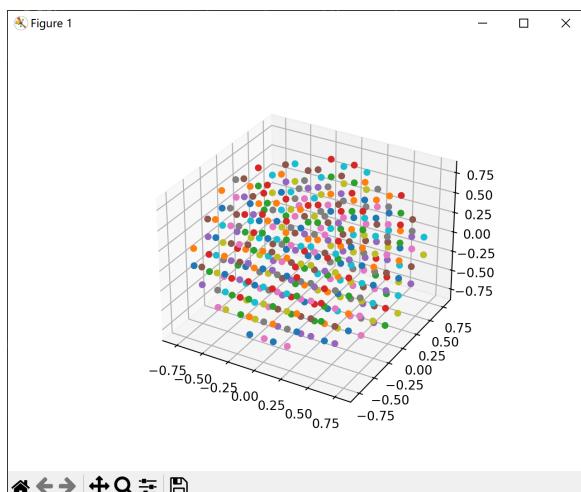


(a) 6.1.1.1

```
import matplotlib.pyplot as plt
import numpy as np
ax=plt.subplot(*args: 111, projection='3d')
z = np.linspace(start: 0, stop: 1, num: 100)
x = z * np.sin(20 * z)
y = z * np.cos(20 * z)
z=-z
color=(0.33, 0, 0.15)
ax.plot(*args: x, y, z, c=color, lw=40)
plt.rcParams['font.sans-serif'] = ['SimHei'] # 设置中文字体为黑体
plt.rcParams['axes.unicode_minus'] = False # 解决负号显示问题
plt.title('你的物理作业写得就是一坨这个！！')
plt.show()
```

(b) 6.1.1.2

3D 散点图

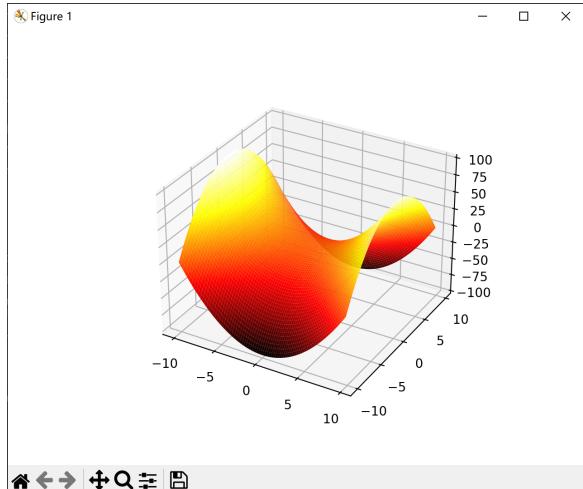


(a) 6.1.1.3

```
import matplotlib.pyplot as plt
import numpy as np
ax=plt.subplot(*args: 111, projection='3d')
for x in np.linspace(-1, stop: 1, num: 10):
    for y in np.linspace(-1, stop: 1, num: 10):
        for z in np.linspace(-1, stop: 1, num: 10):
            if x**2+y**2+z**2<=1:
                ax.scatter(x, y, z)
plt.show()
```

(b) 6.1.1.4

3D 曲面图

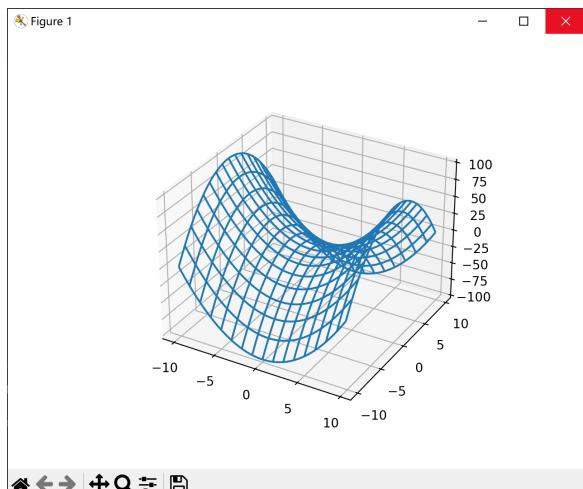


(a) 6.1.1.5

```
import matplotlib.pyplot as plt
import numpy as np
ax1=plt.subplot( *args: 111,projection='3d')
x = np.arange(-10, 10, 0.25)
y = np.arange(-10, 10, 0.25)
x, y = np.meshgrid( *xi: x, y)
z1=x**2-y**2
ax1.plot_surface(x, y, z1, rstride=1, cstride=1, cmap='hot')
plt.show()
```

(b) 6.1.1.6

3D 线框图

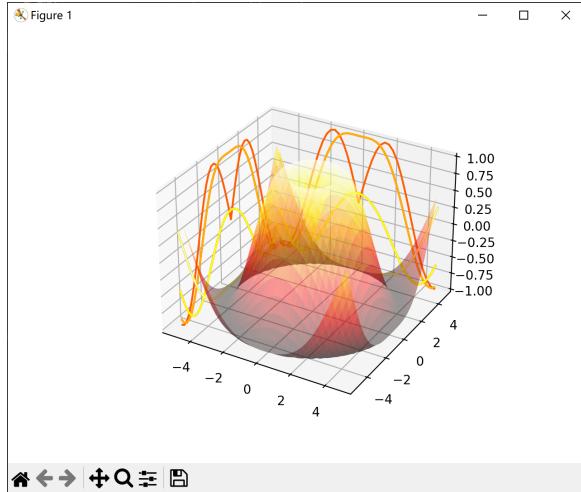


(a) 6.1.1.7

```
import matplotlib.pyplot as plt
import numpy as np
ax1=plt.subplot( *args: 111,projection='3d')
x = np.arange(-10, 10, 0.25)
y = np.arange(-10, 10, 0.25)
x, y = np.meshgrid( *xi: x, y)
z1=x**2-y**2
ax1.plot_wireframe(x, y, z1, rstride=5, cstride=5, cmap='hot')
plt.show()
```

(b) 6.1.1.8

3D 等高线图



(a) 6.1.1.9

```
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax1 = fig.add_subplot(111, projection='3d')
x = np.arange(-5, 5, 0.25)
y = np.arange(-5, 5, 0.25)
x, y = np.meshgrid(x, y)
r = np.sqrt(x**2 + y**2)
z = np.sin(r)
ax1.contour(*args: x, y, z, zdir='x', offset=-5, cmap='hot')
ax1.contour(*args: x, y, z, zdir='y', offset=5, cmap='hot')
ax1.plot_surface(x,y,z,cmap='hot',alpha=0.4)
plt.show()
```

(b) 6.1.1.10

6.2 浅析 3D 图

6.2.1 3D 空间的创建

要绘制 3D 图像，首先需要创建一个 3D 绘图区域，即 Axes3D 对象，同创建极坐标系的方法一样，在 plt.subplot 或 fig.add_subplot 函数中令 projection=‘3d’，有三件事要注意：一是有些版本的 matplotlib 支持在 gca 函数中令 projection=‘3d’，有些则不支持，个人建议是尽量别用这个方法了；二是我们令 projection=‘3d’ 时其实是在使用 Axes3D 类，在比较高版本 Matplotlib 里，Axes3D 已经被集成到 Matplotlib 的核心模块中，因此不需要显式导入它，否则需要在开头进行导入：

```
from mpl_toolkits.mplot3d import Axes3D
```

三是注意 projection=‘3d’ 中，‘d’ 是小写！！

与极坐标系图不同，3D 图像不存在可以代替设置 projection 的 pyplot 方法，对于那些专门绘制特殊 3D 图像如曲面图、等高线图的方法，它们只能在 projection=‘3d’ 的情况下被使用，否则会直接报错。

6.2.2 3D 图像再划分——何时要 meshgrid

我们先简单回忆一下平面图像是如何作出来的，首先通过 $x=np.linspace()$, x 大量取点，一一映射到 y , 从而确定平面上每一个点的坐标，现在对于三维坐标系，自变量可以是一个，也可以是两个，这两种情况实际上很大程度上决定了后面可以用哪些 3D 方法以及要不要使用 meshgrid。

变量只有一个的比较简单，比如 z 是自变量， x 和 y 都是它的因变量，这相当于将

平面直角坐标系的因变量轴拆出了两个轴，但是它们依然都是受 z 控制的。下面是一个单变量的例子：

```
import numpy as np
import matplotlib.pyplot as plt
ax=plt.subplot(projection='3d')
z=np.linspace(start: 0, stop: 50, num: 500)
x=np.sin(z)
y=np.cos(z)
ax.plot(*args: x, y, z)
plt.show()
```

图 53: 6.2.2.1

因为我们分别用 z 计算（定义）了 x 和 y, z 的定义是一个一维 numpy 数组，因此 x 和 y 也是一维数组，类型相同，我们就可以把它们一起丢进 plot 进行作图了。最终效果如下，可见垂直 z 轴任何一个平面截图像，永远只能截出一个点。

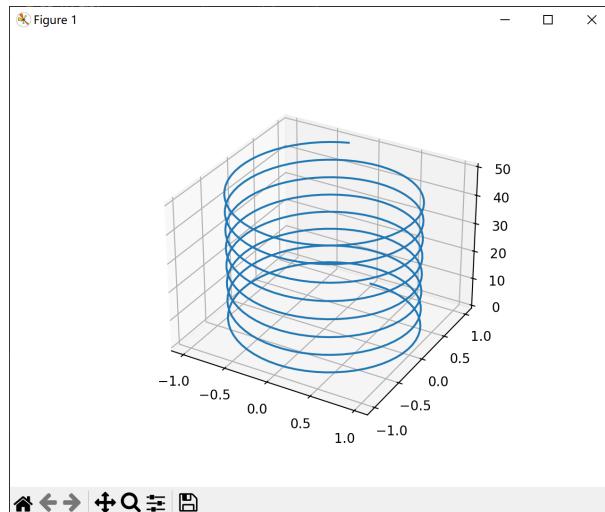


图 54: 6.2.2.2

如果变量有两个，比如令 $x=np.linspace(a,b,n)$, $y=np.linspace(c,d,m)$, $z=f(x,y)$, 此时我们就不能贸然直接计算 z 了。因为我们说 x,y 两个变量时，我们希望的是 x,y 这个平面作为自变量，平面上的每个点对应一个 z 值，但是两个数组直接运算时，如果它们长度不同，即 $m \neq n$, 那么程序会直接报错，因为长度不同的两个数组是不能直接运算的；如果它们长度相同，即 $m = n$, 那么元素会一一对应进行运算，最后生成一个由每一对元素计算结果组成的，长度一样的新一维数组，这显然不是我们想要的，呃，或者说大多数时候不会是我们想要的。

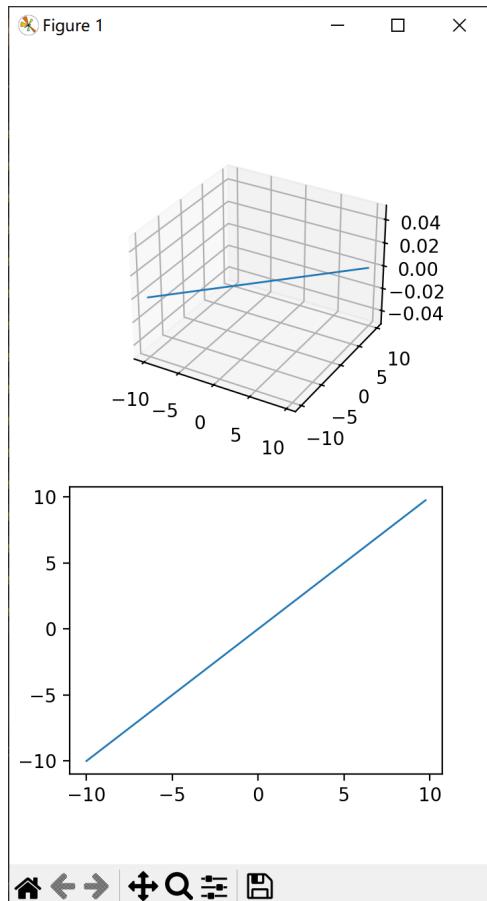
下面这个例子可以很好地展现长度相同的 x,y 直接运算得到 z 会发生什么：

```

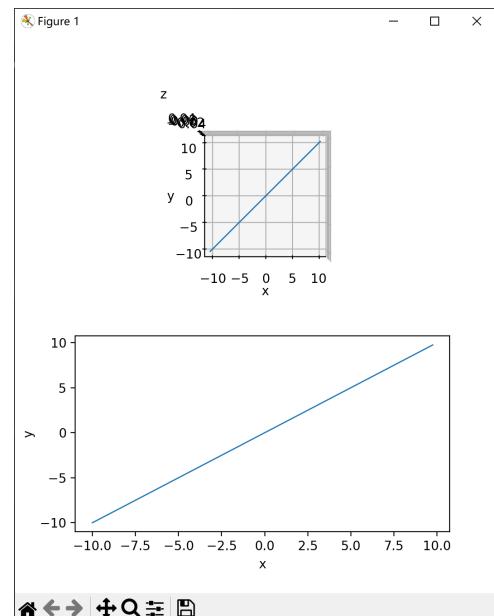
1 import matplotlib.pyplot as plt
2 import numpy as np
3 ax1=plt.subplot(*args: 211,projection='3d')
4 ax2=plt.subplot(212)
5 x = np.arange(-10, 10, 0.25)
6 y = np.arange(-10, 10, 0.25)
7 z1=x**2-y**2
8 ax1.plot(*args: x,y,z1,lw=1)
9 ax1.set_xlabel('x')
10 ax1.set_ylabel('y')
11 ax1.set_zlabel('z')
12 ax1.view_init(elev=90, azim=-90)
13 ax2.plot(*args: x,y,lw=1)
14 ax2.set_xlabel('x')
15 ax2.set_ylabel('y')
16
17 plt.show()

```

图 55: 6.2.2.3



(a) 6.2.2.4 未正对状态



(b) 6.2.2.5 z 轴正对状态

但是如果我们在 $z = x^2 - y^2$ 前加上 $x, y = np.meshgrid(x, y)$, 图像就会变成如下这样:

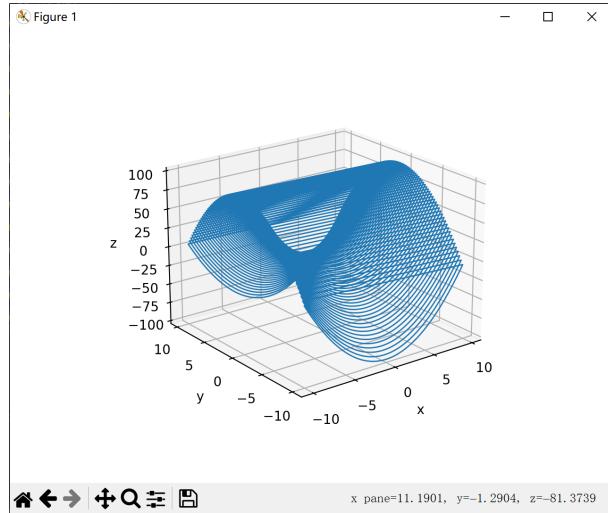


图 57: 6.2.2.5

可以看到已经很接近我们心目中 3D 图的样子了对吧, `meshgrid` 的作用就是生成二维网格, 或者更确切的说, $x, y = np.meshgrid(x, y)$ 是将 x, y 这两个长度可以不同的二维数组变成两个可以运算的二维数组 (两个可以运算的矩阵) 具体做法是在另一个方向上复制自己至对方的长度 (???) 没关系, 我们来看一个例子:

```
try 8.1.py
Try 8.2.py
meshgrid Try x

C:\Users\金御风\Py
[[0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [0 1 2 3 4 5]
 [[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]
 [5 5 5 5 5]]
```

0	(0,0)	1	(1,0)	2	(2,0)	3	(3,0)	4	(4,0)	5	(5,0)
1	(0,1)		(1,1)		(2,1)		(3,1)		(4,1)		(5,1)
2	(0,2)		(1,2)		(2,2)		(3,2)		(4,2)		(5,2)
3	(0,3)		(1,3)		(2,3)		(3,3)		(4,3)		(5,3)
4	(0,4)		(1,4)		(2,4)		(3,4)		(4,4)		(5,4)
5	(0,5)		(1,5)		(2,5)		(3,5)		(4,5)		(5,5)

图 58: 6.2.2.6

它的代码放在后面, 在左边我分别令 x, y 是 0~5 六个整数组成的列表, 然后对它们

用 meshgrid，最后分别打印出 x, y，可以看到它们分别在对方的方向上把自己 (0, 1, 2, 3, 4, 5) 复制了 6 次，如此一来二者就可以形成一张二维网格，可以进行运算。对照右边的网格图，我专门调整了 x,y 轴位置和刻度位置，可以很清楚的看出 x 打印出来就是所有横坐标的矩阵，y 打印出来就是所有纵坐标的矩阵。至于为什么 meshgrid 被设计的将 x,y 在不同的方向复制，这是否与矩阵的运算有关，请读者自行思考。

总之，如果我们希望两个自变量组成的平面能够作为自变量，即形成网格，就需要在运算前对它们使用 meshgrid，将它们转换为可以运算的二维数组，这样因变量的也会成为一个二维数组，它们就可以用来生成图像了。

一个小提醒，在两个自变量取点数一样多的情况下，线图（plot）和点图（scatter）都是不会报错的，但是曲面图，网格图，等高线图仍然会报错，它们的绘制必须提前使用 meshgrid 方法。

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 x=range(0,6)
4 y=range(0,6)
5 x, y = np.meshgrid(*xi: x, y)
6 print(x)
7 print(y)
8 ax=plt.subplot()
9 ax.set_aspect('equal')
10 ax.xaxis.set_ticks_position('top')
11 plt.xticks(np.linspace(start: 0, stop: 5, num: 6))
12 plt.yticks(list(np.linspace(start: 0, stop: 5, num: 6)))
13 ax.invert_yaxis()
14 for spine in ax.spines.values():
15     spine.set_color('None')
16 for i in range(0,6):
17     for j in range(0,6):
18         if i!=j:
19             plt.text(i,j,'({},{})'.format(*args: i,j),color='b')
20             plt.text(j,i,'({},{})'.format(*args: j,i),color='b')
21         else:
22             plt.text(i,j,'({},{})'.format(*args: i,j),color='b')
23 ax.grid(True)
24 plt.show()
```

图 59: 6.2.2.7

6.3 各类 3D 图的方法

6.3.1 线图与点图

这一节我们开始详细介绍各种 3D 图绘制的方法。首先是线图和点图，3D 图的线图，点图与平面图其实差异不大，平面图中的那些设置在 3D 图中基本一样适用。最本

质的区别还是多了一个维度，要向 plot 或 scatter 传递的参数多了一个。至于 x,y,z 是否需要在建立关系的时候使用 meshgrid 构建网格，完全依照实际需求。很有意思的是，如果我们让 $x,y = np.meshgrid(x,y)$, x,y 变成了两个二维数组，而 z 还是一维数组，这时把它们丢进 scatter 或 plot 并不会出现报错，这时因为 numpy 数组自带一套广播机制，维度小的数组会通过在一个方向上复制自身来匹配其它数组的维度，具体请参考[这里](#)[8.2.1](#)。

6.3.2 曲面图

曲面图基本语法如下：

```
ax.plot_surface(x, y, z, **kwargs)
```

其中 x,y,z, 都是二维数组，通常是 x,y 由 meshgrid 函数生成，z 由 x,y 计算得到。`**kwargs` 指其它设置的参数，其中包括我们早就熟悉的透明度 alpha，颜色映射 cmap，还有线宽 linewidth/lw，透明度和颜色映射都没什么好说的，忘了颜色映射的话 [click here: 8.7](#)。线宽在这里出现可能会让你觉得很奇怪，这里线宽其实是曲面表面网格线的宽度，如果令其为 0，则不显示网格线。（但是在我这个版本的 matplotlib 中 3D 曲面图网格线被整得很奇怪，它的出现与否并不受 lw 是否为 0 控制，而是受另一个参数抗锯齿 antialiased 是否 =True 控制，antialiased=True 则由网格线，反之没有，而默认情况下就是 antialiased=True，所以如果这两个参数你都不动，是会有网格线的；同时我们也知道了如果想要去掉网格线，只需要令 antialiased=False 就行了。）

曲面上的网格线长这样：

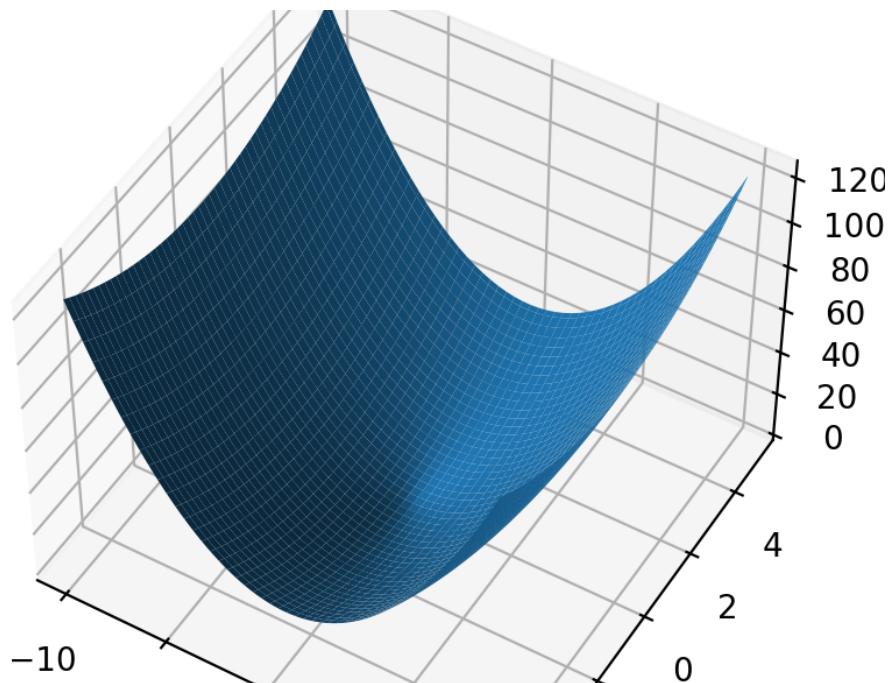


图 60: 6.3.2.1

其它一些参数包括抗锯齿 antialiased，行和列的采样点数 rcount 和 ccount 和阴影 shade。

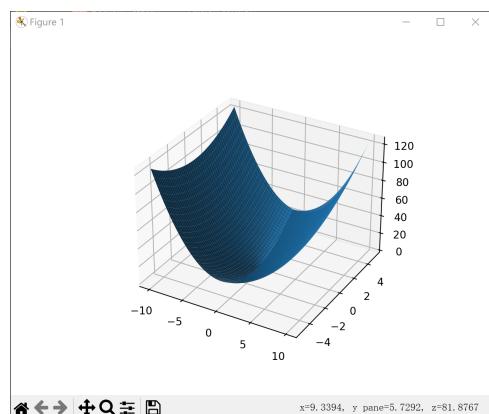
1. antialiased：我真的不想讲这个参数，在我目前看来它除了刚刚说的控制网格线的生成，没有任何用处，理论上它 =True 时能够使图形边缘更加平滑，反之图形边缘可能显得锯齿状，但可以提高渲染速度，适合需要快速绘制的场景，但我个人觉得效果很不明显。它默认下是 True。

2. rcount 和 ccount：rcount 用于指定沿行方向的采样点数。ccount 则用于指定沿列方向的采样点数，你可能会说，采点数难道不是我们在用 np.linspace 方法调用 x,y 时就确定了的吗，这其实是一个挺复杂的问题。**我们在一开始用 np.linspace 方法去的点是原始数据，而最后真正的绘制（专业一点叫渲染）是从原始数据中抽取一部分来进行绘制**。比如说，我对 x,y 的定义是 np.linspace(-5, 5, 100)，那么它们在 meshgrid 后会形成一个 100*100 的网格，共 10000 个数据点，但是如果令 rcount=50, ccount=50，意味着最后是从 100 行中抽取 50 行，100 列中抽取 50 列参与渲染，也就是只有 2500 个点。**总之，rcount 和 ccount 的取值一定是小于 linspace 中的设定的，它们的作用是限制原始数据量，减少渲染时的计算量，避免因为数据量过大而导致绘图缓慢或内存不足，但同时图像精度也会降低**。因此，为了在高精度和渲染速度之间找到平衡，建议的做法是增大 linspace 的取点数，即抬高精度上限，然后调整 rcount 和 ccount 值找到最优位置。（旧版本的 matplotlib 有两个参数 rstride 和 cstride，控制网格线的密度，但在新版本中已被基本废弃，被 rcount 和 ccount 取代。）

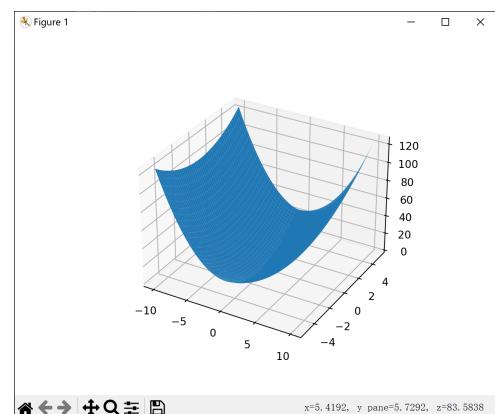
默认情况下 rcount=50, ccount=50.

3. shade：说来它的功能有点好笑，它用于控制是否为表面添加阴影效果，取值为布尔值。因为阴影效果可以使表面的立体感更强（? maybe）。shade=True 时有阴影，反之没有。默认情况是 True。

有意思的是，曲面为单一颜色时阴影有无很明显，但是采用颜色映射 cmap 时则几乎没有区别。



(a) 6.3.2.2 shade=True



(b) 6.3.2.3 shade=False

6.3.3 线框图

线框图的函数是 `plot_wireframe`, 方法如下, 其实跟曲面图一模一样, 就是设置上略有不同:

```
ax.plot_wireframe(x, y, z, **kwargs)
```

线框图函数的设置也有经典的颜色、透明度、线宽, 不过这里的线宽就是线框的线宽, 既然其主体是线条, 自然少不了线框的样式 `linestyle`, 跟之前说过的一模一样, `'-'` (实线)、`'--'` (虚线)、`'.'` (点线) 等。还有线框图也有 `rcount` 和 `ccount`, 跟曲面图是一样的。

因为与曲面图相似度很高, 到这里我们已经将线框图介绍完了, 下面展示一个简单的线框图代码示例:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 # 定义函数
4 def f(x, y):
5     return np.sin(np.sqrt(x**2 + y**2))
6 # 创建网格数据
7 x = np.linspace(-6, 6, num=30)
8 y = np.linspace(-6, 6, num=30)
9 X, Y = np.meshgrid(x, y)
10 Z = f(X, Y)
11 # 绘制线框图
12 ax = plt.subplot(*args: 111, projection='3d')
13 ax.plot_wireframe(X, Y, Z, color='blue', linewidth=1, linestyle='-', alpha=0.8)
14 # 设置坐标轴标签
15 ax.set_xlabel('X')
16 ax.set_ylabel('Y')
17 ax.set_zlabel('Z')
18 # 设置标题
19 ax.set_title('3D Wireframe Plot')
20 plt.show()
```

图 62: 6.3.3.1

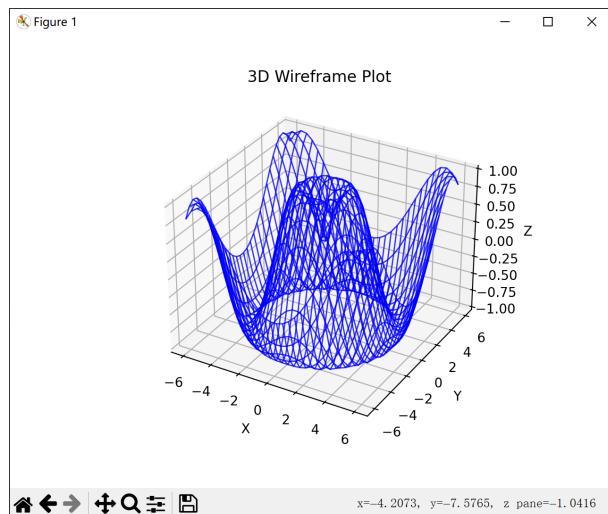


图 63: 6.3.3.2

6.3.4 等高线图

等高线图其实是一种平面图，我们一般并不会像第一部分所举的例子那样吧等高线图画仅空间坐标系的一个平面里，之所以把它放在这里说，是因为首先等高线图反映的其实是 3D 图，它相当于是将一幅 3D 图拍扁了，用二向箔降维打击了（？），其次它的方法与 3D 图非常相似，它的绘制也需要传递 x, y, z 三个参数，所以我们可以将其视为一种特殊的 3D 图。

我们首先看一个例子，上图：

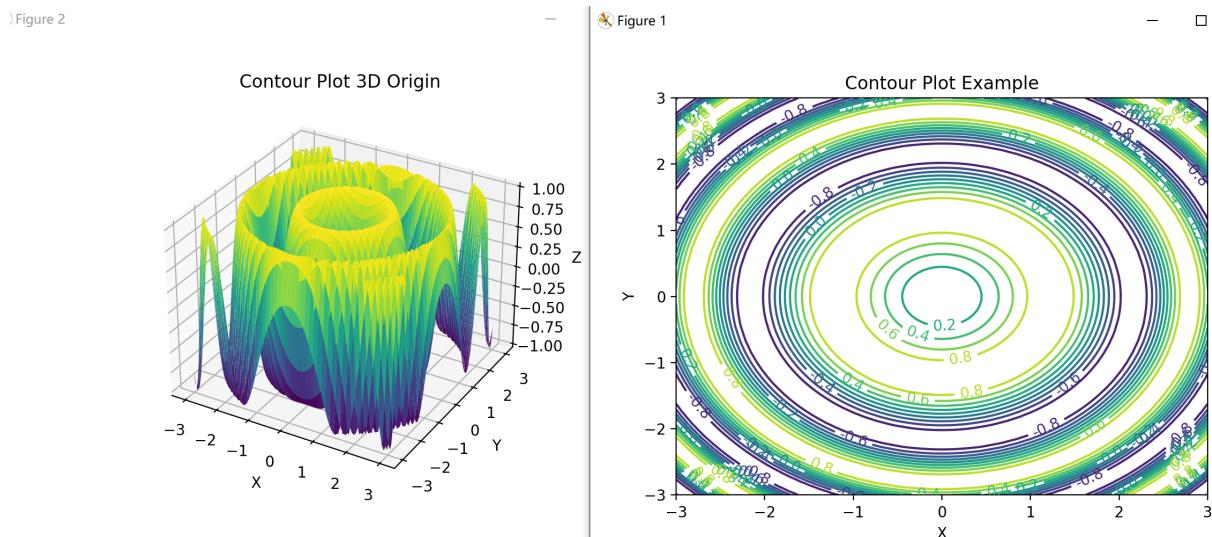


图 64: 6.3.4.1

```

import numpy as np
import matplotlib.pyplot as plt
cmap='viridis'
fig1=plt.figure()
fig2=plt.figure()
ax1=fig1.add_subplot()
ax2=fig2.add_subplot(projection='3d')
x = np.linspace(-3, stop=3, num=100)
y = np.linspace(-3, stop=3, num=100)
X, Y = np.meshgrid(x, y)
Z = np.sin(X**2 + Y**2)
# 绘制等高线图并添加等高线标签
contours = ax1.contour(*args: X, Y, Z, levels=10, cmap=cmap, linewidths=1.5, linestyles='solid')
ax1.clabel(contours, inline=True, fmt='%1.1f', fontsize=10)
# 设置标题和坐标轴标签
ax1.set_title('Contour Plot Example')
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax2.set_title('Contour Plot 3D Origin')
ax2.set_xlabel('X')
ax2.set_ylabel('Y')
ax2.set_zlabel('Z')
ax2.plot_surface(X, Y, Z, cmap=cmap)
plt.show()

```

图 65: 6.3.4.2

为了展示等高线图与它的祖宗 3D 曲面图的紧密关联，我在上面这个例子中将其相应的 3D 图一并绘制出来以作对照。整段代码中真正有关等高线图方法的只有下面这两行：

```
contours = ax1.contour(X, Y, Z, levels = 10, cmap = cmap, linewidths = 1.5, linestyles = 'solid' )  
ax1.clabel(contours, inline = True, fmt = '%1.1f', fontsize = 10 )
```

X,Y,Z: 在等高线图中，原本的 z 轴值成为高度数据，以标签的形式显示出来，其传递方式还是和 3D 图的那些方法一样，x,y,z 占据前三个参数。

levels: 指定等高线的数量或具体值。如果是整数 n，则会自动生成 n+1 个等高线级别，这些级别是等距的；如果是数组，则直接使用数组中的值作为等高线级别。

linestyles 和 linewidths: 可以接受单个浮点数或字符串，也可以接受一个数组或列表，从而为每一条等高线设置不同的线形和线宽。

fontsize: 就是高度标签的字体大小。

inline: 布尔值，True 的时候标签被嵌入等高线中，就是等高线会在标签处中断，反之等高线直接穿过标签（我想没人会喜欢这样吧）。默认是 True，所以我觉得这个参数基本是不用动的。

fmt: 格式化字符串，用于指定标签的格式，比如保留几位小数。详见[8.6.4](#)

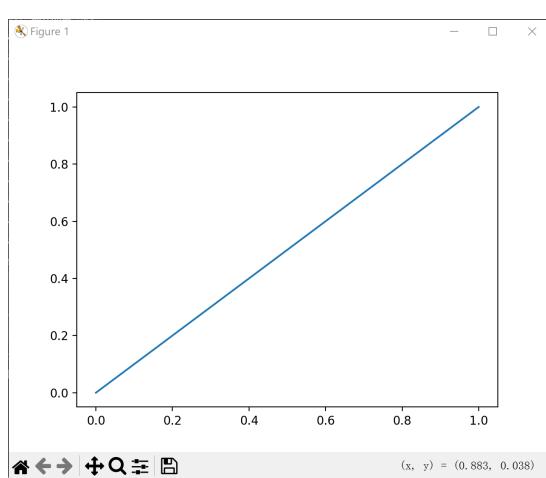
还有一些更复杂的设置，包括抗锯齿 antialiased，locator，origin，extent 等但是很多时候效果并不显著，而且比较繁复这里就不再介绍。

7 事件处理与交互

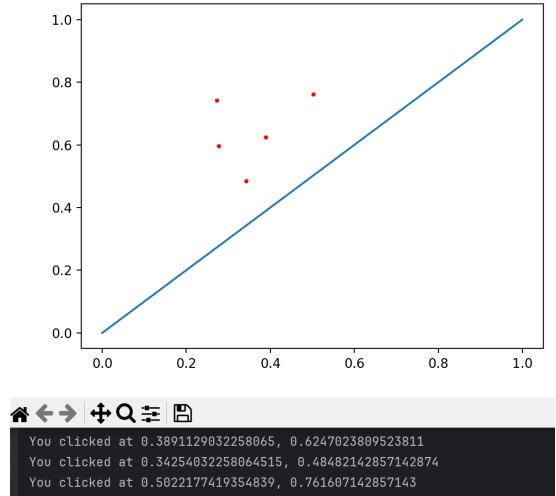
在之前的章节中，我们都是写好一段代码，然后一键运行，它会生成一张相应的图像，但如果我希望在图像生成出来后直接对其进行一些操作呢？这就是我们本章要讨论的内容——事件的交互与处理。友情提示，本章内容难度可能会有一点点大哦。

7.1 什么是事件处理与交互——事件机制概述

为了帮助读者尽快熟悉这个陌生的概念，我们先来看一个简单的事件交互的例子：



(a) 7.1.1.1



(b) 7.1.1.2

注意看，图 7.1.1.1 是一张平平无奇的直线线图，解析式是 $x-y=0$ ，定义域在 $[0,1]$ ，但是如果我们用鼠标在上面点几下，会发现点过的地方出现了红点，同时该点坐标被打印了出来，如图 7.1.1.2 所示。

在这个例子中，我通过鼠标的点击使图像发生了变化，或者更确切的说，让整个程序的运行都发生了改变，因为被点击点的坐标也被打印出来了。外界的这些操作比如鼠标点击，被称作“事件”，这些事件对程序造成影响的过程就叫做事件的交互。

接下来我们来看看这个事件交互是如何实现的，以上示例的程序如图 7.1.1.3：

```
1 import matplotlib.pyplot as plt
2
3 usage
4 def onclick(event):
5     print(f'You clicked at {event.xdata}, {event.ydata}')
6     plt.scatter(event.xdata, event.ydata, c='r', s=5)
7 fig, ax = plt.subplots()
8 ax.plot([0, 1], [0, 1])
9 cid = fig.canvas.mpl_connect('button_press_event', onclick)
10 plt.show()
```

图 67: 7.1.1.3

我们来分析一下这段程序：首先我们定义了一个函数 `onclick`，它接受 `event` 事件对象，这个 `event` 包含所有可能的事件，然后我们通过 `event.xdata`, `event.ydata` 获取了事件发生位置的坐标，也就是鼠标点击的坐标，然后将它们打印出来以及绘制成红点，`fig.canvas.draw()` 表示更新画布，从而让绘制的点立刻显示出来。以上就是函数 `onclick` 的全部功能。

7、8 两行我们很熟悉了，就是创建了图形对象与子图，并在子图上绘制了一条直线。然后就是一行奇怪的命令：

```
cid = fig.canvas.mpl_connect('button_press_event', onclick)
```

这个 `mpl_connect` 是一个监听函数，它能将具体事件与函数连接在一起，比如在上例中，当它监听到 `button_press_event` 这个事件时，它就会调用 `onclick` 函数，从而实现了由事件触发函数。

通过上面这个例子，我们已经初步了解了 `matplotlib` 事件交互与处理的基本逻辑，总结一下，就是首先定义某个事件发生时所调用的函数，这个函数以事件对象 `event` 为参数，这样我们就能在函数里获取 `event` 的属性的数据了，像事件的坐标 `xdata`, `ydata` 就是一种属性。通过事件的属性数据我们可以定义对图像进行哪些操作，这就是一个事件触发函数。最后，我们通过 `mpl_connect` 将具体事件与相应的触发函数连接起来，这样在特定事件被监听到时，对应的函数就会被执行。

`matplotlib` 支持多种事件对象 `event`，如下表：

事件类型	触发情况
<code>button_press_event</code>	鼠标按下
<code>button_release_event</code>	鼠标释放
<code>motion_notify_event</code>	鼠标移动
<code>scroll_event</code>	滚轮滚动
<code>key_press_event</code>	按键按下
<code>key_release_event</code>	按键释放
<code>pick_event</code>	拾取对象
<code>resize_event</code>	窗口大小改变
<code>close_event</code>	关闭窗口

表 1: Matplotlib 事件类型一览表

事件对象的属性就比较复杂了具体可参考表 2，注意区分 `event.x` 和 `event.xdata`，前者是像素坐标，后者才是我们广义的坐标系的坐标。

属性	说明
name	事件名称 (如 'button_press_event')
canvas	触发事件的 FigureCanvas 对象
guiEvent	底层 GUI 事件 (可能为 None)
inaxes	事件发生时鼠标所在的 Axes (否则 None)
x, y	事件发生时的像素坐标 (窗口坐标系)
鼠标事件 (MouseEvent)	
xdata, ydata	数据坐标系中的 x/y (鼠标在 Axes 内时)
button	按钮编号 (1= 左键, 2= 中键, 3= 右键, 'up'/'down' 滚轮)
step	滚轮滚动步长 (仅 scroll_event)
dblclick	是否为双击事件 (Matplotlib 3.4+)
键盘事件 (KeyEvent)	
key	按下的键 (如'a', 'ctrl', 'shift', 'left')
拾取事件 (PickEvent)	
mouseevent	关联的鼠标事件 (MouseEvent)
artist	被拾取的 Artist (如 Line2D、Patch)
ind	若 artist 是 Line2D 或 Collection, 返回被拾取点的索引列表

表 2: Matplotlib 事件对象 (event) 的属性

7.2 一些需要阐明的和解释的

7.2.1 fig 与 canvas

这两个概念比较容易混淆。之前说过，我们用 `fig=plt.figure()` 创建出来的是一个图形对象，它是 Matplotlib 中所有绘图元素的顶级容器，可以包含一个或多个子图，之前我们都是直接对它进行操作，从而创建和管理图形内容。

`canvas` 是这些操作的底层实现，将 `fig` 管理的内容渲染到屏幕上或保存为文件，通常我们不需要直接操作它，因为它一直是自动处理的。而现在，由于它是图形与用户交互的界面，我们需要对这个画布进行直接操作。于是，这个本应是最显眼的概念（毕竟所有图像都是在 `canvas` 上渲染出来的嘛）到现在才正式进入我们的视野。

在级别上，`figure` 是最高级容器，`figure` 对象包含一个 `canvas` 属性，`canvas` 是 `figure` 的绘图区域，负责实际的渲染工作。

7.2.2 artist 艺术家？

在 Matplotlib 中，`Artist` 是一个非常通用的概念，几乎可以描述所有可以被绘制的图形对象。而在事件交互中，`event.artist` 专指那些可以触发事件的对象。

`Artist` 是 Matplotlib 中的一个基类，几乎所有可以被绘制的图形对象都是 `Artist` 的

子类。这些对象包括：

1. 线条 (Line2D)：用于绘制直线或曲线。（通常是 plot 画出来的）
2. 散点 (PathCollection)：用于绘制散点图。（通常是 scatter 画出来的）
3. 矩形 (Rectangle)：用于绘制矩形。（plt.Rectangle 函数画的，一个专门画矩形的函数）
4. 文本 (Text)：用于添加文本注释。（不用说了，text 函数）
5. 图像 (AxesImage)：用于显示图像。（通常是 imshow 画出来的）
6. 坐标轴 (Axes)：整个坐标轴区域。
7. 图形 (Figure)：整个图形窗口。

不难发现，最后两种 Artist 与前四种不一样，它们不是某一种几何图形，而是一种装图形的容器。于是，我们将 Artist 分为两类：原始艺术家 (Primitive Artists) 与容器艺术家 (Container Artists)，前四种称为原始艺术家，后两者是容器艺术家。容器艺术家是包含其他艺术家的对象，它们用于管理其内部的多个原始艺术家。

原始艺术家的访问很简单，它们通常由相应的绘图函数直接返回，比如一个 Line2D 的访问就是如 line, = ax.plot(x, y, label="Line")。容器艺术家中子艺术家的访问则需要用到 get_children() 函数，比如 children = ax.get_children() 可以获得 Axes 中所有子艺术家。

get_children() 函数通常与区分艺术家类型的 isinstance() 函数联合使用，这个函数的具体介绍放到详述[8.10](#)

在事件交互中的 event.artist 在事件交互中，event.artist 是触发事件的具体对象。只有那些可以触发事件的对象才会被赋值给 event.artist。例如：如果你点击了一个散点图中的某个点，event.artist 是被点击的 PathCollection 对象。如果你点击了一条线，event.artist 是被点击的 Line2D 对象。如果你点击了一个矩形，event.artist 是被点击的 Rectangle 对象。[8.10](#)中的那个运用 isinstance() 函数的例子也是 artist 在事件交互中应用的很好的例子。

7.3 事件交互的实战应用

现在你已经掌握了事件交互与处理的基本概念，让我们在实际应用中加深对它的理解吧。

首先 close_event 太简单了，一点意思也没有，这里就不讲它了。

7.3.1 窗口大小改变

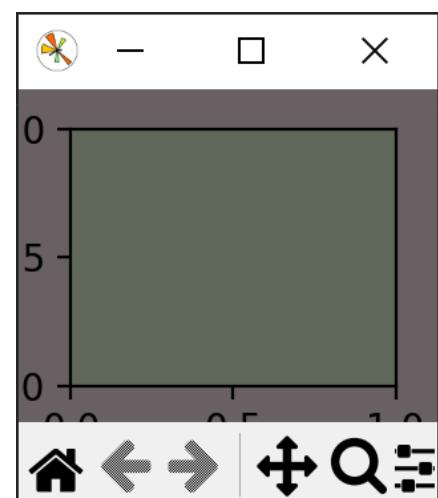
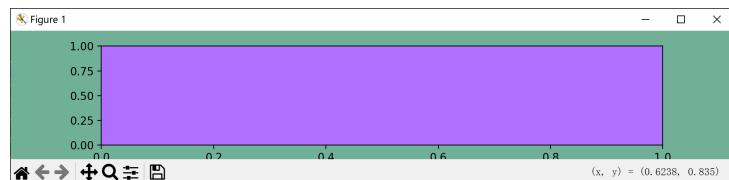
下面我们来看看比较少见但也能整点活的 `resize_event`。这是在改变窗口大小时触发的事件，它的核心属性就是窗口的长宽。首先你必须要记住的是（虽然我其实经常记不住）默认状态下窗口长宽分别是 640 与 480，最大值（全屏下）是 1280 与 625。知道了这些我们就可以开整了 [手动 Doge]，想想看如何让窗口与画布的颜色随着窗口的大小变化做相应的变化？（小提示，它们颜色的设置用 `set_facecolor`，但是这个函数不支持颜色映射哦）

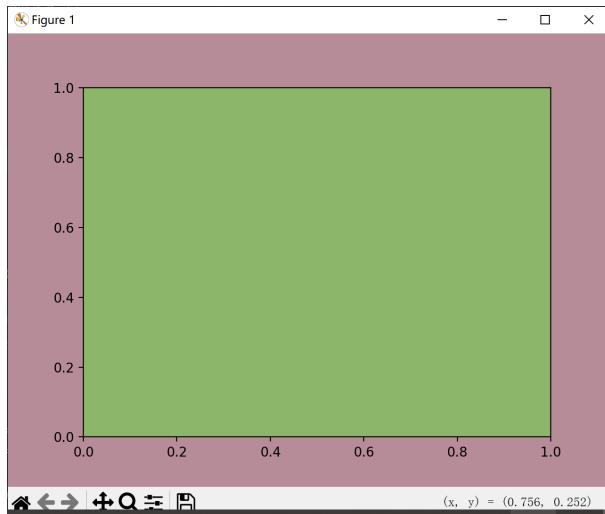
本人写出的代码如下，仅供参考：

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.colors as mcolors
fig,ax=plt.subplots()
1 usage
def on_resize(event):
    a=event.width
    b=event.height
    norm=mcolors.Normalize(vmin=0, vmax=3, clip=True)
    y1=tuple(norm([np.exp((b)/625),np.exp((a)/1280),np.exp((a+b)/1905)]))
    y2=tuple(norm([np.exp((a)/1280),np.exp((b)/625),np.exp((a-b)/700)]))
    fig.patch.set_facecolor(color=y1)
    ax.patch.set_facecolor(color=y2)
    plt.draw()
    print(f'a={a},b={b}')
fig.canvas.mpl_connect('resize_event', on_resize)
plt.show()
```

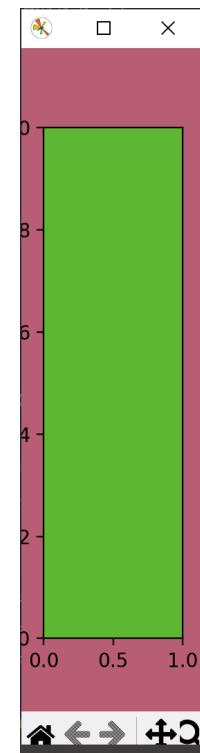
图 68: 7.3.1.1

我们来对它进行一番拉扯，看看效果如何：





(a) 7.3.1.4



(b) 7.3.1.5

由于不能使用 `cmap`, 我们只能通过 RGB 的方式设定变化的颜色, 通过指数函数的使用, 我们将长宽变化的幅度放大, 同时又控制在 01 之间。

7.3.2 拖拽事件

别急着划回 55 页的表格，那里面没有这个事件，它其实是鼠标点击事件与鼠标移动事件的结合产物。在以下这个例子中，我们会用之前提到的 plt.Rectangle 函数创建一个矩形，并让它变得可以拖动。

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 usage
5 class DraggableRectangle:
6     def __init__(self, rect):
7         self.rect = rect
8         self.press = None
9         self.cidpress = self.rect.figure.canvas.mpl_connect('button_press_event', self.on_press)
10        self.cidrelease = self.rect.figure.canvas.mpl_connect('button_release_event', self.on_release)
11        self.cidmotion = self.rect.figure.canvas.mpl_connect('motion_notify_event', self.on_motion)
12
13 usage
14 def on_press(self, event):
15     if event.inaxes != self.rect.axes:
16         return
17     contains, attrd = self.rect.contains(event)
18     if not contains:
19         return
20     self.press = self.rect.xy, (event.xdata, event.ydata)
```

图 71: 7.3.2.1

```

19      1 usage
20      def on_motion(self, event):
21          if self.press is None or event.inaxes != self.rect.axes:
22              return
23          (x0, y0), (xpress, ypress) = self.press
24          dx = event.xdata - xpress
25          dy = event.ydata - ypress
26          self.rect.set_x(x0 + dx)
27          self.rect.set_y(y0 + dy)
28          self.rect.figure.canvas.draw()
29
30      1 usage
31      def on_release(self, event):
32          self.press = None
33          self.rect.figure.canvas.draw()
34
35      fig, ax = plt.subplots()
36      rect = plt.Rectangle( xy: (0.1, 0.1), width: 0.2, height: 0.2, facecolor='red')
37      ax.add_patch(rect)
38      dr = DraggableRectangle(rect)
39      plt.show()

```

图 72: 7.3.2.2

其效果如下，这个红色的矩形是可以拖动的。

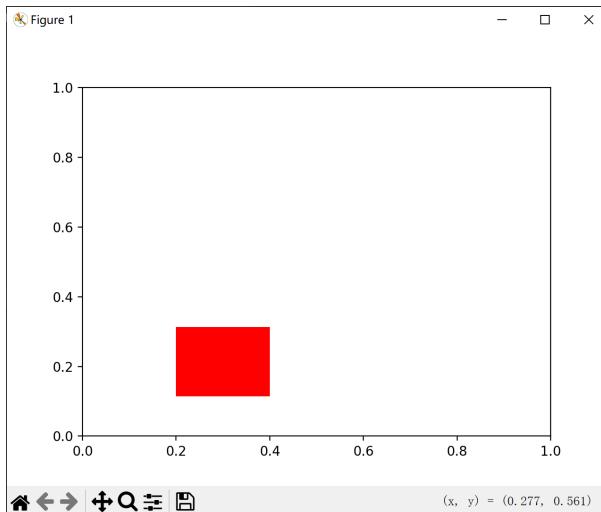


图 73: 7.3.2.3

以上代码还有些地方要简单解释一下。rect.xy 中，xy 是矩形的一个属性，指矩形左下角的坐标。

contains 是判断某要素是否在某区域内的函数，代码中用于判断鼠标点击是否发生在矩形内；你也许会好奇，这个 attrd 是啥，它在后面再也没有被用到，为什么把它之间删掉却会导致矩形不能拖动了。这就要说到 Python 的神奇语法规则，当几个变量直接等于一个拥有相同数量元素的元组时，Python 会自动解包元组，将元素值一一赋给变量，比如这里就是，self.rect.contains(event) 是一个两元素的元组。但是如果删去 attrd，

就变成了 contains 等于元组，因为一般的逗号占位在这种自动解包元组的情况下是不适用的，如果实在要占位需要用下划线：contains, _ = self.rect.contains(event)。set_x, set_y 函数没讲，但是它们的功能一看即知，这里就不多介绍了。

请读者自行思考，如何让矩形碰到边界就不能再被拖动了呢。

7.3.3 鼠标、键盘、滚轮大杂烩

结合之前所学的各种鼠标、键盘、滚轮事件，我们来制作一个绘画板，要求如下：

1. 具有 typical, linear, circle 三种模式；能绘制曲线、直线、圆三种几何图形。这三种模式可以通过键盘按动进行切换
2. 在 typical 模式下，鼠标按住拖动时直接留下痕迹；在 linear 模式下，以鼠标按下位置为起点，松开位置为终点画出直线，且要求按住拖动时直线已经出现并另一头会跟着鼠标跑，同时直线旁标识出直线的长度；在 circle 模式下，拖动时所绘半径（是由圆心到圆周的画法）与圆周均已出现并随着鼠标动，半径的要求与直线一致。
3. 已经绘制的在手动删除前必须保留在画布上，在按下 delete 键后清空整个画布。
4. 画布在未手动调整显示范围时保持不动，不得自动改变范围。手动改变可以通过上下左右键挪动画布，也可以通过滚轮进行整体的缩放。
5. 当前模式能够在画布上方显示出来。

本人所写代码如下：

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots()
5 ax.set_xlim(-10, 10)
6 ax.set_ylim(-10, 10)
7 ax.set_aspect('equal')
8
9 class DrawState:
10     def __init__(self):
11         self.pressed = 0
12         self.line_number = 0
13         self.circle_number = 0
14         self.x_list = []
15         self.y_list = []
16         self.lines = []
17         self.radius_values = []
18         self.exist_lines = []
19         self.circles = []
```

```

20     self.exist_circles = []
21     self.figure_pattern = 'typical'
22     self.x1, self.y1, self.x2, self.y2, self.r_value = None, None, None
23     , None, None
24
25 state = DrawState()
26 ax.set_title(f"Current Mode: {state.figure_pattern}", color='blue')
27
28 def on_click(event):    ##### 鼠标点击事件
29     ax.scatter(event.xdata, event.ydata, color='r', s=10)
30     plt.draw()
31     state.pressed = 1 - state.pressed
32
33 def on_released(event):    ##### 鼠标松开事件
34     ax.scatter(event.xdata, event.ydata, color='r', s=10)
35     plt.draw()
36     state.pressed = 1 - state.pressed
37     state.x_list = []
38     state.y_list = []
39     if state.figure_pattern == 'linear':
40         state.line_number += 1
41         state.exist_lines.append(state.lines[-1])
42         if state.line_number >= 1:
43             for line in state.exist_lines[-state.line_number:]:
44                 ax.add_line(line)
45             fig.canvas.draw()
46     if state.figure_pattern == 'circle':
47         state.circle_number += 1
48         state.exist_circles.append(state.circles[-1])
49         if state.circle_number > 1:
50             for circle in state.exist_circles[-state.circle_number:]:
51                 ax.add_line(circle)
52             fig.canvas.draw()
53
54 def typical_move(event):      ### 一般曲线绘制函数
55     if state.pressed == 1:
56         state.x_list.append(event.xdata)
57         state.y_list.append(event.ydata)
58         ax.plot(state.x_list, state.y_list, c='r')
59         plt.draw()
60
61 def linear_move(event):      ### 直线绘制函数
62     if event.name == 'motion_notify_event' and state.pressed == 1 and event
       .inaxes is not None:
           state.x_list.append(event.xdata)

```

```

63     state.y_list.append(event.ydata)
64     state.x1 = state.x_list[0]
65     state.y1 = state.y_list[0]
66     state.x2 = state.x_list[-1]
67     state.y2 = state.y_list[-1]
68     if None in (state.x1, state.x2, state.y1, state.y2):
69         raise ValueError("One or more coordinates are None. Please
70 check the input data.")
71     state.r_value = np.sqrt((state.x2 - state.x1) ** 2 + (state.y2 -
72 state.y1) ** 2)
73     line, = ax.plot([state.x1, state.x2], [state.y1, state.y2], c='r')
74     state.lines.append(line)
75     r = ax.text((state.x1 + state.x2) / 2 + 0.5, (state.y1 + state.y2)
76 / 2 + 0.5, 'r={}'.format(round(state.r_value, 2)),
77 fontdict={'fontsize': 10, 'fontweight': 'light', 'color
78 ': 'blue'})
79     state.radius_values.append(r)
80     if len(state.lines) > 1:
81         for line in state.lines[:-1]:
82             line.remove()
83     state.lines = state.lines[-1:]
84     plt.draw()
85
86
87     if len(state.radius_values) > 1:
88         for r in state.radius_values[:-1]:
89             r.remove()
90         state.radius_values = state.radius_values[-1:]
91         plt.draw()
92         plt.draw()
93
94     return state.x1, state.x2, state.y1, state.y2, state.r_value
95
96
97 def circle_move(event):    ### 圆形绘制函数
98     x1, x2, y1, y2, r_value = linear_move(event)
99     theta = np.linspace(0, 2 * np.pi, 100)
100    circle, = ax.plot(x1 + r_value * np.cos(theta), y1 + r_value * np.sin(
101 theta), c='r')
102    state.circles.append(circle)
103    if len(state.circles) > 1:
104        for circle in state.circles[:-1]:
105            circle.remove()
106        state.circles = state.circles[-1:]
107        plt.draw()
108        plt.draw()

```

```

103 def on_motion(event):      ##### 整合各图形绘制的函数，鼠标移动控制
104     if state.pressed == 1:
105         if state.figure_pattern == 'typical':
106             typical_move(event)
107             plt.draw()
108         if state.figure_pattern == 'linear':
109             linear_move(event)
110             plt.draw()
111         if state.figure_pattern == 'circle':
112             circle_move(event)
113             plt.draw()
114
115 def pattern_change(event):  ### 键盘事件控制函数
116     if event.name == 'key_press_event' and event.key == 'l':    ### 直线
117         state.figure_pattern = 'linear'
118     elif event.name == 'key_press_event' and event.key == 'r':   ### 圆
119         state.figure_pattern = 'circle'
120     elif event.name == 'key_press_event' and event.key == 't':   ### 一般曲线
121         state.figure_pattern = 'typical'
122     if event.key == 'delete':    #### 清空 但是模式不变
123         ax.clear()
124         ax.set_xlim(-10, 10)
125         ax.set_ylim(-10, 10)
126         ax.set_aspect('equal')
127         state.pressed = 0
128         state.line_number = 0
129         state.circle_number = 0
130         state.x_list = []
131         state.y_list = []
132         state.lines = []
133         state.radius_values = []
134         state.exist_lines = []
135         state.circles = []
136         state.exist_circles = []
137     if event.key == 'up':
138         ax.set_ylim(ax.get_ylim()[0] + 0.1, ax.get_ylim()[1] + 0.1)
139     if event.key == 'down':
140         ax.set_ylim(ax.get_ylim()[0] - 0.1, ax.get_ylim()[1] - 0.1)
141     if event.key == 'left':
142         ax.set_xlim(ax.get_xlim()[0] - 0.1, ax.get_xlim()[1] - 0.1)
143     if event.key == 'right':
144         ax.set_xlim(ax.get_xlim()[0] + 0.1, ax.get_xlim()[1] + 0.1)
145     fig.canvas.draw()
146     ax.set_title(f"Current Mode: {state.figure_pattern}", color='blue')

```

```

147
148 def on_scroll(event):
149     ax.set_xlim(ax.get_xlim()[0] * (1 - event.step * 0.01), ax.get_xlim()[1] * (1 - event.step * 0.01))
150     ax.set_ylim(ax.get_ylim()[0] * (1 - event.step * 0.01), ax.get_ylim()[1] * (1 - event.step * 0.01))
151     fig.canvas.draw()
152
153 fig.canvas.mpl_connect('button_press_event', on_click)
154 fig.canvas.mpl_connect('button_release_event', on_released)
155 fig.canvas.mpl_connect('motion_notify_event', on_motion)
156 fig.canvas.mpl_connect('key_press_event', pattern_change)
157 fig.canvas.mpl_connect('scroll_event', on_scroll)
158
159 plt.show()

```

PDF 没法显示动图，所以这里只好放一张画完后的画布效果图。

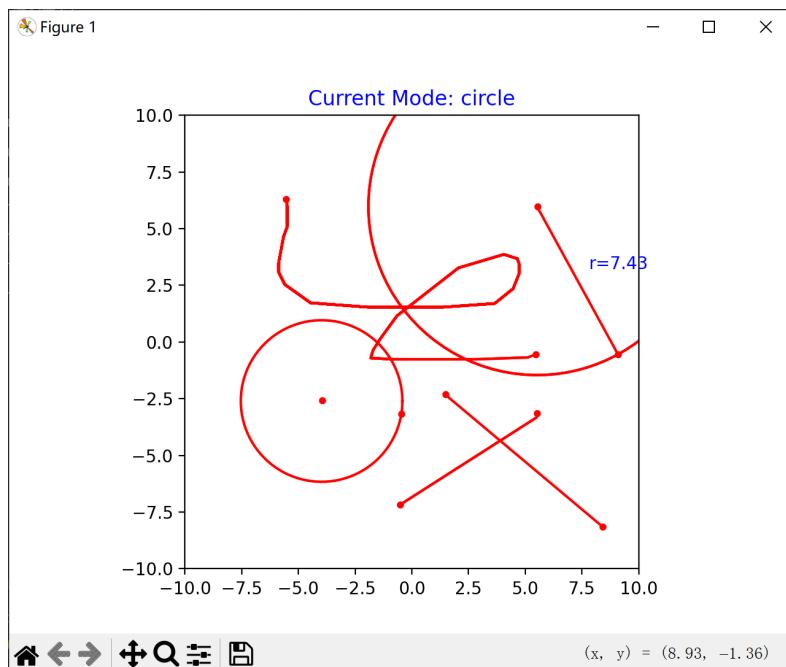


图 74: 7.3.3.1

其实这个程序还有一个早些版本，不用 class，而且也没加上实时展示当前状态以及缩放画布等功能，如下展示。但是不用 class 就意味着大量使用 global 来定义全局变量，通常这种全局变量特别多的程序中，使用 class 是一个比较好的选择。

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots()

```

```

5  pressed = 0
6  x_data = []
7  y_data = []
8  cir = 0
9  lin = 0
10 lines = []
11 radius = []
12 exist_radius = []
13 exist_lines = []
14 exist_linear=[]
15 circle_number = 0
16 line_number = 0
17
18 ax.set_xlim(-10, 10)
19 ax.set_ylim(-10, 10)
20 ax.set_aspect('equal')
21
22 def clickon(event):
23     global pressed, x_data, y_data, cir
24     if event.inaxes is not None and cir == 0:
25         ax.scatter(event.xdata, event.ydata, c='m', s=10)
26         pressed = 1
27         plt.draw()
28     if event.inaxes is not None and cir == 1:
29         ax.scatter(event.xdata, event.ydata, c='m', s=10)
30         ax.text(event.xdata + 0.5, event.ydata + 0.5, '({},{})'.format(
31             round(event.xdata, 2), round(event.ydata, 2)), fontdict={'fontsize': 10,
32             'fontweight': 'light', 'color': 'blue'})
33         pressed = 1
34     plt.draw()
35
36 def release(event):
37     global pressed, x_data, y_data, cir, circle_number, line_number,
38     exist_radius, exist_lines,exist_linear
39     if event.inaxes is not None and cir+lin == 0:
40         ax.scatter(event.xdata, event.ydata, c='m', s=10)
41         pressed = 0
42         plt.draw()
43         x_data = []
44         y_data = []
45     elif event.inaxes is not None and len(x_data) >= 2 and cir+lin == 1:
46         x1, y1 = x_data[0], y_data[0]

```

```

47     theta = np.linspace(0, 2 * np.pi, 100)
48     x_circle = x1 + r * np.cos(theta)
49     y_circle = y1 + r * np.sin(theta)
50     ax.plot(x_circle, y_circle, c='m') # 绘制圆
51     line_number += 1
52     exist_lines.append(lines[-1])
53     exist_linear.append(r)
54     if line_number >= 1:
55         for line in exist_lines[-line_number:]:
56             ax.add_line(line)
57     fig.canvas.draw()
58     plt.draw()
59     x_data = []
60     y_data = []
61     pressed = 0
62 elif len(x_data) < 2 or lin==1:
63     ax.scatter(event.xdata, event.ydata, c='m')
64     plt.draw()
65
66 def moving(event):
67     global pressed, x_data, y_data, cir, lines, radius, exist_lines,
68     exist_radius
69     if event.name == 'motion_notify_event' and pressed == 1 and event.
70     inaxes is not None and cir + lin == 0:
71         x_data.append(event.xdata)
72         y_data.append(event.ydata)
73         ax.plot(x_data, y_data, c='m')
74         plt.draw()
75     if event.name == 'motion_notify_event' and pressed == 1 and event.
76     inaxes is not None and cir + lin != 0:
77         x_data.append(event.xdata)
78         y_data.append(event.ydata)
79         x1, y1 = x_data[0], y_data[0]
80         x2, y2 = x_data[-1], y_data[-1]
81         if cir==1:
82             line, = ax.plot([x1, x2], [y1, y2], c='r')
83         else:
84             line, = ax.plot([x1, x2], [y1, y2], c='m')
85         ra = np.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
86         if cir == 1 and lin == 0:
87             r = ax.text((x1 + x2) / 2 + 0.5, (y1 + y2) / 2 + 0.5, 'r={}'.
88             format(round(ra, 2)), fontdict={'fontsize': 10, 'fontweight': 'light', 'color': 'blue'})
89         if cir == 0 and lin == 1:
90             r = ax.text((x1 + x2) / 2 + 0.5, (y1 + y2) / 2 + 0.5, 's={}'.

```

```

        format(round(ra, 2)), fontdict={'fontsize': 10, 'fontweight': 'light',
        'color': 'blue'})
    87     lines.append(line)
    88     radius.append(r)
    89     plt.draw()
    90     if len(lines) > 1:
    91         for line in lines[:-1]:
    92             line.remove()
    93         lines = lines[-1:] # 保留最后一个线条
    94         fig.canvas.draw()
    95     if len(radius) > 1:
    96         for r in radius[:-1]:
    97             r.remove()
    98         radius = radius[-1:] # 保留最后一个文本
    99         fig.canvas.draw()
100
101 def circle(event):
102     global cir, lin
103     if event.key == 'r':
104         cir = 1 - cir
105         lin = 0
106
107 def linear(event):
108     global lin, cir
109     if event.key == 'l':
110         lin = 1 - lin
111         cir = 0
112
113 def clear_all(event):
114     global pressed, x_data, y_data, cir, lines, radius, exist_lines,
115     exist_radius, circle_number, line_number
116     if event.key == 'delete':
117         ax.clear()
118         ax.set_xlim(-10, 10)
119         ax.set_ylim(-10, 10)
120         ax.set_aspect('equal')
121         fig.canvas.draw()
122         lines = []
123         radius = []
124         exist_radius = []
125         exist_lines = []
126         circle_number = 0
127         line_number = 0
128 cid1 = fig.canvas.mpl_connect('button_press_event', clickon)

```

```

129 cid2 = fig.canvas.mpl_connect('button_release_event', release)
130 cid3 = fig.canvas.mpl_connect('motion_notify_event', moving)
131 cid4 = fig.canvas.mpl_connect('key_press_event', linear)
132 cid5 = fig.canvas.mpl_connect('key_press_event', circle)
133 cid6 = fig.canvas.mpl_connect('key_press_event', clear_all)
134
135 plt.show()

```

而且想必读者在认真思考后会发现，这个先前版本的代码逻辑没有新版那么清晰，新版的代码基本做到了每个事件定义一个函数，而不是多个函数指向同一个事件（这么干多了容易出问题）。

同时在实际操作时本人发现，由于窗口出现后输入总会自动跳为中文，输入字母控制模式时总是有不必要的干扰出现（你懂的），因此改为由 shift 键切换模式更为方便，毕竟只有三个模式可切换。修改 pattern_change 函数的前面部分即可。修改后的 pattern_change 函数如下：

```

1 def pattern_change(event):    ### 键盘事件控制函数
2     if event.name=='key_press_event' and event.key=='shift':    ### 由 shift
3         state.shift_number+=1
4
5     if state.shift_number%3==2:    ### 直线
6         state.figure_pattern='linear'
7     elif state.shift_number%3==1:    ### 圆
8         state.figure_pattern='circle'
9     elif state.shift_number%3==0:    ### 一般曲线
10        state.figure_pattern='typical'
11    if event.key == 'delete':    ##### 清空 但是模式不变
12        ax.clear()
13        ax.set_xlim(-10, 10)
14        ax.set_ylim(-10, 10)
15        ax.set_aspect('equal')
16        state.pressed = 0
17        state.line_number = 0
18        state.circle_number = 0
19        state.x_list = []
20        state.y_list = []
21        state.lines = []
22        state.radius_values = []
23        state.exist_lines = []
24        state.circles = []
25        state.exist_circles = []
26    if event.key == 'up':
27        ax.set_ylim(ax.get_ylim()[0] + 0.1, ax.get_ylim()[1] + 0.1)

```

```
28 if event.key == 'down':
29     ax.set_ylim(ax.get_ylim()[0] - 0.1, ax.get_ylim()[1] - 0.1)
30 if event.key == 'left':
31     ax.set_xlim(ax.get_xlim()[0] - 0.1, ax.get_xlim()[1] - 0.1)
32 if event.key == 'right':
33     ax.set_xlim(ax.get_xlim()[0] + 0.1, ax.get_xlim()[1] + 0.1)
34 ax.set_title(f"Current Mode: {state.figure_pattern}", color='blue')
35 fig.canvas.draw()
```

7.3.4 拾取事件

例子见[8.10](#), 这就是一个经典的拾取事件展示。

8 详述

8.1 如何利用 Python 解方程

利用 Python 解方程的方法很多，包括使用 SymPy，使用 SciPy，牛顿法等等，这一部分我们主要介绍 SciPy 法，这是一种较为简单而且普适性强的方法，可以解决复杂的线形和非线性方程。

8.1.1 SciPy-fsolve 法简介

先看一个示例：

```
from scipy.optimize import fsolve

# 定义方程的函数形式
def func(x):
    return x**2 - 4

# 使用fsolve求解, 初始猜测值为1
solution = fsolve(func, 1)

print(solution) # 输出解
```

图 75: xs1.1.1

这个方法核心依靠 `fsolve` 来通过迭代法解出近似解，`fsolve` 接受两个输入，一个是方程的表达式，需要化为 $f(x)=0$ 的形式，并输入 $f(x)$ ，另一个是猜测值，或者叫初始值。猜测值的选择很重要，不恰当的猜测值可能导致方法不收敛，尤其是在复杂的非线性方程中；即使能够收敛，也可能导致收敛速度变慢。而且对于有多个根的方程，不同的初始值可能会导致收敛到不同的根。

`fsolve` 的工作原理如下：

1. 初始猜测值：`fsolve` 接受一个初始猜测值 `initial-guess`，这是一个数组，用于启动迭代过程。
2. 迭代过程：`fsolve` 会使用某种数值方法（通常是牛顿-拉弗森方法或其变体）来迭代地改进初始猜测值。在每次迭代中，`fsolve` 都会调用 `func` 函数，将当前的猜测值作为

输入，并得到一个结果数组。

3. 收敛条件：fsolve 会检查 func 返回的结果数组。如果这个数组中的所有元素都足够接近于零（即在预设的容忍度范围内），则认为已经找到了方程的根，迭代过程会停止。

4. 返回结果：fsolve 返回的是使得 func 返回的结果足够接近于零的输入值，也就是方程的根的近似值。

5. 多个根：如果方程有多个根，不同的初始猜测值可能会导致收敛到不同的根。因此，对于复杂的方程，可能需要从不同的位置启动多次 fsolve 来找到所有的根。

6. 不收敛：如果 fsolve 无法在预设的迭代次数内找到一个满足收敛条件的解，或者初始猜测值远离任何根，它可能会返回一个警告，并给出最后一个迭代的结果，这个结果可能不是方程的根。

要注意第三条，fsolve 既可以处理单个值，也可以处理多个值，就是说，当我们需要解单个方程时，return 的是一个应该为 0 的表达式，代入未知数会成为单个数值，fsolve 试图让这个值足够接近于零；而如果我们需要解该方程组，这里我们需要 return 一个由多个应该为 0 的表达式组成的 numpy 数组，fsolve 试图让这个数组中的每个值都足够接近于零。解方程组的示例如下：

```
1 import numpy as np
2 from math import log
3 from scipy.optimize import fsolve
4
5 def func(x):
6     x1,x2=x
7     return np.array([x1+x2+1,3*x1+2*x2])
8 solutions=fsolve(func, x0: [0,0])
9
10 print(solutions) # 输出解
```

图 76: xs1.1.2

输出结果是 [2. -3.]，是一个列表。

8.1.2 SciPy-fsolve 法的深入思考与改良

SciPy-fsolve 法不是完美的，初始猜测值选取不当很容易导致漏解以及错解，使用这一方法有诸多技巧，以下技巧均出于本人经验：

1. 可能发生错解的范围难以确定，受到多方面因素影响，但是通常猜测值与实际值在 1 以内可以确保收敛成功；

2. 对于简单的多解方程，比如一元二次方程，通常它的两个实根的中间值会成为一个分界线，大于分界线的解出较大解，小于分界线的解出较小解，（等于时可能倒向一侧也可能直接报错并返回会一个错解）总之几乎不可能直接返回所有解

有一种解决方案：核心思路是以 1 为步长，遍历所有解所在区间（这个区间需要自己估计，不过尽量往大处取也未尝不可，毕竟单个循环下这个计算量还是有限的），将遍历值作为初始值依次带入求解方程，将所有得到的解放入一个列表中，剔除相同的元素，得到的就是完整的解。

上述方法看似简单，实际情况要复杂得多，大多数方程的解不是整数，因此解的类型肯定得设为 float 浮点型，而浮点型有一个问题，如果它是有限的，末尾有可能会带一串无意义的数字，比如 $x^2 = 9$ 的结果有可能是 $x=3.000000008$ ，这些数字的出现完全随机，这就导致了如果我们用简单的“是否相等”进行判定，相同的解可能因为它的存在被判定为不同的解，因此，我们需要设置一个精度值，超过这个精度值可以判定异解，否则为同解。我之所以在这里称之为“精确值”而非“阈值”，因为解的精确性实际上是依赖于这个值的选择的，换句话说，为了剔除同解，我们牺牲了部分的精确性。以下是这个方法的完整代码展示及应用：

```
34 import numpy as np
35 from scipy.optimize import fsolve
36
37 def func(x):
38     return x**2 - 5
39
40 # 使用集合来存储解，自动去除重复
41 solutions_set = set()
42 for i in range(-10, 10, 1):
43     solution = float(fsolve(func, i)[0])
44     solutions_set.add(round(solution, 4))
45
46 # 将集合转换回列表
47 solutions = list(solutions_set)
48
49 print(solutions) # 输出解
```

图 77: xs1.2.1

solution 是每个 i 下得到的解，通过 round 函数我们能将解值保留指定的位数（也就是我们之前说的精确度），然后将经过保留小数操作的解放入 solution-set 集合，因为集合能自动去除重复元素，相同的解就都被剔除了，最后我们又将几何转回列表，得到了包含完整解而没有重复解的列表。

```
4.1.py"
[2.2361, -2.2361]

Process finished with exit code 0
```

图 78: xs1.2.2 Outcome

3. 如果要解一个多变量的方程组，尽量选择简单的线性关系交给 fsolve，复杂的关系留给解完方程后的代数运算，这样能大大提高成功率。我们来看一个例子：

$$\ln\left(\frac{P1}{94.6}\right) = \frac{dH_a}{8.314} \left(\frac{1}{298.15} - \frac{1}{T2} \right) \quad (1)$$

$$\ln\left(\frac{P2}{29.1}\right) = \frac{dH_b}{8.314} \left(\frac{1}{298.15} - \frac{1}{T2} \right) \quad (2)$$

$$xP1 + (1-x)P2 = 760 \quad (3)$$

其中 x 是第一个函数的自变量，需要在 (0, 1) 中遍历（密集取点），在循环中将每次取到的 x 代入上面 3 个方程，用这个方程组解出 P1, P2, T2 这 3 个变量。然后分别以 $x \rightarrow T2, y \rightarrow T2 (y = \frac{xP1}{xP1+(1-x)P2})$ 作线图。（将 T2, y 的历次计算值存入列表，然后交给 plot 函数）

3 个方程组直接解显然不如 2 个方程组 + 一个代数运算式，我们可以选择那 2 个含对数的复杂等式 (1),(2) 作方程组，选择简单的线性关系式 (3) 作为代数运算式，代码如图 xs.1.2.3, 效果如图 xs.1.2.4。为什么会这样？我猜想是对于复杂等式，fsolve 的正确率会大大下降，出现许多误解的点，而 plot 函数的本质是直线连接使用离散的点，因此部分点的严重偏移导致了整张图变得十分抽象。此外，这种选择还带来一个问题，x 在遍历 (0, 1) 时是能取到端点的，这使得 $\frac{760-xP1}{1-x}$ 有分母为 0 的风险，因此在这段代码中我将 1-x 改为了 1.0001-x。

但是，如果我们选择 (1) (3) 作为方程组，(2) 作为代数运算式，fsolve 处理的方程组得到了简化，准确率也随之提升。同时，我们成功规避了 1-x 跑到分母上的问题，因此这种选择更优。代码与效果如图 xs.1.2.5, xs.1.2.6。

```

P_a=94.6
P_b=29.1
dH_a=30.8*10**3
dH_b=33.2*10**3
x = np.linspace( start: 0, stop: 1, num: 100)
x_list = x.tolist()
y_list = []
T2_list = []
for x in x_list:
    1 usage
    def fun(P_and_T):
        P1, T = P_and_T
        # 计算两个方程的值并合并为一个数组
        return np.array([(8.314 / dH_b) * np.log(np.abs((760 - x * P1) / ((1.0001 - x) * 29.1))) - 1 / 298.15 + 1 / T,
                        (8.314 / dH_a) * np.log(np.abs(P1 / 94.6)) - 1 / 298.15 + 1 / T])
    initial_guess = [300, 110]
    solution = fsolve(fun, initial_guess)
    P1, T2 = solution
    T2 -= 273.15
    P2 = (760 - x * P1) / (1.0001 - x)
    y = P1 / (P1 + P2)
    y_list.append(y)
    T2_list.append(T2)
plt.plot(*args: x_list, T2_list, color='r')
plt.plot(*args: y_list, T2_list, color='g')
plt.show()

```

图 79: xs.1.2.3

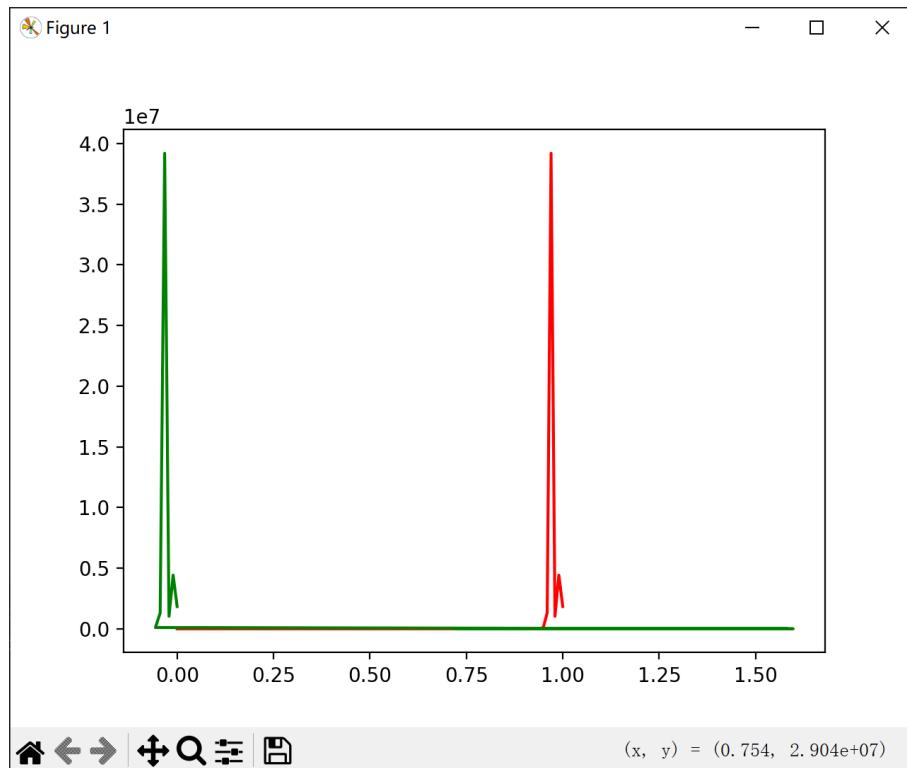


图 80: xs.1.2.4

```

P_a=94.6
P_b=29.1
dH_a=30.8*10**3
dH_b=33.2*10**3
x=np.linspace( start: 0, stop: 1, num: 100)
x_list=x.tolist()
y_list=[]
T2_list=[]
for x in x_list:
    1 usage
    def fun(P_1_and_2):
        P1, P2 = P_1_and_2
        return np.array([(8.314 / dH_a) * np.log(np.abs(P1 / 94.6)) - (8.314 / dH_b) * np.log(np.abs(P2 / 29.1)),
                        x * P1 + (1 - x) * P2 - 760])
    initial_guess = [10,10]
    solution = fsolve(fun, initial_guess)
    P1, P2 = solution
    T=1/(1/298.15-8.314*np.log(abs(P1/94.6))/dH_a)-273.15
    T2_list.append(T)
    y = x*P1 / (x*P1 + (1-x)*P2)
    y_list.append(y)
plt.plot(*args: x_list, T2_list, color='r',label='Boiling point')
plt.plot(*args: y_list, T2_list, color='g',label='Mole fraction in vapor,x_benzene(g)')
plt.legend()
plt.show()

```

图 81: xs.1.2.5

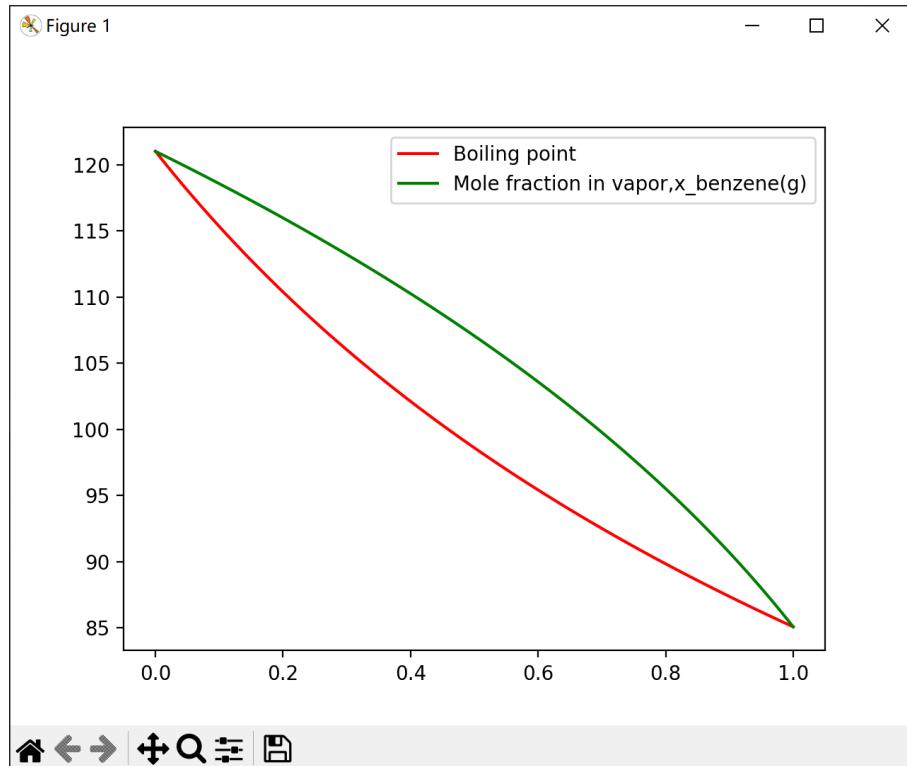


图 82: xs.1.2.6

8.2 numpy 数组与列表，元组

8.2.1 区别与联系

它们的区别如下表所示：

	numpy 数组	列表	元组
表中数据类型	必须相同	可以不同	可以不同
内存分配	连续存储	不连续存储	不连续存储
能否添加元素（改变大小）	否	能	否
能否改变元素	能	能	否
向量化计算	支持	不支持	不支持

除此之外，它们也有一些共同点，比如都支持索引和切片操作，都可以迭代。

numpy 数组相比元组和列表有许多优点，首先是 NumPy 数组在数值计算方面通常比列表和元组更快，特别是在处理大型数据集时。此外，NumPy 数组由于其连续存储和固定数据类型，通常比列表和元组更节省内存。其支持向量化计算其实也是得益于连续存储这一特性。（连续内存分配是指数据在内存中存储时，其物理地址是连续的。这意味着，如果一个数组的第一个元素存储在内存地址 A，第二个元素存储在 A+1，以此类推，那么这个数组就是连续存储的。）

最后，numpy 数组有一套广播机制，它允许不同形状的数组在数学运算中协同工作，而不需要显式地匹配它们的形状。广播遵循以下规则：

1. 如果两个数组的维数不同，形状较小的数组会在其前面补 1，直到两个数组的形状长度相同。
2. 如果两个数组的形状在某个维度上不匹配，那么较小的形状会从 1 开始广播，直到匹配较数组的形状。
3. 如果一个数组在某个维度上的大小是 1，而另一个数组在该维度上的大小大于 1，那么大小为 1 的数组会沿着这个维度复制其值，以匹配另一个数组的形状。

下图 xs2.1.1 是广播机制的一个示例。数组 a 形状为 (3,)，是一个包含了 3 个元素的一维数组，数组 b 形状为 (3, 1)，是一个包含了一个元素的三维数组。形状 (shape) 是一个整数的元组，用来描述数组在每个维度上的大小。对于一维数组来说，形状只包含一个整数，即数组的长度。（对于一维数组，这里的逗号是必需的，即使只有一个数字，也要用逗号表示这是一个元组。在 Python 中，单个元素的元组需要在数字后面加上逗号来区分它和普通的数字。）

```

import numpy as np

a = np.array([1, 2, 3]) # shape (3, )
b = np.array([[1], [2], [3]]) # shape (3, 1)

# 广播使得两个数组可以进行元素级别的运算
c = a + b # 结果 c 的形状是 (3, 3)
print(c)

# 输出:
# [[2 3 4]
#  [3 4 5]
#  [4 5 6]]

```

图 83: xs.2.1.1

从效果上讲，广播机制使得上面例子中的一维数组 `array([1,2,3])` 在另外两个维度上复制了这个维度，形成 `array([1,2,3],[1,2,3],[1,2,3])` 这样一个 (3, 3) 的三维数组。同时 `array([1],[2],[3])` 在每个维度上大小为 1，在广播下每个维度的值沿着改维度复制直到形成一个可以匹配的 (3, 3) 数组。所以最后是 `array([1,2,3],[1,2,3],[1,2,3])+array([1, 1, 1],[2, 2, 2],[3, 3, 3])`。

8.2.2 相互转换

numpy 数组与列表、元组可以相互转换：

数组-> 列表： `list-from-array = array.tolist()`

数组-> 元组： `tuple-from-array = tuple(array)`

列表-> 数组： 都是用 `np.array(列表/元组)`

元组-> 列表： `list-from-tuple = list(tuple)`

数组-> 元组： `tuple-from-list = tuple(list)`

(注意转换是单向的：一旦你将列表转换为元组，你就不能将元组转换回原来的列表，因为元组是不可变的。任何对元组的修改都会创建一个新的对象。)

8.3 图例的高级设置

8.3.1 位置

我们之前说过，图例位置的设定是 loc=' ', 通常默认的是 ‘best’，也就是对图像影响最小的地方，除此之外，通过传递字符串或元组给 loc 参数，我们可以人为设定图例出现在图中的什么位置。

字符串：

常用的字符串参数包括‘best’、‘upper right’、‘upper left’、‘lower left’、‘lower right’、‘right’、‘center left’、‘center right’、‘lower center’、‘upper center’、‘center’等。

元组：

可以指定图例的确切位置，如 loc = (x, y)。注意这里的 x,y 是相对横纵坐标轴的相对位置，比如 x=0.5 是指在横轴一半的位置，因此 x,y 值必须在 (0, 1) 内才是有效的。

此外，注意 plt.legend() 函数每次被调用时都会清除之前的位置设置，并在当前图表上绘制一个新的图例。也就是说，如果你多次调用 plt.legend()，只有最后一次调用的图例会被显示，之前的图例会被覆盖。所以，不要试图干出图 xs.3.1.1 所示的这种事，你最终只会得到一个图例。

```
for i in np.arange(0,1,10):
    for j in np.arange(0,1,10):
        plt.legend(loc=(i,j))
```

图 84: xs.3.1.1

这也提醒我们，关于图例的设置必须集中在一个 plt.legend() 里边，否则一部分设置就会被另一部分覆盖。

8.3.2 边框与背景

frameon 参数控制是否显示图例边框，默认为 True，如果改为 False 则无边框。

注意只有有边框的时候才可以设置背景——背景颜色和边框颜色！

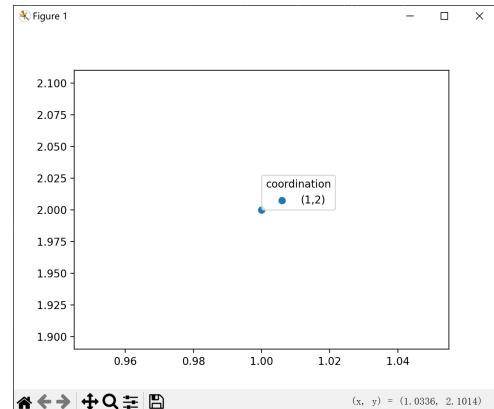
edgecolor 用于设置图例边框的颜色，facecolor 用于设置图例背景的颜色，传递给它们的颜色可以是简写。

8.3.3 标题与字体大小

标题的设置参数是 title，它与 plot 里设置的 label 不同，title 位于整个图例框上面，标示整个图例的名字，label 只是这个图例中每条线或点的标识。如图 xs.3.3.1 和 xs.3.3.2 所示，coordination 为标题，而 (1, 2) 是 label。

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 x=1
4 y=2
5 plt.scatter(x,y,label='(1,2)')
6 plt.legend(loc=(0.5,0.5),frameon=1,title='coordination')
7 plt.show()
```

(a) xs.3.3.1

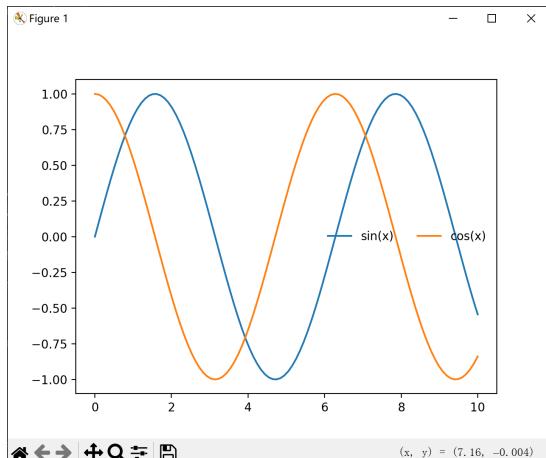


(b) xs.3.3.2

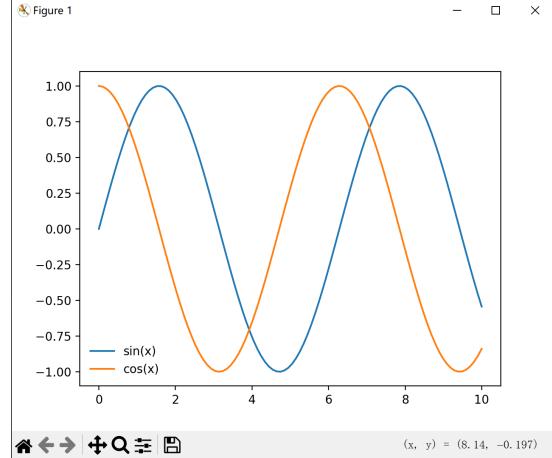
fontsize 参数用于设置图例中的字体大小。通常 fontsize=10 12 比较正常。

8.3.4 列数

ncol 参数用于设置图例分为几列展示。ncol 的取值必须是大于 0 的整数，若 ncol 大于总图例数，所有图例会被以最大列数呈现（就是排成一行）而不会报错，但是小于等于 0 一定会报错。不同列数的区别如图 xs.3.4.1 和 xs.3.4.2 所示：



(a) xs.3.4.1



(b) xs.3.4.2

8.4 scatter 函数的高级设置

8.4.1 点的形状

通过 `marker= ''`, 常用的点形状选项包括: '`o`' (圆形), '`s`' (正方形), '`^`' (上三角形), '`v`' (下三角形), '`x`' (X 形) 等。示例:

```
plt.scatter(x, y, marker = ' s')
```

8.4.2 点的透明度

通过 `alpha=`, `alpha` 值为 0~1, 越小越透明, 0 表示完全透明, 1 表示不透明。示例:

```
plt.scatter(x, y, alpha = 0.5)
```

8.4.3 边缘颜色

`edgecolors` 参数用于设置点的边缘颜色, 可以是单一颜色或颜色序列, 示例:

```
plt.scatter(x, y, edgecolors = ' red')
```

如果想让边缘消失, 我们可以将边缘颜色设为无色, 即 `edgecolors = 'none'`

8.4.4 边缘宽度

`linewidths` 参数用于设置点边缘的宽度。当我们修改透明度时, 会注意到边缘有一圈深色的边, `linewidth` 设置的就是那个边缘的宽度。示例:

```
plt.scatter(x, y, linewidths = 2)
```

8.4.5 颜色映射

我们首先要介绍一个函数——归一化函数 (Normalize), 简单来说, 它的功能是把一个数值范围映射到另一个数值范围, 在颜色映射中, 我们需要用它来把数据值映射到颜色映射表上 Normalize 函数的一般用法如下:

```
import matplotlib.colors as mcolors
```

```
norm = mcolors.Normalize(vmin = 10, vmax = 50)
```

由此不难看出, `Normalize` 函数是 `mcolors` 模块的一个函数, 而 `mcolors` 库需要从 `matplotlib` 单独导入。在 `Normalize` 中我们指定 2 个参数, `vmin` 和 `vmax`, 我们可以把它们

理解为一个框来框取需要映射的范围，在 vmin vmax 范围的数据会被 Normalize 线形映射到 0 1 范围内，超过该范围的数据，小于的会被统一映射为 0，大于的会被统一映射为 1。但是在有些版本的 matplotlib 中，这个函数将范围以外的数据做了等间距延伸，如下图 xs.4.4.5.1, xs.4.4.5.2 所示：

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 import matplotlib.colors as mcolors
5 x=[1,2,3,4,5,6,7,8,9,10]
6 norm=mcolors.Normalize(vmin: 1, vmax: 5)
7 y=norm(x)
8 print(y)

```

(a) xs.4.4.5.1

```
C:\Users\金御风\PycharmProjects\pythonProject.1\venv
[0.    0.25 0.5  0.75 1.    1.25 1.5  1.75 2.    2.25]
```

(b) xs.4.4.5.2

但是如果我们在 Normalize 中添加： clip=True 或者 clip=1，Normalize 就会遵循上面说的第一种方式映射，如图 xs.4.4.5.3, xs.4.4.5.4 所示：

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 import matplotlib.colors as mcolors
5 x=[1,2,3,4,5,6,7,8,9,10]
6 norm=mcolors.Normalize(vmin: 1, vmax: 5, clip=True)
7 y=norm(x)
8 print(y)

```

(a) xs.4.4.5.3

```
C:\Users\金御风\PycharmProjects\pythonProject.1\venv
[0.    0.25 0.5  0.75 1.    1.    1.    1.    1.    ]
```

(b) xs.4.4.5.4

但是如果我们想把已知数据映射到其他范围，不仅仅是 (0, 1) 呢？运用一点点数学方法，即可得到如下公式，其中 a,b 为新范围的上界与下界, λ 是 (0, 1) 范围的映射值， x 是新范围的映射值：

$$x = \lambda(b - a) + a$$

很有意思的是，Normalize 函数对我们很友好，可以接受 numpy 数组，列表，元组类型的输入，返回值是 numpy 数组，这意味着我们可以直接对整个返回值套用以上公式，然

后直接得到映射到指定范围的数组。一个简单的示例如下，我们选择性地将 1~10 中的 1~5 映射到 100~200：

```
1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 import matplotlib.colors as mcolors
5 x=np.array([1,2,3,4,5,6,7,8,9,10])
6 norm=mcolors.Normalize(vmin=1, vmax=5, clip=True)
7 y=norm(x)*100+100
8 print(y)
```

(a) xs.4.4.5.3

```
[100. 125. 150. 175. 200. 200. 200. 200. 200.]
```

(b) xs.4.4.5.4

介绍完 Normalize，我们接着说说颜色映射，这里我们只探讨在 scatter 函数里使用颜色映射，实际上在 matplotlib 中许多函数都支持颜色映射，其余这些我们放在详述[8.9](#)里说。scatter 函数中使用颜色映射示例：

```
norm = mcolors.Normalize(-20, 20, clip=0)
```

```
plt.scatter(x, y, c=z, cmap='viridis' norm=norm)
```

因为我们是将一个与函数值相关的值传递给 c，所以实际使用中我们常常把 z 就设成 y，通过使用 Normalize 函数，我们将指定范围的函数值映射成一个新范围 norm，然后将这个 norm 赋值给 scatter 中的 norm 变量，这个 norm 范围其实是有颜色渐变的区域。至于 cmap= 的对象，它是一个系统自带的映射表，或者说一种映射规则，选择不同的映射表可以产生不同的效果。常用的映射表有：

1. 顺序色彩映射：viridis、plasma、inferno、magma、Blues 等
2. 发散色彩映射：coolwarm、RdBu、PiYG、PRGn 等。
3. 定性色彩映射：Set1、Set2、Set3、Accent、Paired 等。
4. 循环色彩映射：twilight、hsv 等。

其实这个 norm 也可以不设，单纯是 `plt.scatter(x, y, c=z, cmap='viridis' norm=norm)` 也能产生颜色渐变，但是效果会稍有区别。

8.5 简写问题

8.5.1 哪些参数可以简写

- 颜色、线型和标记：在绘制线条时，可以使用简写来同时指定颜色、线型和标记。例如，'r-o' 表示红色实线和圆圈标记。
- 导入模块：我们在程序开头写的 `import matplotlib.pyplot as plt` 其实就是将 `plt` 作为 `Matplotlib.pyplot` 的简写。

8.5.2 如何简写

按理说这一部分应该放在“简写的使用规则”之后的，但是为了方便在后面使用一些简写的例子，还是先说如何简写。

颜色简写

颜色	全称	简写
红色	red	r
蓝色	blue	b
绿色	green	g
青色	cyan	c
品红色	magenta	m
黄色	yellow	y
黑色	black	k
白色	white	w

(a) 基本颜色名称和缩写

颜色	全称
紫色	purple
橙色	orange
粉色	pink
灰色	gray
青色	teal
银色	silver
金色	gold

(b) 额外颜色名称

图 90: xs.5.2.1

要注意的是，有些颜色可以简写，而另一些没有相应的简写，只能用全称来指定。（以及不要问我为什么青色有两种）

线型简写

这个其实我们之前在 *Pyplot* 绘图的基本方法讲过了，这里再提一次：

实线	'solid'	'—'
虚线	'dashed'	'—'
点线	'dotted'	'··'
点划线	'dashdot'	'-·'
无线条（仅显示标记）	'None'	''

点形简写

点形简写之前也提到过，这里再说一次。

标记	描述
'o'	圆圈标记
's'	正方形标记
'D'	菱形标记
'^'	上三角标记
左三角标记	
'>'	右三角标记

8.5.3 简写的使用规则

可简写可不简写

1. pyplot 中的大多数函数和 axes 中的部分都是既支持简写也支持全称的，其中以 plot 函数最为灵活。作为 matplotlib 中最常用的函数之一，它既可以接受简写形式的格式字符串（例如'b-o' 表示蓝色实线和圆圈标记），也可以接受详细的参数（例如 color='blue', linestyle='-', marker='o' ）。

简单来说，使用 pyplot 类的函数时基本可以放心大胆地简写，使用 axes 类时得注意点。

2. 有些参数本身可简写可不简写，如颜色 "color=" 可以简写为 "c="，还有线条宽度 linewidth 可以简写为 lw，标记大小 markersize 可以简写为 ms。

只能简写

1. 颜色简写：在使用格式化字符串时，颜色值必须使用简写。

举个例子：plt.plot(x, y, 'b') 是正确的，而 plt.plot(x, y, 'blue') 反而会报错，因为这里颜色参数的指定使用了格式化字符串，因此颜色值必须简写为'b'。

2. 有些参数的值，如标记 ('marker')，只有简写形式，没有对应的关键字参数。

3. 有些参数本身只能简写，比如图例的位置只能写作"loc= "。

不能简写

1. 复杂的参数设置：当需要设置的参数较为复杂，或者需要详细配置时，应使用完整的参数名称，而不是简写。（??）

2. 格式字符串位置错误：格式字符串必须紧跟在 x、y 数据后面，不能放在其他参数后面。否则只能用全称。

举个例子：plt.plot(x, y, 'r-o') 是正确的，格式字符串'r-o'紧跟在 x,y 后面，而 plt.plot(x, y, label='data', 'r-o') 就是错误的。

3. 有些参数本身不能简写，如：标记 marker，线形 linestyle，与图例边框及背景有关的 frameon、edgecolor、facecolor，图例字体大小 fontsize 等。

注意：这里真的非常非常容易混淆！！！“只能简写”部分提到的格式化字符串和这里的格式字符串其实不是一个概念。格式化字符串原本指的是一种在编程中用于生成具有特定格式的字符串的方法。它允许程序员在字符串中嵌入变量，并在运行时替换这些变量为具体的值，从而生成符合预期格式的输出。不同的编程语言有不同的格式化字符串的语法和方法。在 Python 中，主要有百分号 (%) 格式化, str.format(), f-string (格式化字符串字面量) (Python 3.6+) 三种方法，分别如下：

```
name = "Kimi"  
age = 30  
print("Hello, %s. You are %d years old." % (name, age))
```

输出： Hello, Kimi. You are 30 years old.

图 91: xs.5.3.1

```
name = "Kimi"  
age = 30  
print("Hello, {}. You are {} years old.".format(name, age))
```

输出： Hello, Kimi. You are 30 years old.

图 92: xs.5.3.2

```
name = "Kimi"  
age = 30  
print(f"Hello, {name}. You are {age} years old.")
```

输出： Hello, Kimi. You are 30 years old.

图 93: xs.5.3.3

你可能会说，之前例子中的 `plt.plot(x, y, 'b')` 不属于上面任何一种方法啊，的确。使用'b'这样的字符串来指定颜色是一种特殊的语法，它是 `plot()` 函数特有的简写方式。这种简写方式不是 Python 字符串格式化的一部分，而是 Matplotlib 库提供的一种快捷方式，用于设置线条的颜色、线型和标记。

你可以认为这种 Matplotlib 的格式化字符串的标志就是省略了参数 `color=`，同时注意这种简写方式是硬编码在 Matplotlib 中的，所以它只在 `plot()` 函数中有效。

而格式字符串其实就是那些用来指定颜色、线型和标记等绘图样式的简写字符串，格式字符串的组成部分可以两个或者三个组合在一起，例如'r-o'表示红色实线和圆圈标记。(通常按照“颜色 + 线形 + 点型”的顺序)；此外，在 `scatter` 函数中，我们常常经常使用简写形式例如'bo'来表示蓝色圆圈。这算是一种利用格式字符串的特殊的格式化字符串方法，Matplotlib 中特有的方法。

但不管怎么说，无论单个还是两三个组合，只要是省略了参数的格式字符串用法，都必须直接跟在 `x,y` 后面。(或者跟在指定散点坐标的 [] 后面)

8.6 Pyplot 绘图的其他方法

因为是简要介绍，这部分内容都采用结合具体示例讲解各段代码功能的形式。

8.6.1 箱型图

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 speeds = [39.6, 28.6, 42.2, 35.4, 30.7, 28.7, 30.6, 29.4, 31.0, 28.5, 35.6, 37.0, 36.5, 24.7, 27.1,
4           31.7, 27.4, 34.5, 36.0, 28.9, 27.6, 25.9, 39.4, 37.0, 27.8, 43.9, 38.6, 33.1, 30.7, 31.1,
5           35.2, 39.6, 38.9, 29.7, 37.5, 26.3, 27.0, 25.2, 33.2, 28.2, 34.3, 40.8, 26.4, 36.1, 43.1,
6           35.8, 26.5, 29.1, 37.7, 42.2, 27.2, 35.7, 28.9, 32, 27.3, 27.6, 30.0, 29.6, 29.7, 28.5]
7 plt.boxplot(speeds, notch=True, vert=True, patch_artist=True)
8 plt.show()
```

图 94: xs.6.1.1

1. `speeds`：这是传递给 `boxplot()` 函数的主要参数，它是一个数据集合，可以是列表、元组、NumPy 数组、pandas DataFrame 或 Series。数据集合可以是单变量的（单个列表或数组）或多变量的（列表的列表或二维数组）。
2. `notch`：这是一个布尔值参数，用于决定箱型图中的箱体是否带有“缺口”。缺口表示中位数的置信区间，可以帮助比较不同箱型图的中位数是否显著不同。如果设置为 `True`，则箱体会有一个缺口；如果设置为 `False`，则箱体是完整的矩形。
3. `vert`：这也是一个布尔值参数，用于指定箱型图是垂直绘制 (`True`) 还是水平绘制 (`False`)。默认值是 `vert=True`，即垂直绘制。

4. `patch_artist` : 这是一个布尔值参数，用于决定是否用填充色块来绘制箱体和触须。如果设置为 `True`，则箱体和触须会被填充；如果设置为 `False`，则它们是空心的。填充色块可以增强视觉效果，使得箱型图更加醒目。

(但是填充色块的颜色仅靠 `pyplot` 中的方法无法设定，这也是 `pyplot` 相较于 `axes` 的局限性的体现)

以下是这段代码的效果图：

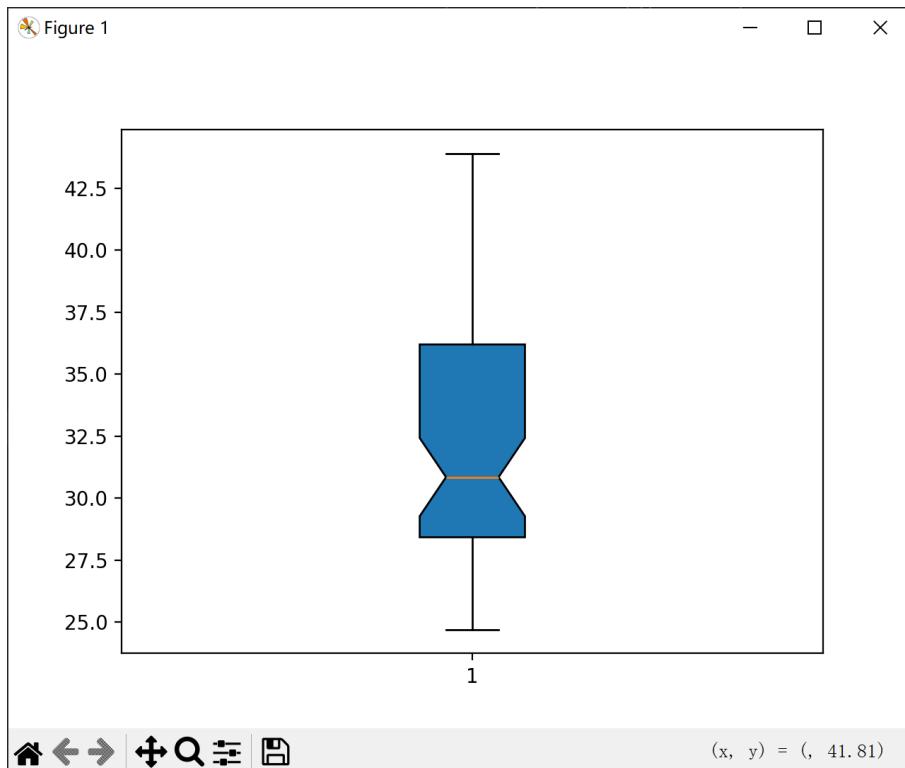


图 95: xs.6.1.2

这个函数还有其他一些参数，比如：

1. `whiskerprops` : 用于设置触须的样式，如颜色、线型等。
2. `capprops` : 用于设置箱型图上“帽子”（即表示异常值的点）的样式。
3. `boxprops` : 用于设置箱体的样式。
4. `medianprops` : 用于设置中位数线的样式。
5. `flierprops` : 用于设置异常值点的样式。
6. `manage_ticks` : 用于决定是否自动管理刻度标签。
7. `labels` : 用于设置箱型图每个箱体的标签。

8.6.2 阶梯图

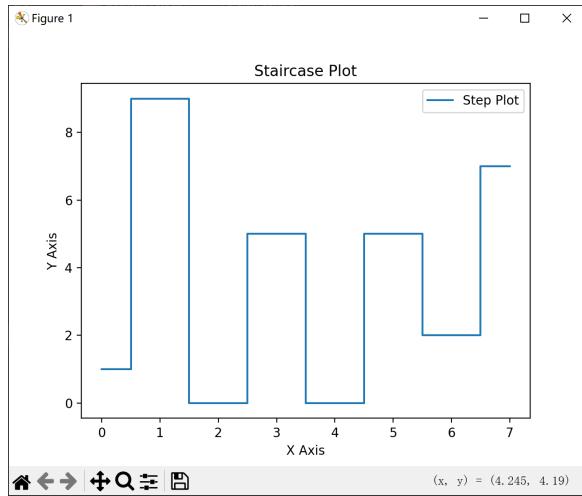
```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 # 生成随机数据
5 x = np.arange(8)
6 y=[1, 9, 0, 5, 0, 5, 2, 7]
7 # 绘制阶梯图
8 plt.step(x, y, where='mid', label='Step Plot')
9 plt.xlabel('X Axis')
10 plt.ylabel('Y Axis')
11 plt.title('Staircase Plot')
12 plt.legend()
13 plt.show()
```

图 96: xs.6.2.1

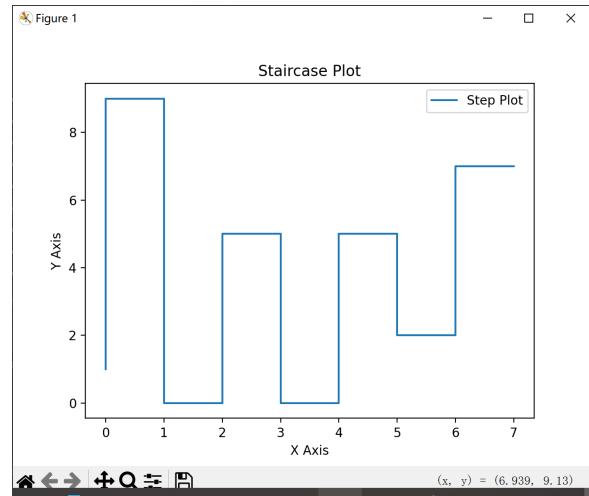
阶梯图很容易与直方图 hist(histtype=step) 混淆，其实二者有本质区别，step 类直方图会创建一个阶梯状的直方图，其中每个条形的顶部是平的，条形之间没有间隙。这种类型的直方图通常用于显示数据的分布，而不是连续变化的数据。而阶梯图则是将数据点用直线段连接起来，形成一个阶梯状的图形。阶梯图通常用于显示随时间变化的离散数据。阶梯图的本质更像是折线图。

where 参数决定了阶梯图的阶梯位置，它影响垂直线段与水平线段的连接方式。where 参数有三个可选值：'pre'、'post' 和'mid'。下面是每个值的具体含义：

1. 'pre'：这是默认值。当 where='pre' 时，每个阶梯图的垂直线段会在每个数据点之前开始，即在前一个数据点的 y 值处开始绘制水平线段，直到当前数据点的 y 值。
2. 'post'：当 where='post' 时，每个阶梯图的垂直线段会在每个数据点之后开始，即在当前数据点的 y 值处开始绘制水平线段，直到下一个数据点的 y 值。
3. 'mid'：当 where='mid' 时，每个阶梯图的垂直线段会在每个数据点的中点开始，即在两个连续数据点 y 值的中间位置开始绘制水平线段，这样可以使得阶梯图的每个台阶正好位于两个数据点的中间位置。



(a) xs.6.2.2 where='mid'



(b) xs.6.2.3 where='pre'

8.6.3 茎叶图

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 # 生成随机数据
5 speeds = [39.6, 28.6, 42.2, 35.4, 30.7, 28.7, 30.6, 29.4, 31.0, 28.5, 35.6, 37.0, 36.5, 24.7, 27.1,
6           31.7, 27.4, 34.5, 36.0, 28.9, 27.6, 25.9, 39.4, 37.0, 27.8, 43.9, 38.6, 33.1, 30.7, 31.1,
7           35.2, 39.6, 38.9, 29.7, 37.5, 26.3, 27.0, 25.2, 33.2, 28.2, 34.3, 40.8, 26.4, 36.1, 43.1,
8           35.8, 26.5, 29.1, 37.7, 42.2, 27.2, 35.7, 28.9, 32, 27.3, 27.6, 30.0, 29.6, 29.7, 28.5]
9 # 绘制茎叶图
10 plt.stem(speeds)
11 plt.xlabel('Data')
12 plt.ylabel('Value')
13 plt.title('Stem Plot')
14 plt.show()

```

图 98: xs.6.3.1

这个茎叶图……说真的没什么好设定的，其实 matplotlib 没有专门化茎叶图的函数，这里是通过 stem 函数创建类似于茎叶图的图形。stem() 函数主要用于创建条形图，但是可以通过调整参数来模拟茎叶图的效果，但是它不会像传统的茎叶图那样显示数据的茎和叶。

如果你需要创建一个真正的茎叶图，你可能需要使用其他库，比如 seaborn。

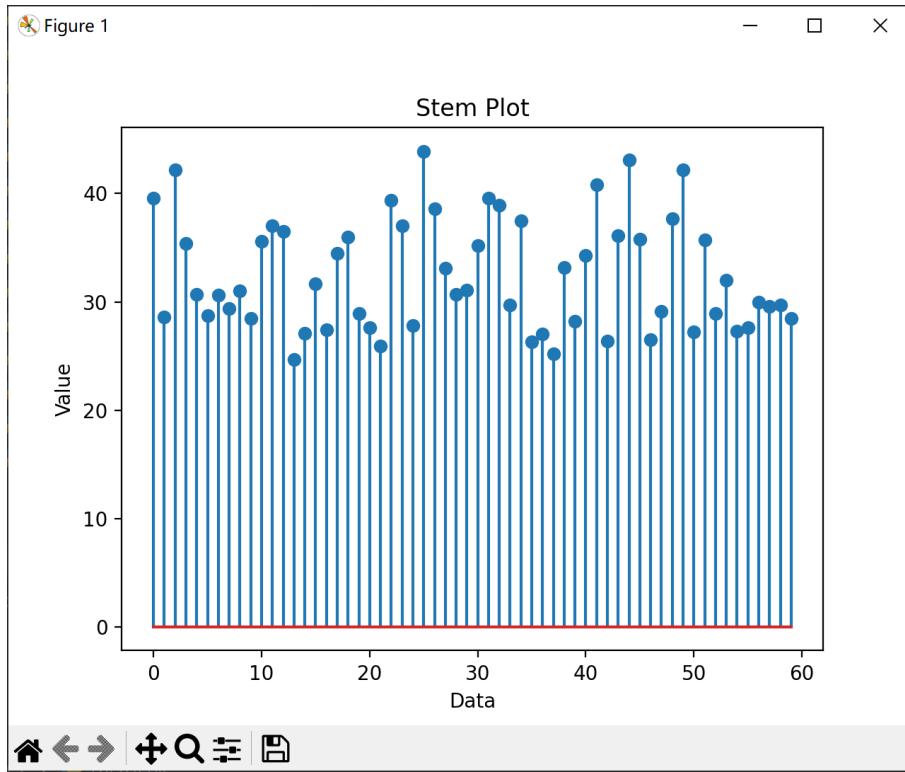


图 99: xs.6.3.2

8.6.4 饼图

```

1 import matplotlib.pyplot as plt
2
3 # 数据
4 sizes = [215, 130, 245, 210]
5 labels = ['Python', 'C++', 'Ruby', 'Java']
6 colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
7
8 # 绘制饼图
9 plt.pie(sizes, labels=labels, colors=colors, autopct='%.1f%%', startangle=140)
10 plt.title('Pie Chart')
11 plt.show()

```

图 100: xs.6.4.1

pie() 首先接受数据列表，据此计算出各类所占百分比。注意我们向 pie() 传递的前三个列表：数据、图例标签、颜色元素数都一样，且一一对应。1. autopct='%.1f%%'：这个参数用于在饼图的每个扇区中显示百分比。'%.1f%%' 是一个格式化字符串，表示保留一位小数的百分比值。

在 Python 的字符串格式化中，'%.1f%%' 是一个格式化字符串，用于将一个浮点数格式化为保留一位小数的百分比值。这里的% 是格式化操作符，1.1f 指定了浮点数

的格式，其中 1 表示整数部分保留一位，.1 表示小数部分保留一位，f 表示浮点数。最后的 %% 表示输出一个百分号（因为在字符串格式化中，% 是一个特殊字符，用于引入格式化指令。如果你想在字符串中实际打印一个% 符号，你需要使用两个% 来转义它，即%%）。

2. startangle=140：这个参数用于设置饼图的起始角度。饼图是从 x 轴正方向（通常是 3 点钟方向）开始绘制的，startangle 参数允许你改变起始角度。在这个例子中，startangle=140 意味着饼图将从 140 度的位置开始绘制，这通常是为了使饼图更加美观或者符合特定的布局需求。

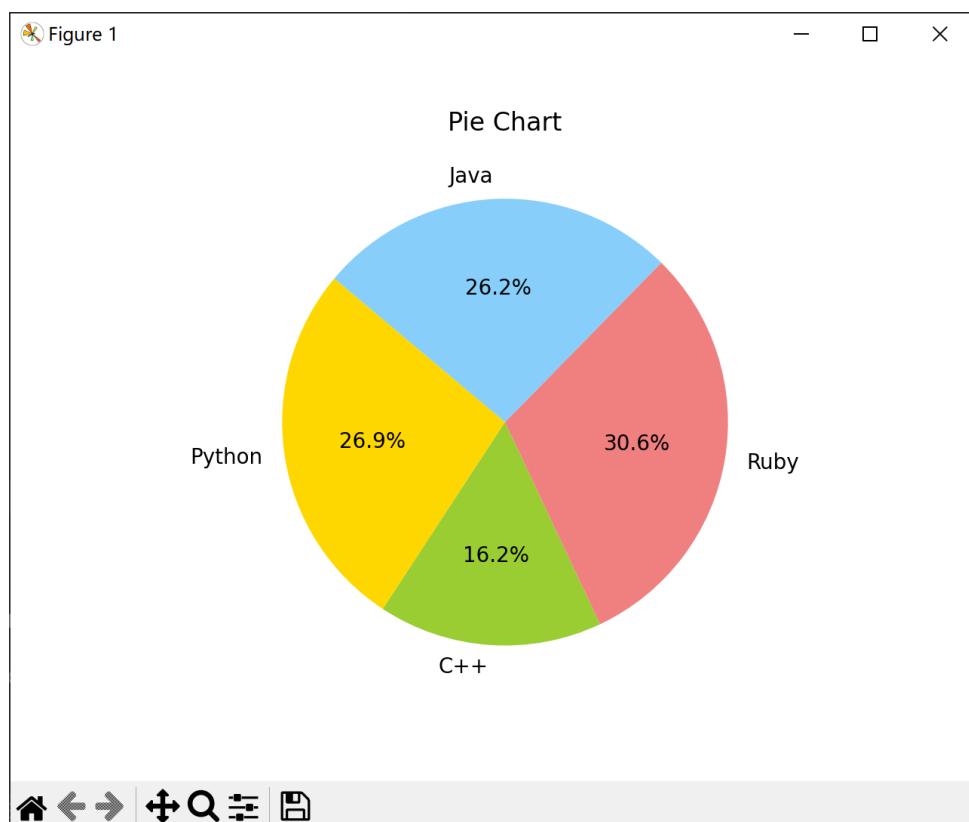


图 101: xs.6.4.2

8.6.5 两曲线间的颜色填充

```
import matplotlib.pyplot as plt
import numpy as np
# 创建一些数据
x = np.linspace( start: 0 , stop: 10 , num: 100)
y1 = np.sin(x)
y2 = np.cos(x)
# 绘制两条线，并为它们添加标签
plt.plot(*args: x, y1, 'r-' , label='sin(x)')
plt.plot(*args: x, y2, label='cos(x)')
plt.fill_between(x, y1, y2, where = (y1 > y2), color='yellow', alpha=0.3)
# 添加图例，设置无边框，两列显示
plt.legend(frameon=0, ncol=1, fontsize=10, edgecolor='b', facecolor='y')
plt.show()
```

图 102: xs.6.5.1

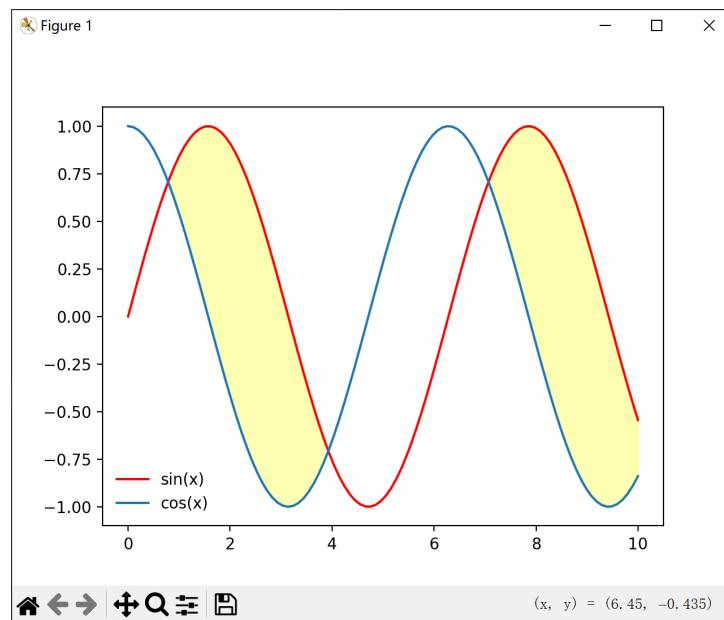


图 103: xs.6.5.2

注意 $x, y1, y2$ 是 3 个长度严格相等的数组，所以我们不能通过改变前面那个 x 来限制填充区域（比如传递给它一个新数组）。Instead，我们需要设置后面那个 $where$ ， $where$ 参数接受的其实是一个布尔数组，长度也和 $x, y1, y2$ 严格相等（可理解为与所取点数一样多的 True 或 False），最终 True 的那些区域会被上色。因此在吧这个例子中，只有 $y1$ 在 $y2$ 上方的区域被填充颜色了。

理解了这些，我们就可以通过逻辑语句调控上色区域，比如令 $where = (y1 >$

$y2) \& (x <= 3)$ 可以得到以下效果：

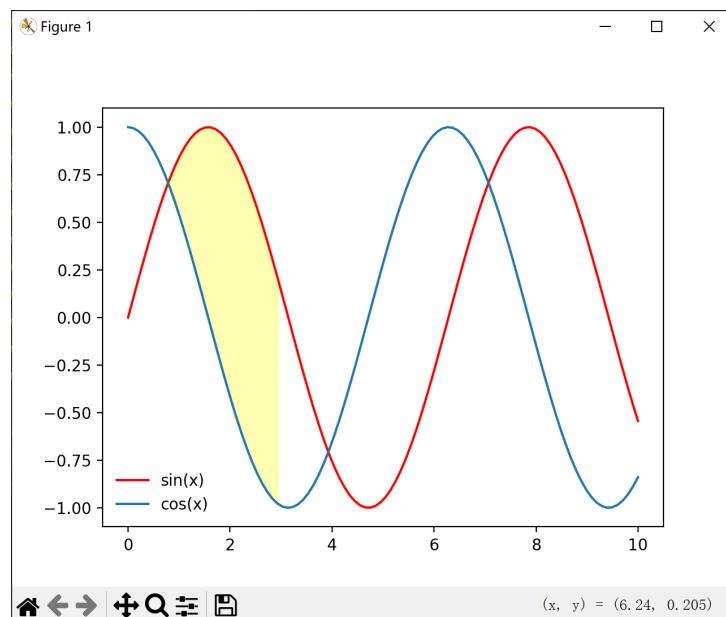


图 104: Enter Caption

相反，我们也可以删去 where，得到下图：

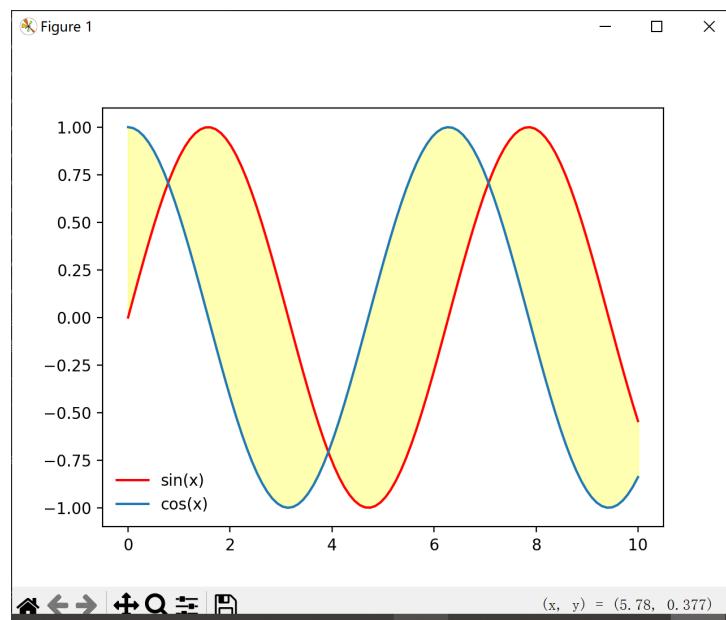


图 105: xs.6.5.4

8.6.6 堆积面积图

```
import matplotlib.pyplot as plt
import numpy as np
# 数据
labels = ['G1', 'G2', 'G3', 'G4']
data = np.array([[20, 34, 30, 35], [25, 32, 34, 20], [21, 31, 37, 21], [26, 31, 35, 27]])
x = np.arange(len(labels))
# 绘制堆积面积图
plt.stackplot(x, *args: data, labels=labels)
plt.legend()
plt.show()
```

图 106: xs.6.6.1

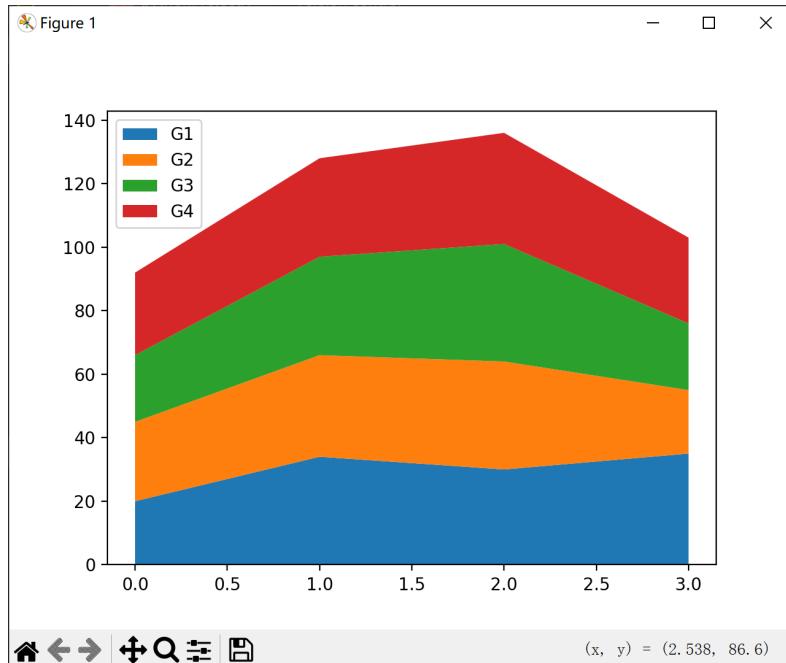


图 107: xs.6.6.2

通过这个例子不难看出，我们首先向 `stackplot` 传递一个关于 x 轴的数组，决定了横轴的取点（实际上这里 x 可以是列表甚至是元组，但是这样你取不了太多点），接着传递的 data 是一个长度与 x 相同的数组（其实它也可以是列表），其中的每个元素都是列表（也可以是数组）（[看到了吧，很多时候列表，数组就是可以随便混着玩](#)），而这个列表里包含的元素数决定了堆积图的层数，大小决定了每层堆多厚。你可以仔细观察一下是不是这样？

在程序中，我们给每层都带上了 label，最后形成图例，其实我个人认为图例并不总是必须的，毕竟我目前没发现堆积图有什么实际用途，除了好看。

说到好看，请读者思考一下如何运用堆积图生成一段好看的彩虹？（图例就真不必了）

```
> import matplotlib.pyplot as plt
> import numpy as np
> x=np.linspace(-6, stop: 6, num: 100)
> y=18-0.5*x**2
> y_list=[]
> for i in range(8):
>     y_list.append(list(y+i*0.5))
> y_array=np.array(y_list)
> plt.stackplot(x, *args: y_array, colors=['w', 'purple', 'b', 'c', 'g', 'y', 'orange', 'r'])
# plt.legend()
plt.title('Rainbow')
plt.show()
```

图 108: xs.6.6.3

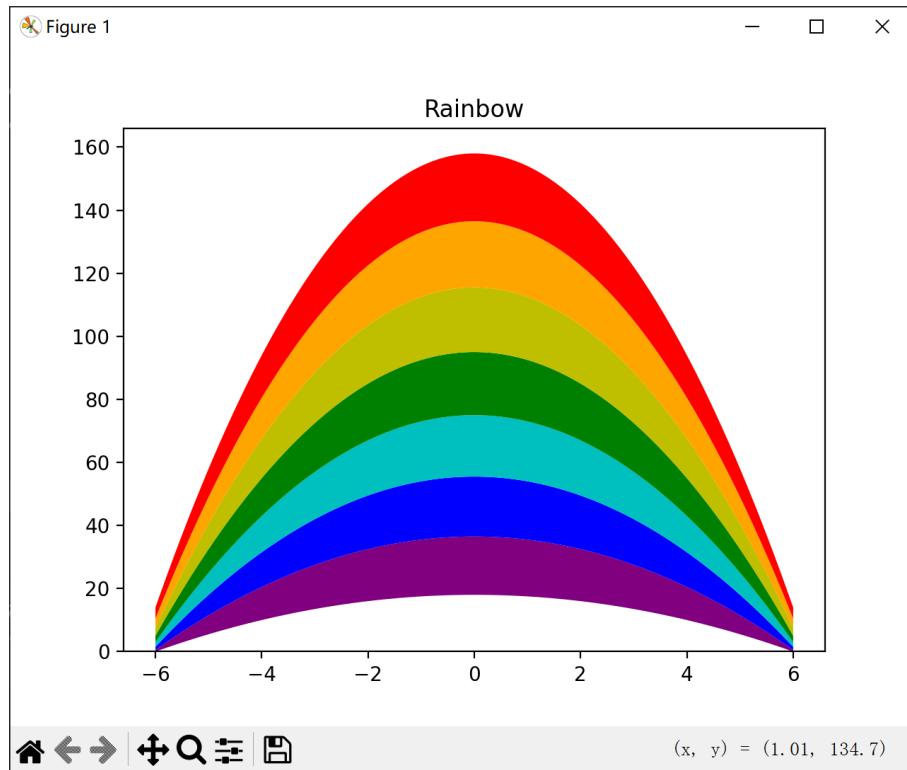


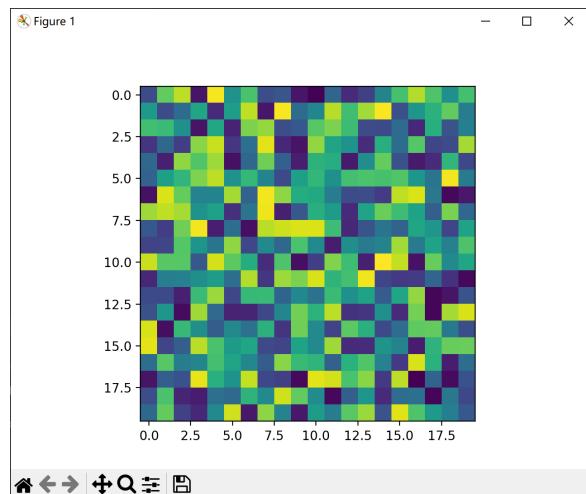
图 109: xs.6.6.4

其实颜色可以直接用 matplotlib 里面的颜色映射 cmap，但是对于这种对象比较少的情境本人还是喜欢手动指定颜色。

8.6.7 imshow() 函数

这是一个特殊的 pyplot 绘图函数，看它的名字似乎与 plt.show() 函数有些联系，实际上二者功能完全不同，plt.show() 的功能是将图形显示在屏幕上，是 Matplotlib 的最终渲染步骤，而 imshow() 并不能渲染图像，它的作用是绘制（在这之前它会创建一个图形对象，但这不代表它能代替 Figure () 函数，它不能单独作为创建图形对象的工具用，所以这个功能基本可以忽略）。

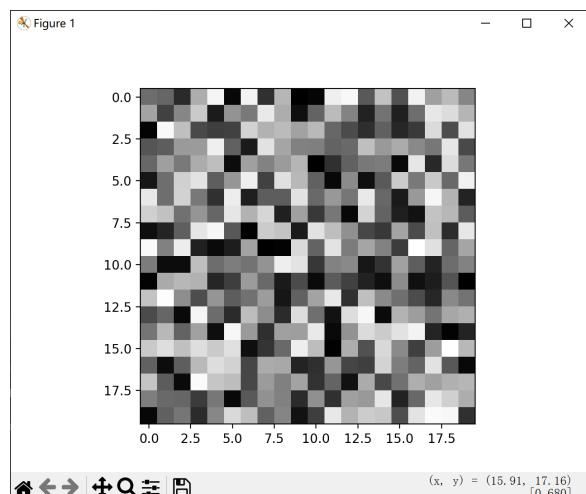
imshow() 的绘制可以概括为“二维数组及三维数组的可视化”，当它接受一个二维数组时，它可以绘制它的灰度图或色块图。此时 imshow() 接受两个参数，第一个是二维数组，第二个是颜色映射类型，**默认是 viridis**，我们也可以指定其它颜色映射类型，如果指定 cmap='grey'，绘制出来的就是灰度图。（最后一行 print 不重要）



(a) xs.6.7.1 默认 ‘viridis’

```
import matplotlib.pyplot as plt
import numpy as np
data = np.random.rand(20,20)
fig2,ax2=plt.subplots()
ax2.imshow(data)
plt.show()
print(data)
```

(b) xs.6.7.2 默认 ‘viridis’



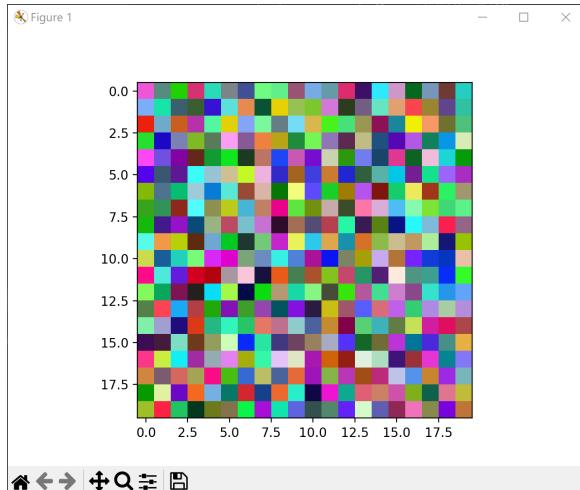
(a) xs.6.7.3 灰度图

```
import matplotlib.pyplot as plt
import numpy as np
data = np.random.rand(20,20)
fig2,ax2=plt.subplots()
ax2.imshow(data,cmap='grey')
plt.show()
print(data)
```

(b) xs.6.7.4 灰度图

当它接受一个三维数组时，第三个维度的数据就成为了颜色映射的依据，而不需要再设定 cmap 映射的类型。注意，可不是任意一个单位数组都能拿来用 imshow 可视化

的，首先第三个维度的长度为 3 或 4，其次其取值必须都在 [0, 1]，因为第三个维度作为颜色维度，它是遵循 RGB 或者 RGBA 来表示颜色的，我们知道 RGB 是 3 个 [0, 1] 范围的值组成的元组，RGBA 就是加上一个 alpha 透明度。因此，**第三个维度的加上实质是将原来二维数组的每个元素（数值）扩展成了一个包含三个或四个元素的列表**，用来指定这个点的颜色，原本在该二维平面上代表每个点的数值只有一个，因此必须借助一套映射规则来确定这个数值代表的颜色；现在代表每个点的数值有 3 个甚至 4 个，因此可以使用 RGB 或者 RGBA 直接确定颜色。



(a) xs.6.7.5

```
import matplotlib.pyplot as plt
import numpy as np
data = np.random.rand(20,20,3)
fig2,ax2=plt.subplots()
ax2.imshow(data)
plt.show()
print(data)
```

(b) xs.6.7.6

By the way, `np.random.rand` 的作用是随机生成指定个 [0,1] 范围内的值组成的数组，维度就是传递的参数个数，比如 `np.random.rand (4, 4, 3)` 就是生成一个三维数组，前两个维度长度为 4，第三个维度长度为 3，所有元素都在 [0,1] 范围内：

```
[[[0.63266539 0.50823386 0.52930336]
 [0.01332785 0.98283563 0.65469559]
 [0.65323118 0.36066021 0.25827955]
 [0.14107622 0.25872315 0.69132002]]

 [[0.84132864 0.20340719 0.29201145]
 [0.26392285 0.91025209 0.7071805 ]
 [0.17779751 0.3477427 0.52669649]
 [0.45224341 0.43521973 0.30448008]]

 [[0.08845291 0.03858424 0.28258737]
 [0.36498679 0.16931326 0.20487695]
 [0.17056632 0.64502098 0.12498202]
 [0.93411936 0.68720714 0.60767124]]

 [[0.25420004 0.2245403 0.17799378]
 [0.41635985 0.26097162 0.52997111]
 [0.87225997 0.73954795 0.38398717]
 [0.27223947 0.37083772 0.77577136]]]
```

图 113: xs.6.7.7

8.7 颜色映射

8.7.1 颜色映射的选取

颜色映射本质上是一个函数，它将一个标量值（通常是归一化后的数据值）映射到一个颜色。Matplotlib 提供了多种预定义的颜色映射，这些颜色映射可以根据数据的性质（如连续性、离散性、是否包含负值等）进行选择。

连续颜色映射	viridis	从深蓝到浅黄的渐变	适用于连续数据， 颜色从浅到深或从 一种颜色过渡到另一种颜色
	plasma	从深红到浅黄	
	inferno	从深黑到亮黄	
	magma	从深红到浅黄	
	cividis	从深蓝到浅黄	
散度颜色映射	coolwarm	从蓝色到红色，中间为白色	适用于数据具有中 心值（如零点）的场景， 颜色从中间值向两端渐变
	bwr	从蓝色到白色再到红色	
	Spectral	从蓝色到绿色再到黄色和红色	
循环颜色映射	twilight	从紫色到蓝色再到红色，最后回到紫色	适用于周期性数据， 颜色在起点和终点处平滑
	hsv	基于 HSV 颜色空间的循环颜色映射	
离散颜色映射	tab10	包含 10 种颜色的离散颜色映射	适用于离散数据， 颜色之间没有渐变关系， 每种颜色代表一个类别
	tab20	包含 20 种颜色的离散颜色映射	
	Accent	包含 8 种颜色的离散颜色映射	
杂项颜色映射	flag	颜色块交替	特殊用途
	prism	彩色条纹	
	ocean	从浅蓝到深蓝的渐变	

颜色映射的选取可以参考以下标准：

1. 连续数据：优先选择连续颜色映射。
2. 包含正负值的数据：优先选择散度颜色映射。
3. 分类数据：优先选择离散颜色映射。
4. 色盲友好：优先选择色盲友好的颜色映射（如 viridis 、 cividis ）

以上只是一些常用的的颜色映射，事实上 matplotlib 的颜色映射远不止这些，要想知道所有的颜色映射有两种做法：一是利用如下代码：

```
import matplotlib.pyplot as plt
colormaps=plt.colormaps()
print(colormaps)
```

图 114: xs.8.7.1

短短三行代码，却可以将所有颜色映射装在一个列表里打印出来。第二个方法是随便在一个支持颜色映射的函数中使用 cmap，并故意将这个映射的名字拼错，然后 Python 就会在报错中将所有正确的映射名称告诉你。（I am serious.）

8.7.2 哪些地方可以 cmap

并非所有绘图函数都支持颜色映射 cmap, 支持的有: plt.imshow()、plt.contourf()、plt.scatter()、plt.plot_surface() 这些是我们已经认识的函数, 还有一些比较冷僻, 若感兴趣请自行了解: plt.pcolormesh() (用于绘制伪彩色图)、plt.tripcolor() (用于绘制三角网格上的颜色图)、plt.hexbin() (用于绘制六边形的二维直方图)、plt.streamplot() (用于绘制流线图)、plt.quiver() (用于绘制箭头图)。

8.7.3 norm 的作用

之前我们说我们可以用 Normalize 函数设置一个 norm, 然后赋值给绘图函数中的 norm, 从而使数值线性映射到 cmap, 这里必须像我的读者道个歉, 那里主要是为了介绍 Normalize 这个有用的归一化函数, 其实如果我们希望数值线性映射给 cmap, 是不需要另外设 norm 的, 因为 cmap 的默认映射模式就是线性映射。设置 norm 通常是因为数值的波动较大, 或范围跨度很大, 使得线性映射并不能很好的实现可视化, 从而需要设置其它映射模式。(我个人称其为映射模式, 因为比较形象, 其实专业点叫数据规范化, Normalize 代表的是最简单的线性规范化)

规范化方法还有:

两斜率规范化: 适用于数据有正有负, 且中心值 (如零) 有特殊意义的情况。这种规范化的实质是如下公式:

$$\text{normalized_value} = \begin{cases} \frac{x - \text{vcenter}}{\text{vmin} - \text{vcenter}} & \text{if } x < \text{vcenter} \\ \frac{x - \text{vcenter}}{\text{vmax} - \text{vcenter}} & \text{if } x \geq \text{vcenter} \end{cases}$$

图 115: 两斜率规范化公式

其中 vmin 是数据的最小值, vmax 是数据的最大值, vcenter 是一个中心值, 通常选择为数据集的中位数或平均值, 正是 vcenter 选取的灵活性使得这种方法具有其它归一化方法所不具有的灵活性, 可以根据数据的分布特性作相应的调整。

(其实动手你就会发现, 当 vcenter=vmin 时, 两斜率规范化就成了线形规范化)

方法就是首先导入:

```
from matplotlib.colors import TwoSlopeNorm
```

然后设置一个 norm:

```
norm = TwoSlopeNorm(vcenter= , vmin=data.min(), vmax=data.max())
```

我们之前讲过，这个方法的关键是 vcenter 的选取，因此它是一个必要的参数，vmax 和 vmin 如果不用额外限制映射范围的话，没必要管它。

边界规范化：适用于离散数据，可以自定义颜色边界。

老样子，先导入：

```
from matplotlib.colors import BoundaryNorm
```

然后设置一个 norm：

```
norm = BoundaryNorm(boundaries, ncolors = )
```

boundries 是一个单调递增的数组，定义了颜色映射的边界。它的最小值和最大值分别是数据组的修行者和最大值；换句话说 boundries 相对于是对原数据组数据范围的一个分割，比如数据范围是 [-1,1]，那么它的 boundries 就可以是 [-1, -0.5, 0, 0.5, 1]，或者，我们直接用 linspace 函数对数据范围进行等分，但是这个划分并不一定得是等间距的，一切依据实际数据的特性而定。那么这样划分是为了什么呢？是为了确定颜色映射的分段。还是 [-1, -0.5, 0, 0.5, 1] 这个例子，它有 5 个数值，我们称之为 5 个边界，这 5 个边界确定的是 4 个区间，这个区间数就是颜色映射的分段数，cmap 会被分成相应段，然后依次映射到这些区间。

cmap 可以有两种设法，一种是人为创建一个颜色列表，比如我令 `cmap = ListedColormap(['blue', 'green', 'yellow', 'red'])`，4 种颜色，刚好等于区间数，这样这四种颜色就会依次映射到那四个区间。还有一种就是用 matplotlib 自带的颜色映射，你可能会说，那些 matplotlib 自带的颜色映射如 viridis，一般都是有 256 种颜色，而这些颜色是连续的，这时数据区间数与颜色数不等，但其实不用担心，因为 Matplotlib 会自动处理这种不匹配，根据 boundaries 的数量从 cmap 中采样颜色。

ncolors 也是一个必要的参数，ncolors 的值通常等于 boundaries 的数量减 1；但是有时会出现数据范围过小，导致全部是一种颜色、区分不开的情况。这个时候可以适当上调 ncolors 的值，注意 ncolors 的值不能低于区间数，一般调高一些能够使得颜色变化更多，但是太高了又会变回单一颜色，总之需要一定的尝试。

对数规范化：适用于数据范围跨度较大的情况，将数据取对数后进行规范化。用法是先导入 LogNorm：然后设置 `norm=LogNorm()`，在绘图函数中加上 `norm=norm` 就行了。

LogNorm 没有必要的设置。3.30 以上版本的 matplotlib 支持在 `LogNorm()` 中设置底数 `base=`。除此之外可以在 `LogNorm` 中设置起始与终止点 `vmin` 与 `vmax`，但其实不设也行。

对称对数规范化：适用于数据范围跨度较大且包含零值的情况。其实包含零值倒是其次，

个人认为其相对于对数规范化最大的优势在于设置更多，可操控的地方更多，但也更加复杂。方法与 LogNorm 基本一致，先导入：

```
from matplotlib.colors import SymLogNorm
```

然后令 `norm=SymLogNorm()`，`SymLogNorm` 的参数如下：

```
norm = SymLogNorm(linthresh = , linscale = , vmin = , vmax = , base = )
```

注意 `SymLogNorm` 有必要的设置，就是第一个参数 `linthresh`。`vmin` 和 `vmax` 就是起始与终止点，这个很好理解，而且设不设无所谓。`base` 之前说过，默认是 10。下面重点介绍前两个参数，它们比较难理解。

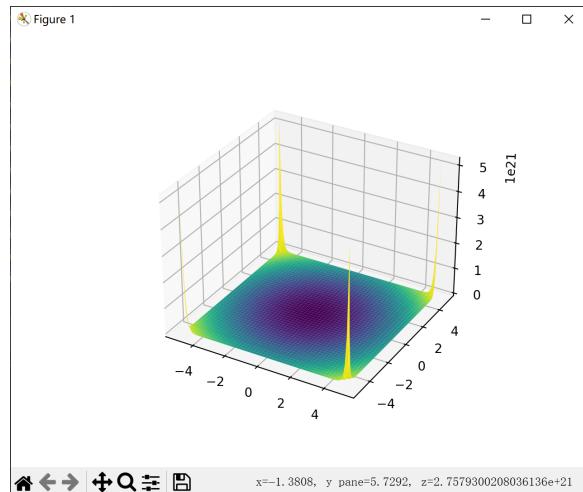
linthresh

`linthresh` 是线性范围的阈值。落在 $[-\text{linthresh}, \text{linthresh}]$ 范围内的输出值（因变量值）采用线性映射，从而避免对数刻度在零附近的无穷大问题。

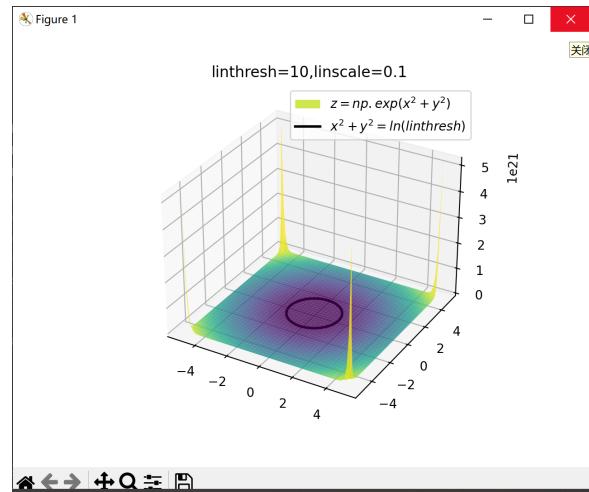
linscale

`linscale` 是线性范围的缩放因子。它控制线性范围 $[-\text{linthresh}, \text{linthresh}]$ 相对于对数范围的相对大小。具体来说，`linscale` 的值决定了线性范围在颜色映射中所占的比例。`linscale` 的范围是 $(0, \infty)$ ，默认是 1，这个比例很难说，目前看最好的办法是以底数的次方依次尝试，比如如果底数为 10，那就依次试 0.1, 1, 10, 100 看效果。

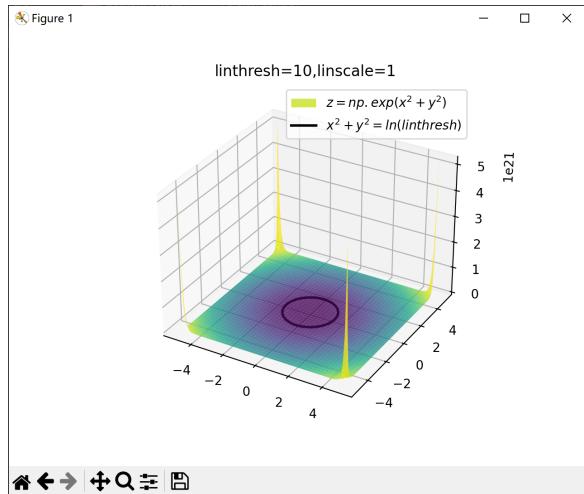
这个比例越高，意味着整个颜色条中更多的颜色集中在线性映射范围内，范围外的颜色就会比较少比较单调，下面一组图可以很形象的反映这个趋势：



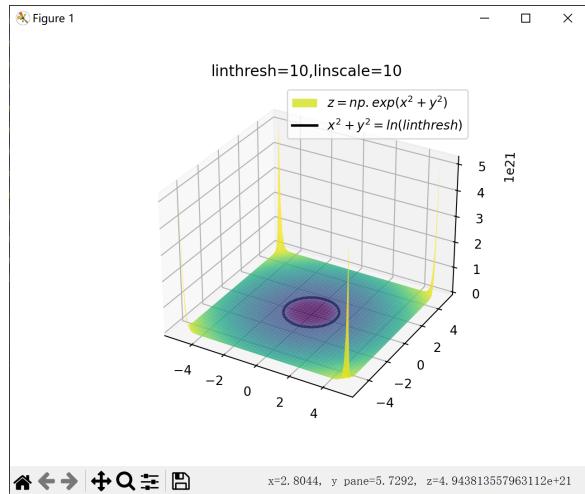
(a) xs.7.3.1 Just LogNorm



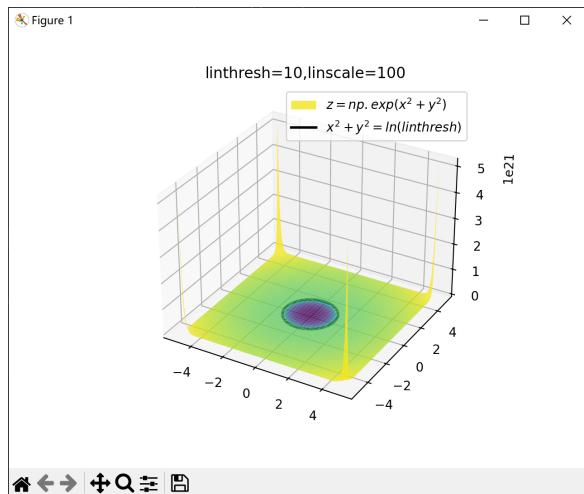
(b) xs.7.3.2 SymLogNorm Try 1



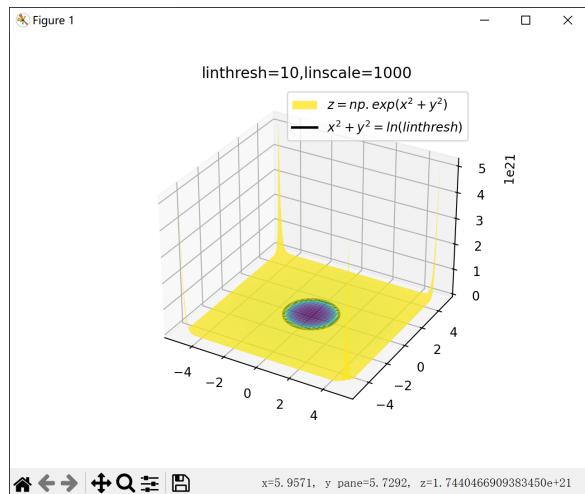
(a) xs.7.3.3 SymLogNorm Try 2



(b) xs.7.3.4 SymLogNorm Try 3



(a) xs.7.3.5 SymLogNorm Try 4



(b) xs.7.3.6 SymLogNorm Try 5

以上实验过程所使用的代码如下：

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib.colors import LogNorm
4 from matplotlib.colors import SymLogNorm
5 linthresh=10
6 linscale=1000 # 方便修改
7 ax=plt.subplot(projection='3d')
8 x=np.linspace(-5, stop: 5, num: 100)
9 y=np.linspace(-5, stop: 5, num: 100)
10 x,y=np.meshgrid(*xi: x, y)
11 z=np.exp(x**2+y**2) # 定义函数
12 norm1=LogNorm()
13 norm2=SymLogNorm(linthresh=10, linscale=linscale, base=np.e) # 定义两个norm
14 ax.plot_surface(x, y, z, cmap='viridis', norm=norm2, alpha=0.8,
15                  label='$z=np.exp(x^2+y^2)$') # 主要图像
16 theta=np.linspace(start: 0, 2*np.pi, num: 100)
17 r=np.sqrt(np.log(linthresh))
18 plt.plot(*args: r*np.sin(theta), r*np.cos(theta), c='k', lw=2,
19           label='$x^2+y^2=\ln(\lnthresh)$') # 黑圈标明阈值范围
20 plt.title(f'linthresh=10, linscale={linscale}')
21 plt.legend() # 标题与图例
22 plt.show()

```

图 119: xs.7.3.7

8.8 projection 参数：投影类型的指定

projection 参数用于指定坐标轴的投影类型，说白了就是你想建一个什么坐标系，可以通过这个设置参数实现。这个参数可以在 plt.subplot(), plt.subplots(), fig.add_subplot(), fig.add_axes() 这四个创建子图的函数中使用，用于指定子图坐标轴的类型，如果你已经忘记了这四个函数的用法，请 [click here:4.1](#)。

projection 参数可以接受以下这些投影类型：

1. 'aitoff': Aitoff 投影，是一种等面积投影，用于将球面数据投影到平面上，常用于天文学等领域。
2. 'hammer': Hammer 投影，也是一种等面积投影，与 Aitoff 投影类似，但形状略有不同，同样适用于球面数据的投影。
3. 'lambert': Lambert 投影，包括 Lambert 等角切圆锥投影和 Lambert 等面积投影等，用于将地球表面的地理坐标投影到平面上，常用于地图绘制。
4. 'mollweide': Mollweide 投影，是一种等面积的伪圆柱形投影，用于将球面数据投影到平面上，能够较好地保持面积关系，适用于全球地图的绘制。
5. 'polar': 极坐标投影，将笛卡尔坐标系中的数据转换为极坐标系中的数据，常用于绘制极坐标图，如雷达图等。
6. 'rectilinear': 默认的笛卡尔坐标系投影，即普通的二维平面坐标系。
7. '3d': 三维投影，用于绘制三维图形，如三维散点图、三维曲面图等。

绝大多数 pyplot 方法都是默认平面直角坐标系，但也有少数支持其他投影方式，比如之前讲过的 plt.polar() 就可以绘制极坐标系图，这些方法无疑为切换投影类型提供了快捷方式，但是要更灵活地为每个子图设置不同的投影类型，还是需要运用 Axes 方法和 projection 参数设置，这也是 Axes 方法灵活性的体现之一。

8.9 颜色的另类定义

8.9.1 RGB 模式

我们还可以通过 RGB 模式自定义颜色。我们向 color 或传递一个包含 3 个 0~1 之间小数的元组，如 $c = (0, 0, 0.8)$ ，这三个数分别代表红色，绿色和蓝色的分量，越接近 0，颜色越深；反之，越接近 1 颜色越浅，这个例子产生的就是淡蓝色。

8.9.2 RGBA 模式

没什么好讲的，RGBA 模式就是再 RGB 模式的基础上加了一个 alpha 透明度，是一个四元素元组。

8.9.3 十六进制模式

matplotlib 也支持用十六进制定义颜色，可参考下表：

颜色名称	十六进制值	颜色示例
蓝色 (Blue)	#1f77b4	
橙色 (Orange)	#ff7f0e	
绿色 (Green)	#2ca02c	
红色 (Red)	#d62728	
紫色 (Purple)	#9467bd	
棕色 (Brown)	#8c564b	
粉色 (Pink)	#e377c2	
灰色 (Gray)	#7f7f7f	
黄色 (Yellow)	#bcbd22	
青色 (Cyan)	#17becf	
深蓝色 (Navy)	#000080	
浅蓝色 (Sky Blue)	#87CEEB	
深绿色 (Forest Green)	#228B22	
浅绿色 (Lime Green)	#32CD32	
深红色 (Crimson)	#DC143C	
浅红色 (Salmon)	#FA8072	
深紫色 (Indigo)	#4B0082	
浅紫色 (Lavender)	#E6E6FA	
深黄色 (Gold)	#FFD700	
浅黄色 (Light Yellow)	#FFFFE0	
深青色 (Teal)	#008080	
浅青色 (Aqua)	#00FFFF	

表 3: Matplotlib 颜色十六进制定义表

顺带一提，使用 matplotlib.colors 模块中的 to_hex 函数可以将 RGB 值转换为十六进制：例如：

```
1 from matplotlib.colors import to_hex
```

```
2  
3 # 将 RGB 值转换为十六进制  
4 hex_color = to_hex((0.5, 0.3, 0.8)) # RGB 值范围为 0 到 1  
5 print(hex_color) # 输出: #7f4ccc
```

我不知道是不是多少精度的 RGB 值能找到一个对应的十六进制编码，感兴趣的读者可以自行尝试。

8.10 isinstance() 函数

isinstance() 是 Python (不是 matplotlib 哟) 中的一个内置函数，用于检查一个对象是否是某个类（或其子类）的实例。如果对象是该类的实例，函数返回 True，否则返回 False。其语法如下：

```
isinstance ( object , classinfo )
```

其中 object 是需要检测的对象，classinfo 是判断的类，简单说就是判断 object 是不是 classinfo 类的。

因为它是 Python 的内置函数，它的应用不只是在判断 matplotlib 中的类上，比如判断数据类型，print(isinstance([1, 2, 3], list))，其输出就是 True。有时候 classinfo 不是一个类，而是多个类组成的元组，这时候 isinstance 检查的就是对象是否属于元组里面多个类中的任意一个，比如 print(isinstance(123, (int, float, str))) 的输出是 True。

在 Matplotlib 的事件处理中，isinstance() 常用于区分不同类型的 Artist 对象。例如：

```
1 import matplotlib.pyplot as plt
2 from matplotlib.collections import PathCollection
3 usage
4 def onpick(event):
5     artist = event.artist
6     if isinstance(artist, plt.Line2D):
7         print("You picked a line")
8     elif isinstance(artist, PathCollection):
9         print("You picked a scatter point")
10    elif isinstance(artist, plt.Rectangle):
11        print("You picked a rectangle")
12    else:
13        print("You picked an unknown artist")
14
15 fig, ax = plt.subplots()
16 line, = ax.plot(*args: [0, 1], [0, 1], picker=5)
17 scatter = ax.scatter(x: [0.4], y: [1], picker=5)
18 rect = plt.Rectangle(xy: (0.2, 0.2), width: 0.4, height: 0.4, picker=5)
19 ax.add_patch(rect)
20 fig.canvas.mpl_connect(s: 'pick_event', onpick)
21 plt.show()
```

图 120: xs.8.10.1

我们分别点击点，矩形和直线，会发现程序分别判断了它们的类型，并打印出相应的语句。同时不难发现虽然程序中有一句在 else 条件下 print("You picked an unknown artist")，但是这句话并不会被打印出来，因为整个画布上能发生交互的 artist 只有那三个，别的地方你点了也没用，所以根本就没有 unknown artist。

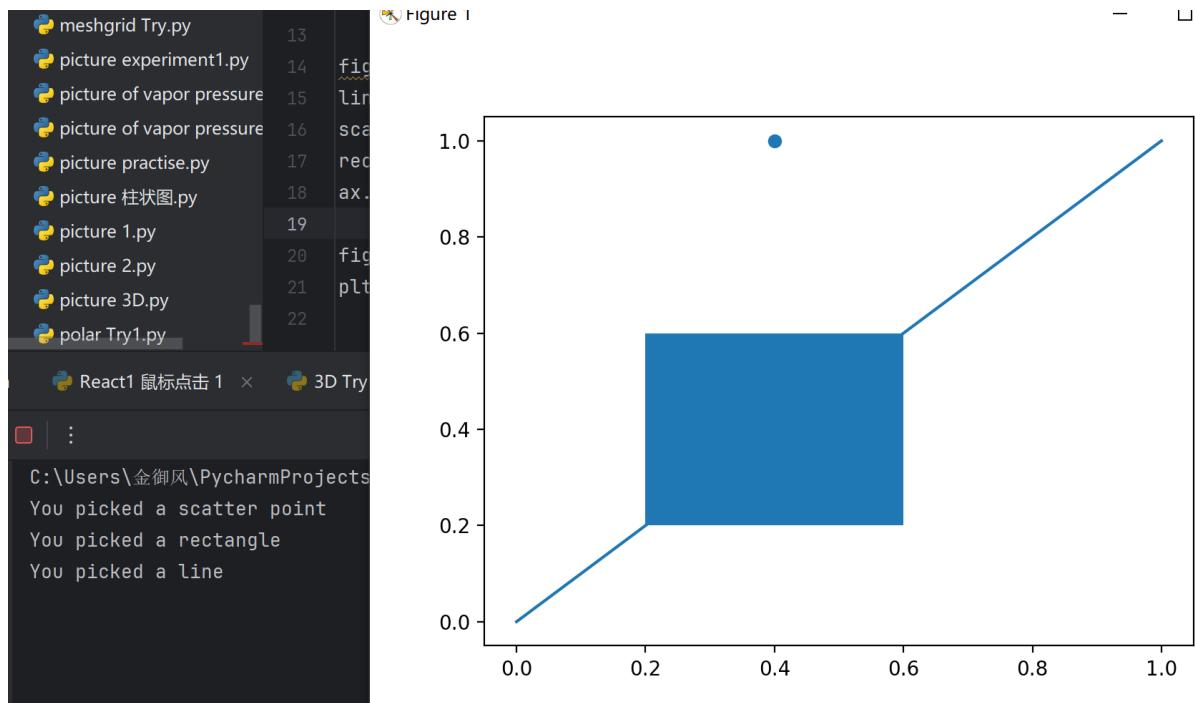


图 121: xs.8.10.2