目录

Introduction 1.1

第一部分: Python从0到1 1.2

第一章: Python基础知识 1.2.1

1.1 Python基础 1.2.1.1

1.1.1 Hello, Python 1.2.1.1.1

1.1.2 数据类型与变量 1.2.1.1.2

1.1.3 数据结构 1.2.1.1.3

1.1.4 分支与循环 1.2.1.1.4

1.2 Python进阶 1.2.1.2

1.2.1 函数 1.2.1.2.1

1.2.2 模块 1.2.1.2.2

1.2.3 类与对象 1.2.1.2.3

1.2.4 文件IO 1.2.1.2.4

1.2.5 异常处理 1.2.1.2.5

1.2.6 正则表达式 1.2.1.2.6

第二章：数值计算利器---NumPy 1.2.2

2.1: NumPy基础 1.2.2.1

2.1.1 ndarray对象 1.2.2.1.1

2.1.2 形状操作 1.2.2.1.2

2.1.3 基础操作 1.2.2.1.3

2.1.4 随机数生成 1.2.2.1.4

2.1.5 索引与切片 1.2.2.1.5

2.2: NumPy进阶 1.2.2.2

2.2.1 堆叠操作 1.2.2.2.1

2.2.2 花式索引与布尔索引 1.2.2.2.2

2.2.3 广播机制 1.2.2.2.3

2.2.4 线性代数 1.2.2.2.4

2.2.5 排序和条件筛选 1.2.2.2.5

2.2.6 结构化数组 1.2.2.2.6

第三章：结构化数据大杀器---Pandas 1.2.3

3.1 Pandas基础 1.2.3.1

3.1.1 Series对象 1.2.3.1.1

3.1.2 DataFrame对象 1.2.3.1.2

3.1.3 Index对象 1.2.3.1.3

3.1.4 Series对象数据选择 1.2.3.1.4

3.1.5 DataFrame数据选择方法 1.2.3.1.5
3.1.6 数值运算方法 1.2.3.1.6
3.1.7 数值运算与缺失值处理 1.2.3.1.7
3.2 Pandas进阶 1.2.3.2
3.2.1 多级索引的取值与切片 1.2.3.2.1
3.2.2 多级索引的数据转换与累计方法 1.2.3.2.2
3.2.3 Concat与Append操作 1.2.3.2.3
3.2.4 合并与连接 1.2.3.2.4
3.2.5 分组聚合 1.2.3.2.5
3.2.6 创建透视表和交叉表 1.2.3.2.6
3.2.7 字符串操作方法 1.2.3.2.7
3.2.8 日期与时间工具 1.2.3.2.8
3.2.9 时间序列的高级应用 1.2.3.2.9
第四章：可视化工具---Matplotlib 1.2.4
4.1 Matplotlib基础 1.2.4.1
4.1.1 画图接口 1.2.4.1.1
4.1.2 线形图 1.2.4.1.2
4.1.3 散点图 1.2.4.1.3
4.1.4 直方图 1.2.4.1.4
4.1.5 饼图 1.2.4.1.5
4.1.6 手动创建子图 1.2.4.1.6
4.1.7 网格子图 1.2.4.1.7
4.1.8 更复杂的排列方式 1.2.4.1.8
4.2 Matplotlib进阶 1.2.4.2
4.2.1 配置颜色条 1.2.4.2.1
4.2.2 设置注释 1.2.4.2.2
4.2.3 自定义坐标刻度 1.2.4.2.3
4.2.4 配置文件与样式表 1.2.4.2.4
4.2.5 绘制三维图 1.2.4.2.5
4.2.6 曲面三角剖分 1.2.4.2.6
第二部分: 机器学习与综合实战 1.3
第五章: 机器学习 1.3.1
5.1 机器学习概述 1.3.1.1
5.1.1 机器学习概述 1.3.1.1.1
5.1.2 机器学习的主要内容 1.3.1.1.2
5.1.3 怎样评估模型性能 1.3.1.1.3
5.2 监督学习 1.3.1.2
5.2.1 近朱者赤近墨者黑---kNN 1.3.1.2.1
5.2.2 最简单的回归算法---线性回归 1.3.1.2.2

5.2.3 别被我的名字蒙蔽了---逻辑回归 1.3.1.2.3
5.2.4 用概率说话---朴素贝叶斯分类器 1.3.1.2.4
5.2.5 最接近人类思维的算法---决策树 1.3.1.2.5
5.2.6 好还不够，我要最好---支持向量机 1.3.1.2.6
5.2.7 群众的力量是伟大的---随机森林 1.3.1.2.7
5.2.8 知错能改善莫大焉---Adaboost 1.3.1.2.8
5.3 无监督学习 1.3.1.3
5.3.1 物以类聚人以群分---k-Means 1.3.1.3.1
5.3.2 用密度来聚类---DBSCAN 1.3.1.3.2
5.3.3 分久必合---AGNES 1.3.1.3.3
5.3.4 最重要的才是我想要的---PCA 1.3.1.3.4
5.3.5 不忘初心---多维缩放 1.3.1.3.5
第六章: 综合实战 1.3.2
6.1 泰坦尼克生还预测 1.3.2.1
6.1.1 简介 1.3.2.1.1
6.1.2 探索性数据分析 1.3.2.1.2
6.1.3 特征工程 1.3.2.1.3
6.1.4 构建模型进行预测 1.3.2.1.4
6.1.5 调参 1.3.2.1.5
6.2 人脸性别识别 1.3.2.2
6.2 人脸性别识别 1.3.2.2.1
6.2.1 OpenCV入门 1.3.2.2.2
6.2.2 人脸位置检测 1.3.2.2.3
6.2.3 人脸性别识别 1.3.2.2.4

Introduction

第一部分: Python从0到1

Python凭借着其简单的语法、丰富的第三方模块使其成为了各种领域中最受欢迎的语言之一。在这一部分中，将会从Python的安装开始，一步一步地引导读者学会Python的语法知识，以及在学习机器学习之前需要掌握的一些数据科学相关的第三方模块。准备好了吗？Let's Go！

Chapter1 Python语言

由于 Python 语言的简洁性、易读性以及可扩展性，在国外用 Python 做科学计算的研究机构日益增多，一些知名大学已经采用 Python 来教授程序设计课程。例如卡耐基梅隆大学的编程基础、麻省理工学院的计算机科学及编程导论就使用 Python 语言讲授。众多开源的科学计算软件包都提供了 Python 的调用接口，例如著名的计算机视觉库 OpenCV 、三维可视化库 VTK 、医学图像处理库 ITK 。而 Python 专用的科学计算扩展库就更多了，例如如下3个十分经典的科学计算扩展库：NumPy、Pandas 和 Matplotlib ，它们分别为 Python 提供了快速数组处理、结构化数据处理以及绘图功能。因此 Python 语言及其众多的扩展库所构成的开发环境十分适合工程技术、科研人员处理实验数据、制作图表，甚至开发科学计算应用程序。

Python 语言相关实训已在 `educoder` 平台上提供，若感兴趣可以输入链接进行体验。

链接：https://www.educoder.net/paths/85

1.1.1 Hello，Python

Python为何物？

说出来您可能不信，现在大名鼎鼎的编程语言 Python 是由 Guido van Rossum 在 1989 年圣诞节期间，为了打发无聊的圣诞节而编写的一个编程语言。而也正是因为作者的无聊，我们现在才得以使用风靡全球的编程语言 Python 。(无聊是个好东西)

Python 为什么这么流行呢？主要原因有两个：

- 首先是因为本身语法简单，而且提供了非常完善的基础代码库，覆盖了如网络、文件、GUI、数据库、正则表达式等大量内容。您可别小看了这些基础代码库，它会让我们在实现功能时减少我们的代码量，将我们从底层代码中解放出来，更加的关心我们需要实现的业务功能。
- 其次是大家觉得 Python 这么好用，就会有很多具有开源精神的开发者去开发各种各样的第三方模块。接着大家可能会去尝试使用这些模块，然后发现这些模块挺好用的，然后就可能会把这些好用的模块或者 Python 语言推荐给自己的朋友、同事。就这样，不断地完善了 Python 的生态。以前用 C 语言想要实现一个功能可能需要 30 行代码，而用 Python 可能只需要 1 行代码。这也就使得编程的门槛越来越低，越来越流行。

事物都有两面性，既然 Python 有这么大的优点，那肯定也有不小的缺点。 Python 最为致命的确定就是运行速度慢。由于 Python 是解释型语言，您所编写的代码在执行时，解释器会一行一行的翻译成 CPU 能理解的机器码，这个翻译过程是非常耗时的，所以 Python 的运行速度很慢。而像 C 语言所编写的程序在运行前会直接将代码翻译成 CPU 能够理解的机器码，所以 C 语言编写的程序运行时非常快。

不过值得庆幸的是，大量的应用程序不需要这么快的运行速度，因为用户根本感觉不出来。例如开发一个下载 短视频 的网络应用程序，C 程序的运行时间需要0.001 秒，而 Python 程序的运行时间需要 0.1 秒，慢了 100 倍，但由于网络更慢，需要等待 5 秒。您可以思考一下，用户真的能感觉到 5.001 秒和 5.1秒 的区别吗？

这就好比布加迪威龙和五菱宏光在北京三环路上行驶的道理一样，虽然理论时速高达 430 公里，但由于三环路堵车的时速只有 20 公里，因此，作为乘客，您感觉的时速是 20 公里。

所以选择 Python 作为编程语言是一种比较明智的选额。

如何安装Python？

相信现在您可能已经迫不及待地想要在自己的机器上安装 Python 并开始 Python 之旅了。不过在安装 Python 前还需要了解一个知识，就是 Python 其实有两个不兼容的版本，一个是 2.x 版，另一个是 3.x 版。但由于 2.x 版只维护到 2020 年，而且 Python 官方都直接建议直接使用 3.x 版。所以在这里建议您安装最新的Python 3.7 版。

想要安装 Python 3.7 可以 Python 官网(https://www.python.org/downloads/)根据您机器的操作系统的类型下载对应的安装程序，然后运行安装即可。

喜闻乐见的**Hello，World**

一般在学编程时都先学着写一个 “Hello，World” 来表征一下自己的已经开始了编程之旅。那么我们来看一下怎样写 Python 版的 “Hello，World” 。(本书使用的操作系统是 Windows)

首先在自己喜欢的一个目录中创建一个文本文件，并在文件中输入如下代码：

```
print('Hello, World')
```

然后修改文件的名字，如 first.py。**(**注意：**Python** 代码文件的后缀名为 **py)**

接着进入命令行，进入到 first.py 所在的目录。例如 first.py 在 D:/code 目录下，所以需要在命令行输入：

```
cd d:/code
```

最后只要在命令行输入如下命令就可以运行我们写好的程序了。

```
python first.py
```

运行程序后看到的输出和我们的预期一致。

```
Hello, World
```

1.1.2 数据类型与变量

Python内置数据类型

如果让您解释一下什么是计算机的话，您可能会说计算机顾名思义就是可以做数学计算的机器。的确如此，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、视频等各种各样的数据，不同的数据，需要定义不同的数据类型。在 Python 中，能够直接处理的数据类型称为内置数据类型，内置数据类型有：整数、浮点数、字符串和布尔值。

整数

`Python` 可以处理任意大小的整数，当然包括负整数，在程序中的表示方法和数学上的写法一模一样，例如：`1，100，-8080，0` 等等。

由于计算机使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用 `0x` 前缀和 `0-9，a-f` 表示，例如：`0xff00，0xa5b4c3d2` 等等。

浮点数

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的，比如，`1.23x10^9` 和 `12.3x10^8` 是完全相等的。浮点数可以用数学写法，如 `1.23，3.14，-9.01` 等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把 `10` 用 `e` 替代，`1.23x10^9` 就是 `1.23e9`，或者 `12.3e8，0.000012` 可以写成 `1.2e-5` 等等。

字符串

字符串是以单引号 `'` 或双引号 `"` 括起来的任意文本，比如 `'abc'，"xyz"` 等等。请注意，`''` 或 `""` 本身只是一种表示方式，不是字符串的一部分，因此，字符串 `'abc'` 只有 `a`，`b`，`c` 这3个字符。如果 `'` 本身也是一个字符，那就可以用 `""` 括起来，比如 `"I'm OK"` 包含的字符是 `I`，`'`，`m`，`空格`，`O`，`K` 这6个字符。

如果字符串内部既包含 `'` 又包含 `"` 怎么办？可以用转义字符 `\` 来标识，比如：`'I\'m \"OK\"!'`

表示的字符串内容是：`I'm "OK"!`

转义字符 `\` 可以转义很多字符，比如 `\n` 表示换行，`\t` 表示制表符，字符 `\` 本身也要转义，所以 `\\` 表示的字符就是 `\`，可以在 Python 的交互式命令行用 `print()` 打印字符串看看：

```
>>> print('I\'m ok.')
I'm ok.
>>> print('I\'m learning\nPython.')
I'm learning
Python.
>>> print('\\\n\\')
\
\
>>> print('\tHello')
    Hello
```

如果字符串里面有很多字符都需要转义，就需要加很多 `\` ，为了简化， Python 还允许用 `r''` 表示 `''` 内部的字符串默认不转义，可以自己试试：

```
>>> print('\\\t\\')
\       \
>>> print(r'\\\t\\')
\\\t\\
```

部分转义字符：

转义字符	描述
\	(在行尾时) 续行符
\\	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格(Backspace)
\n	换行
\t	横向制表符

对于中文字符串的表示，为保证 Python 程序处理中文字符时不乱码，需要在 Python 程序最前面加上 `#coding = utf-8` ,例如：

```
>>> #coding=utf-8
>>> a = "你好世界！"
>>> print(a)
你好世界！
```

字符串运算符

字符串可以使用 `+` 号和 `*` 号进行运算。可以组合和复制，运算后会生成一个新的字符串。

Python 表达式	结果	描述
len("abcd")	4	计算元素个数
"123" + "456"	"123456"	连接
"Hi!" * 4	'Hi!Hi!Hi!Hi!'	复制
"3" in "123"	True	元素是否存在

字符串内置函数

字符串包含了以下内置函数

函数名	描述
len(list)	计算字符串长度
string.capitalize()	字母字符串首字母大写
string.split(str)	将字符串按 `str` 的形式分割

str(a)	将变量转换为字符串

```
#使用逗号运算符给多个变量赋值
t1,t2 = "1-2-3-4","abcd"
print(len(t1))
print(t2.capitalize())
print(t1.split("-"))
#创建一个整型变量a
a = 123
print(str(a))
```

以上实例输出结果：

```
7
Abcd
['1','2','3','4']
"123"
```

布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有 `True`、`False` 两种值，要么是 `True` ，要么是 `False` ，在Python中，可以直接用 `True` 、 `False` 表示布尔值（请注意大小写），也可以通过布尔运算计算出来：

```
>>> True
True
>>> False
False
>>> 3 > 2
True
>>> 3 > 5
False
```

布尔值可以用 `and` 、 `or` 和 `not` 运算。

`and` 运算是与运算，只有所有都为 `True` ， `and` 运算结果才是 `True` ：

```
>>> 5 > 3 and 3 > 1
True
```

`or` 运算是或运算，只要其中有一个为 `True` ， `or` 运算结果就是 `True` ：

```
>>> 5 > 3 or 1 > 3
True
```

`not` 运算是非运算，它是一个单目运算符，把 `True` 变成 `False` ， `False` 变成 `True` ：

```
>>> not 1 > 2
True
```

1.1.3 数据结构

数据结构：即人们抽象出来的描述现实世界实体的数学模型（非数值计算）及其上的操作（运算），在计算机上的表示和实现。

数据结构就是指按一定的逻辑结构组成的一批数据，使用某种存储结构将这批数据存储于计算机中，并在这些数据上定义了一个运算集合。

`Python` 中内置了 `4` 种常用的数据结构，分别为：列表，元组，集合，字典。

列表

什么是列表

列表是最常用的 `Python` 数据类型，它可以作为一个方括号内的逗号分隔值出现。列表的数据项不需要具有相同的类型创建一个列表，只要把逗号分隔的不同的数据项使用方括号括起来即可。

简单来说列表是由一系列元素按特定顺序排列组成。你可以创建包含字母表中所有字母、数字或一些字符串的列表；可以将其他数据类型放入列表中，甚至可以将另一个列表放在列表中。

在 `Python` 中，用方括号 `[]` 来表示列表，并用逗号来分隔其中的元素。例如：

```
>>>a = []   #创建一个空的列表
>>>b = [1,2,3,4] #创建一个含数字的大小为 4 的列表
>>>c = [1,'a',2,'b',3,'c',4,'d'] #创建包含多种数据类型的列表
>>>a = [b,c]  #创建包含其他列表的列表
>>>print(a)
[[1,2,3,4],[1,'a',2,'b',3,'c',4,'d']]
```

如何访问列表中的元素

先可以考虑前面所学的字符串类型的数据访问，例如有一字符串：`"abcd"` 。如果我们需要看它的某个特定位置上的字符是什么，则只要知道它的索引位置就行了，索引位置如下图：

字符串	a	b	c	d
索引 ->	0	1	2	3
索引 <-	0	-3	-2	-1

计算机存储数据的位置都是从 `0` 号位置开始存储的，习惯使用从左往右进行访问，特殊情况可从右往左访问，即从 `0` 号位置开始，到 `-1` 号位置在最右边（最后）往左（前）访问。要访问上面那个字符串 `"abcd"` 中的字符 `b` 的话，我们可以用这样一种形式：

```
"abcd"[1]     #或 "abcd"[-3]
```

字符 `b` 的索引位置在字符串 `"abcd"` 中为 `1` (或 `-3`)，所以可以通过加中括号 `"[]"` (中括号内为索引位置)的形式访问。 通常我们习惯将这个字符串赋值给一个变量然后通过变量名进行操作：

```
>>> a = "abcd"
>>> print(a[1])   #输出变量a储存的字符串索引位置为2的字符
b
```

与字符串的索引一样，列表索引从 `0` 开始。列表可以进行截取、组合等。使用索引位置来访问列表中的值，同样你也可以使用方括号的形式访问索引位置，如下所示：

```
>>> list = ['physics', 'chemistry', 1997, 2000]
>>> print(list[0])
physics
>>> print(['physics', 'chemistry', 1997, 2000][0])
physics
```

以上两种形式都可以访问列表 `list` 索引位置为 `0` 的数据 `physics`，第一个通过变量间接访问，第二个是直接访问，形式均为：

```
列表[索引位置]
```

可见当一个变量被赋值为某种数据类型时，该变量就相应变为了赋值的数据类型。例如：

```
>>> a = 10     #此时a的数据类型为整数类型
>>> a = ['physics', 'chemistry'] #此时变为列表
>>> a[1]      #对应 列表[索引位置] 的形式来访问特定位置
```

若要继续访问列表 `['physics', 'chemistry']` 中元素字符串 `physics` 的某个位置上的字符，可继续采用后面加 `[]` 的形式，例如：

```
>>> a = 10     #此时a的数据类型为整数类型
>>> a = ['physics', 'chemistry'] #此时变为列表
>>> print(a[0])    #打印第一个元素（这里为字符串）
physics
>>> print(a[0][1])    #打印第一个元素的第二个位置上的字符
h
```

注意 `[]` 内的索引数字大小必须小于要访问元素长度，例如： `a = ['physics', 'chemistry']` 以 `a[x]` 访问列表元素，则 `x` 要小于列表元素个数 `2` 并且要大于 `-3`，以 `a[x][y]` 访问列表元素（这里是字符串，单个数据元素不用此操作）内的元素时，则 `y` 要小于列表元素里的元素长度，例如列表 `0` 号元素 `physics` 长度为 `7`，则 `y` 小于 `7`，且大于 `-8` 依次类推。

列表的相关操作

列表的相关操作有以下几种：增加元素、删除元素、替换元素、列表运算符、列表内置函数

增加元素

通过使用 `append()` 函数来在列表末尾处添加列表元素:

```
>>> list = []          #创建空列表
>>> list.append('Google')   #使用append()添加元素
```

```
>>> list.append('Runoob')
>>> print(list)
['Google', 'Runoob']  #结果
```

添加的元素按 `append()` 函数从上到下的先后顺序在列表中从左至右的顺序存放。

删除元素

通过使用 `pop()` 函数来删除列表末尾处的列表元素：

```
>>> list = ['Google', 'Runoob']
>>> list.pop()
>>> print(list)
['Google']
```

这里需要注意的是 `pop()` 函数也可以通过指定索引位置来删除列表特定位置的数据，例如：

```
>>> list = ['Google', 'Runoob']
>>> list.pop(0)
>>> print(list)
['Runoob']
```

还可以使用 `remove()` 函数来删除指定的内容：

```
>>> list = ['Google', 'Runoob']
>>> list.remove('Google')
>>> print(list)
['Runoob']
```

两种方式都可以将列表中的元素删除，可在不同情形下使用。

替换元素

如果想要改变一个有数据的列表某个特定位置上的数据，我们可以通过类似赋值的方式进行：

```
>>> list = ['Google', 'Runoob']
>>> list[0] = "Baidu"
>>> print(list)
['Baidu', 'Runoob']
```

列表运算符

与字符串一样，列表之间可以使用 + 号和 * 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的列表。

Python 表达式	结果	描述
len([1, 2, 3])	3	计算元素个数
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	连接
['Hi!',] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	复制

3 in [1, 2, 3]	True	元素是否存在

列表内置函数

Python 列表包含了以下内置函数

函数名	描述
len(list)	计算列表元素个数。
max(list)	返回列表中元素最大值。
min(list)	返回列表中元素最小值。
list(str)	将字符串转换为列表。

```
#使用逗号运算符给多个变量赋值
t1,t2 = [10,30,50],[20,30,50]
print(len(t1))
print(min(t2))
print(max(t1))
#创建一个字符串a
a = "abcd"
print(list(a))
```

以上实例输出结果：

```
3
20
50
['a','b','c','d']
```

元组

Python 的元组与列表类似，不同之处在于元组的元素不能修改。元组使用小括号，列表使用方括号。元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

创建元组

```
tup1 = ()
```

元组中只包含一个元素时，需要在元素后面添加逗号，避免与括号运算符混淆。

```
tup1 = (50,)
```

与列表一样，元组的创建也可以嵌套进行。

```
tup = (1,2,3,"abc",(1,2,3,"abc"),[1,2,3,"abc"])
```

访问元组

与列表一样元组可以使用索引位置来访问元组中的值，如下实例:

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7 )

print("tup1[0]: ", tup1[0])
print("tup2[-1]: ",tup2[-1])
```

以上实例输出结果：

```
tup1[0]:  physics
tup2[-1]: 7
```

嵌套的元组访问元素与列表一样，通过再后面添加中括号的形式。

```
tup = (1,2,3,"abc",(10,20,30))
print("tup[3][1]: ",tup[3][1])
print("tup[4][0]: ",tup[4][0])
```

以上实例输出结果：

```
tup[3][1]: b
tup[4][0]: 10
```

修改元组

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合，如下实例:

```
tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')

# 以下修改元组元素操作是非法的。
# tup1[0] = 100

# 创建一个新的元组
tup3 = tup1 + tup2
print(tup3)
```

以上实例输出结果：

```
(12, 34.56, 'abc', 'xyz')
```

元组运算符

与列表一样，元组之间可以使用 `+` 号和 `*` 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的元组。

Python 表达式	结果	描述

len((1, 2, 3))	3	计算元素个数
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	连接
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	复制
3 in (1, 2, 3)	True	元素是否存在

元组内置函数

`Python` 元组包含了以下内置函数

函数名	描述
len(tuple)	计算元组元素个数。
max(tuple)	返回元组中元素最大值。
min(tuple)	返回元组中元素最小值。
tuple(list)	将列表转换为元组。

```
#使用逗号运算符给多个变量赋值
t1,t2 = (10,30,50),(20,30,50)
print(len(t1))
print(min(t2))
print(max(t1))
#创建一个列表a
a = [1,2,3]
print(tuple(a))
```

以上实例输出结果：

```
3
20
50
(1,2,3)
```

集合

集合（ `set` ）与数学意义相同，是一个无序的元素不重复的序列。

我们可以使用大括号 `{ }` 或者 `set()` 函数创建集合，注意：创建一个空集合必须用 `set()` 而不是 `{ }` ，因为 `{ }` 是用来创建一个空字典。

创建格式：

`parame = {value01,value02,...}`

或者：

`set(value)`

```
>>> a = {'apple', 'orange', 'pear'}
>>> a
{'apple', 'orange', 'pear'}
>>> s = set()   #使用set()创建一个空的集合
```

集合去重

重复元素在 `set` 中自动被过滤,即去重功能：

```
>>> fruit = {'apple', 'orange', 'apple', 'pear'}
>>> print(basket)
{'orange', 'pear', 'apple'}    #集合是无序的
>>> s = set([1, 1, 2, 2, 3, 3])   #对列表去重
>>> s
{1, 2, 3}
```

添加元素

通过 `add(key)` 方法可以添加元素到 `set` 中，可以重复添加，但不会有效果：

```
>>> s = {1,2,3,4}
>>> s
{1, 2, 3, 4}
>>> s.add(4)      #添加重复元素 4
>>> s
{1, 2, 3, 4}
```

删除元素

通过 `remove(key)` 方法可以删除元素：

```
>>> s.remove(4)     #删除元素 4而不是索引位置 4
>>> s
{1, 2, 3}
```

集合运算

两个 `set` 可以做数学意义上的交集、并集等运算：

```
>>> a = {1,2,3,4}
>>> b = {4,5,6,7}
>>> a - b           # 只在集合a中包含元素
{1,2,3}
>>> a | b           # 集合a或b中包含的所有元素
{1,2,3,4,5,6,7}
>>> a & b           # 集合a和b中都包含了的元素
{4}
>>> a ^ b           # 不同时包含于a和b的元素
{1,2,3,5,6,7}
```

常用操作

计算集合的大小可使用函数 `len()`

```
>>> thisset = {"Google", "Runoob", "Taobao"}
>>> len(thisset)
3
```

使用 `in` 判断元素是否在集合中：

```
>>>thisset = {"Google", "Runoob", "Taobao"}
>>> "Runoob" in thisset
True
>>> "Facebook" in thisset
False
```

字典

字典是 `Python` 最强大的数据类型之一，通过 `键-值(key-value)对` 的方式建立数据对象之间的映射关系。字典的每个键-值对用冒号 `:` 分割，每个 `键-值对` 间用逗号 `,` 分隔开，字典是包含在 `{}` 中。

每个键都与一个值相关联，我们可以使用键来访问与之相关联的值。与键相关联的值可以是数字、字符串、列表乃至字典。事实上，可将任何 `Python` 对象用作字典中的值。

创建字典

字典的创建格式如下所示：

`d = {key1 : value1, key2 : value2 }`

键必须是唯一的，但值则不必。

值可以取任何数据类型，但 `键` 必须是不可变的，如字符串，数字或元组。

一个简单的字典实例：

```
>>> dict = {'Alice': '2341', 'Beth': '9102'}
>>> print(dict)
{'Alice': '2341', 'Beth': '9102'}
```

访问字典中的值

要获取与 `键` 相关联的值，可依次指定字典名和放在方括号内的 `键` 。我们访问列表等其他类型是通过方括号 `[]` 内添加索引位置的形式，这里字典我们把索引位置用字典中的 `键(key)` 来代替。

把相应的键放入到方括号中（ `name[key]` ）：

```
>>> dict = {'Name': 'Runoob', 'Age': 7}
>>> print ("dict['Name']: ", dict['Name'])
dict['Name']:  Runoob
>>> print ("dict['Age']: ", dict['Age'])
dict['Age']:  7
```

添加元素

字典是一种动态数据结构，可随时在字典中添加键—值对。要添加键—值对时，可依次指定字典名、键和键对应的值。

下面在字典 `menu` 中添加两道菜的菜名和价格：

```
>>> #coding=utf-8
#创建并初始化menu字典
>>> menu = {'鱼':40, '猪肉':30, '番茄':15, '拉面':10}
#向menu字典中添加菜名和价格
>>> menu['果汁'] = 12
>>> menu['煎蛋'] = 2
######输出新的menu
>>> print(menu)
{'鱼': 40,'猪肉': 30,'番茄': 15,'拉面': 10, '果汁': 12,'煎蛋': 2}
```

新的 `menu` 字典包含 `6` 个 `键-值对` ，新增加的两个 `键-值对` （菜名和对应价格）添加在了原有 `键-值对` 的后面，注意字典中 `键-值对` 的排列顺序和添加顺序没有必然联系，Python 不关心字典中 `键-值对` 的排列顺序，而只关心键与值得对应关系。

同理，字典和列表一样，可以先创建一个空字典，然后可以不断向里面添加新的键-值对。

修改元素

字典和列表一样，里面的值都是可以修改的。要修改字典中的值，可直接指定字典中的键所对应的新值。例如，将 `menu` 中的 `fish` 价格改为 `50` 。

```
>>> #coding = utf-8
#创建并初始化menu字典
>>> menu = {'鱼':40, '猪肉':30, '番茄':15, '拉面':10}
# 修改menu字典中菜fish的价格
>>> menu['鱼'] = 50
# 打印输出新的menu
>>> print(menu)
{'鱼': 50, '猪肉': 30, '番茄': 15, '拉面': 10}
```

删除元素

我们可以通过 `del` 方法删除字典中不需要的 `键-值对` ，使用 `del` 方法时，要指定字典名和要删除的 `键` 。

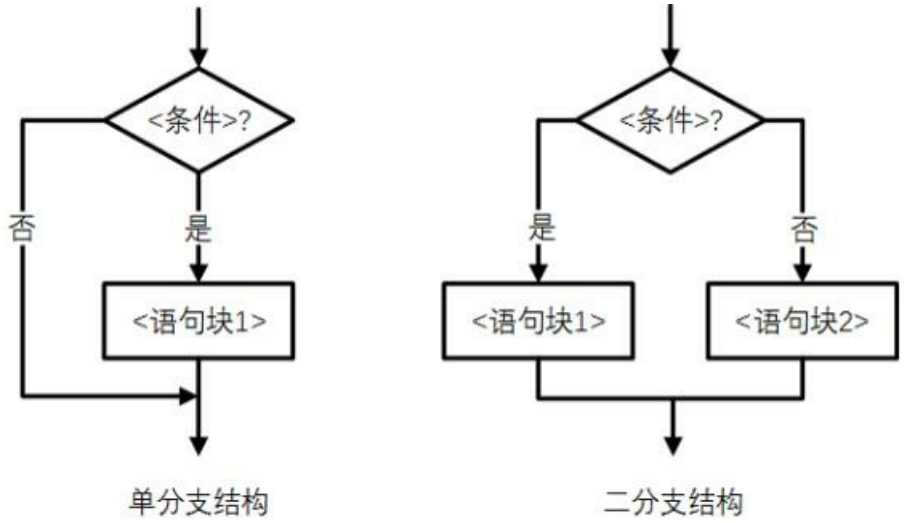
例如，在 `menu` 菜单中删除键 `noodles` 和它的值。

```
###### 创建并初始化menu字典
>>> menu = {'鱼':40, '猪肉':30, '番茄':15, '拉面':10}
###### 删除noodles键值对
>>> del menu['拉面']
###### 打印输出新的menu
>>> print(menu)
{'鱼':40, '猪肉':30, '番茄':15}
```

1.1.4 分支与循环

分支结构

`Python` 中的分支结构又叫条件语句， `Python` 条件语句是通过一条或多条语句的执行结果（ `True` 或者 `False` ）来决定执行的代码块。可以通过下图来简单了解条件语句的执行过程:



`Python` 语言中，分支结构的使用if语句实现，其一般的形式如下：

```
if <条件1>:
    <语句块1>
elif <条件2>:
    <语句块2>
elif <条件3>:
    <语句块3>
:
else:
    <语句块n>
```

当 `if` 语句执行时，首先会测试 `<条件1>` ，如果结果为 `True` ，则执行 `<语句块1>` ， `<条件1>` 为 `False` ，则会测试 `<条件2>` ，依此类推， `Python` 会执行第一个测试结果为 `True` 的 `<语句块>` ，如果所有的 `<条件>` 均为 `False` ，则执行 `else` 后的 `<语句块n>` 。任何语句的部分也还可以嵌套 `if` 语句。

例如现在需要编写一个程序将学生的数学成绩转换成 `A, B, C, D` 三等，规则是：成绩高于 `85` 分则为 `A` ，成绩在 `70到85分` 之间的为 `B` ，成绩在 `60到70分` 之间的为 `C` ，成绩低于 `60` 分的为 `D` .那么代码如下：

```
score = 88

if score > 85:
    print('A')
elif score >= 70:
    print('B')
elif score >= 60:
    print('C')
else
```

```
    print('D')
```

会看到代码的输出为: `A`

循环结构

for循环

`for` 循环在 `Python` 中是一个通用的迭代器，可以遍历任何有序的序列对象内的元素。 `for` 语句可用于字符串、列表、元组、文件等可迭代对象。 `for` 循环格式如下：

```
for <循环变量> in <遍历结构>:
    <语句块1>
else:
    <语句块2>
```

`for` 循环也可称为“遍历循环”，是因为 `for` 语句的循环执行次数是根据遍历结构中元素个数确定的。遍历循环可以理解为从遍历结构中逐一提取元素，放在循环变量中，对于每个所提取的元素执行一次语句块。

while循环

```
while <条件>:
    <语句块1>        #重复执行的部分，称为循环体
else:                #可选部分
    <语句块2>
```

如果不是执行 `break` 语句而结束循环， `<语句块2>` 才会被执行。

循环控制语句

循环控制语句可以更改语句执行的顺序。 `Python` 支持以下循环控制语句：

控制语句	描述
break语句	在语句块执行过程中终止循环，并且跳出整个循环
continue语句	在语句块执行过程中终止当前循环，跳出该次循环，执行下一次循环。
pass语句	pass是空语句，是为了保持程序结构的完整性。

如果要计算 `1-100` 的整数之和，从 `1` 写到 `100` 有点困难，幸好 `Python` 提供一个 `range()` 函数，可以生成一个整数序列，再通过 `list()` 函数可以转换为 `list` 。比如 `range(5)` 生成的序列是从 `0` 开始小于 `5` 的整数： `list(range(5))`

所以 `range(101)` 就可以生成 `0-100` 的整数序列，那么计算 `1` 到 `100` 的整数和代码如下：

```
sum = 0
for x in range(101):
    sum = sum + x
print(sum)
```

1.2.1 函数

函数是什么

函数一词来源于数学，但编程中的「函数」概念，与数学中的函数是有很大不同的，编程中的函数在英文中也有很多不同的叫法。在 `BASIC` 中叫做 `subroutine` (子过程或子程序)，在 `Pascal` 中叫做 `procedure` (过程)和 `function` ，在 `C` 中只有 `function` ，在 `Java` 里面叫做 `method` 。

定义: 函数是指将一组语句的集合通过一个名字(函数名)封装起来，要想执行这个函数，只需调用其函数名即可。

python中函数的定义

`Python` 中函数是通过保留字 `def` 来进行定义。语法格式如下：

```
def <函数名>([<形式参数>]):
    <函数体>
    return <返回值>
```

函数名可以是任何有效的 `Python` 标识符；参数是调用该函数时传递给它的值，可以有零个、一个或多个，当传递多个参数时各参数由逗号分隔，当没有参数时也要保留圆括号。

函数的参数

形参和实参

函数在调用的时候，可以传入参数，有形参和实参，简单点说，形参就是函数接收的参数，而实参就是你实际传入的参数。

形参：形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，形参只在函数内部有效。

实参：实参可以是常量、变量、表达式、函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须有确定的值，以便把这些值传送给形参。函数调用结束返回主调用函数后则不能再使用该形参变量。

```
def calc(x,y):#定义一个函数，参数有x和y，x和y就是形参
    print(x*y)#输出x乘以y的值

calc(5,2)#调用上面定义的函数，5和2就是实参
```

函数的四种形参类型

位置参数

位置参数，字面意思也就是按照参数的位置来进行传参，有几个位置参数在调用的时候就要传几个，否则就会报错了。

```
# name，sex为位置参数/必填参数
def my(name,sex):
    print(name,sex)
    return name

my('wwww','男')
```

默认参数

默认参数就是在定义形参的时候，给函数默认赋一个值，比如说数据库的端口这样的，默认给它一个值，这样就算你在调用的时候没传入这个参数，它也是有值的。

```
# port=3306为默认值参数
def connect(ip,port=3306):
    print(ip,port)

#如果给一个port值，则传新给的值
connect('118.1.1.1',3307)
#如果不填，则使用默认参数
connect('118.1.1.1')
```

可变参数

可变参数有以下几种特点：

`1` 、可变参数用 `*` 来接收，不是必传的； `2` 、它把传入的元素全部都放到了一个元祖里； `3` 、不显示参数个数，后面想传多少个参数就传多少个，它用在参数比较多的情况下 `4` 、如果位置参数、默认值参数、可变参数一起使用的的话，可变参数必须在位置参数和默认值参数后面。

```
#实例：发送报警短信 参数前面加*代表参数组
def send_sms(*phone_num):

    #方法1，返回的是元祖
    print(phone_num)
    #方法2，用下面循环的方法，不打印整个元祖，而是打印每一个元素
    # for p in phone_num:
    #     print(p)


send_sms()# 不传参数
send_sms(150)# 传1个
send_sms(151,152,153)# 传N个

# 执行后结果是：
# ()
# (150,)
# (151, 152, 153)
```

关键字参数

关键字参数有以下几种特点：

`1`、关键字参数使用`**`来接收；`2`、返回的是字典；`3`、不限制参数个数，非必传；`4`、当然也可以和上面的几种一起来使用，如果要一起使用的话，关键字参数必须在最后面。

```
def send_sms2(**phone_num):
    print(phone_num)

send_sms2()
send_sms2(name='xiaohei',sex='nan')
send_sms2(addr='北京',country='中国',aa='hahaha')

# 执行后结果是：
# {}
# {'name': 'xiaohei', 'sex': 'nan'}
# {'addr': '北京', 'country': '中国', 'aa': 'hahaha'}
```

函数的返回值-return语句

每个函数都有返回值，如果没有在函数里面指定返回值的话，在`Python`里面函数执行完之后，默认会返回一个`None`，函数也可以有多个返回值，如果有多个返回值的话，会把返回值都放到一个元组中，返回的是一个元组。

为什么要有返回值呢，是因为在这个函数操作完之后，它的结果在后面的程序里面需要用到。

函数中的返回值使用`return`，函数在遇到`return`就立即结束。

什么样的函数需要`return`，什么样的不需要？下面这个只是显示当前日期，就不需要`return`。例如：

```
import datetime

def get_today():
    print(datetime.datetime.today())
```

局部变量和全局变量

`1.` 局部变量意思就是在局部生效的，出了这个变量的作用域，这个变量就失效了。 `2.` 全局变量的意思就是在整个程序里面都生效的，在程序最前面定义的都是全局变量。 `3.` 全局变量如果要在函数中修改的话，需要加`global`关键字声明，如果是`list`、字典和集合的话，则不需要加`global`关键字，直接就可以修改。 `4.` 任何函数都可以修改，所以尽量少用全局变量，原因`1`不够安全。原因`2`全局变量一直占用内存。

```
#如果两种变量都有，会先用局部变量，当没有局部变量则用全局变量

#全局变量，在函数的外面
name='wangcan'

def get_name():

    # 声明修改全局变量
    global name

    # 局部变量，定义在函数内部
```

```
    name='hailong'
    print('函数里面的name：'+name)

def get_name2():
    print('get_name2：',name)

get_name2()  #第一调用，打印wangcan

get_name()  #第二调用，打印函数内的name:hailong
print('函数外面的name：'+name)# 最后打印,因为有函数声明了全局变量，所以打印的是：hailong


#最后的结果是
# get_name2：  wangcan
# 函数里面的name：hailong
# 函数外面的name：hailong
```

递归调用

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

递归调用的意思就是，在这个函数内部自己调用自己，就有点循环的意思，写个递归，如下：

```
#递归应用举例
def test1():
    num = int(input('please enter a number：'))
    if num%2==0:#判断输入的数字是不是偶数
       return True #如果是偶数的话，程序就退出了，返回true
    print('不是偶数请重新输入！')
    return test1()#如果不是偶数的话继续调用自己，输入值

print(test1())#调用test
```

递归调用的特性：

`1.` 必须有一个明确的结束条件。

`2.` 每次进入更深一层递归时，问题规模相比上次递归都应有所减少。

`3.` 递归效率不高，递归层次过多会导致栈溢出（在计算机中，函数调用是通过栈（ `stack` ）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出）。

1.2.2 模块

在 `Python` 程序的开发过程中，为了代码维护的方便，我们可以把函数进行分组，分别放到不同的 `.py` 文件里，这样，每个文件包含的代码就相对较少，这个 `.py` 文件就称之为一个模块（ `Module` ）。模块能够让我们有逻辑地组织 `Python` 代码段，模块中能够定义函数，类和变量，模块里也可以包含可执行的代码。

模块的引入

`Python` 中要用关键字 `import` 来引入某个模块，比如要引用模块 `math` ，就要在文件的开头用 `import math` 来引入。在调用 `math` 模块中的函数时，引用格式为：

`模块名.函数名`

因为这种调用方式可以避免特殊情况的发生：比如在多个模块中可能含有相同名称的函数，这时如果只是通过函数名来调用，程序无法知道是要调用哪个函数。所以如果用上述方法引入模块的时候，调用函数必须加上模块名。

例如：

```
import math

print(fabs(-2))
```

输出结果：

`NameError: name 'fabs' is not defined`

`fabs()` 必须加上 `math` 前缀，例如：

```
import math

print(math.fabs(-2))
```

输出结果：

`2`

有些时候我们只需要用到模块中的某个函数，这时不需要导入整个模块，只需要导入该函数即可，语句格式如下：

`from 模块名 import 函数名1,函数名2....`

通过这种方式导入函数的时候，调用函数时就只能给出函数名，而不能给出模块名了。这种方式导入函数的方法会有这种缺陷：当两个模块中含有相同名称函数的时候，后面一次导入的函数会覆盖前一次导入的函数。例如，假如模块A中有函数 `function()` ，模块B中也有函数 `function()` ，如果先导入模块A中的 `function()` 、后导入模块B中的 `function()` ，那么当我们在后面调用 `function()` 函数的时候，程序是去执行模块B中的 `function()` 函数。

如果想一次性引入模块 `math` 中所有的函数，可以通过如下方式导入：

`from math import *`

自定义模块

每个 Python 文件都可以看作一个模块，模块的名字就是 Python 文件的名字。所以我们完全可以自己写一个 Python 文件，就作为自己定义的模块。例如我们编写了 my_module.py 文件，里面定义了 plus() 函数：

```
#my_module.py

def plus(a,b):
    return a+b
```

之后我们就可以在其他 Python 文件中先 import my_module ，然后通过 my_module.plus(a,b) 来调用 my_module.py 文件中的 plus() 函数。我们也可以直接通过 from my_module import plus 来导入 plus() 函数。

内置模块中的内置函数

我们在安装好了 Python 后，也将 Python 本身带有的库也安装好了， Python 自带的库也叫做 Python 的内置模块。 Python 的内置模块是 Python 编程的重要组织形式，内置模块中的内置函数也极大方便了编程过程中对函数等功能的使用。

Python 中常见的内置模块如下：

1. os 模块：（文件和目录）用于提供系统级别的操作。
2. sys 模块：用于提供对解释器相关的操作。
3. json 模块：处理 JSON 字符串。
4. logging：用于便捷记录日志且线程安全的模块。
5. time&datetime 模块：时间相关的操作，时间有三种表示方式。
6. hashlib 模块：用于加密相关操作，代替了 md5 模块，主要是提供 SHA1 ， SHA224 ， SHA256 ， SHA384 ， SHA512 ， MD5 算法。
7. random 模块：提供随机数。

Python 的内置模块中也有很多使用十分方便的内置函数。

dir() 函数是一个排好序的字符串列表，包含的内容是一个模块里定义过的名字，包含在一个模块里定义的所有模块，变量和函数。例如：

```
# 导入内置math模块
import math

#调用math模块中的dir()函数
content = dir(math)

#输出math模块中所有模块、函数和变量的名字
print(content)
```

输出结果： ['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs','factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin','sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

程序输出了 `math` 模块中所有模块、函数和变量的名字。特殊字符串变量 `__name__` 是指向模块的名字，变量 `__file__` 是指向该模块的导入文件名。

`globals()` 和 `locals()` 函数可被用来返回全局和局部命名空间里的名字。如果在函数内部调用的是 `globals()` 函数，那么返回的是所有在该函数里能够访问的全局名字。如果在函数内部调用的 `locals()` 函数，返回的是能够在该函数里访问的局部命名。 `globals()` 函数和 `locals()` 函数的返回类型都是字典。所以名字们能用 `keys()` 函数摘取。

当一个模块被导入到一个脚本中后，程序只会将模块顶层部分的代码执行一次。因此，如果我们想再次执行模块顶层部分的代码，可以用 `reload()` 函数。该函数便会重新将之前导入过的模块导入。格式如下：

```
reload(module_name)
```

在这里， `module_name` 要直接放模块名，而不能是一个字符串形式。例如我们想重载 `hello` 模块：

```
reload(hello)
```

1.2.3 类与对象

面向对象编程——`Object Oriented Programming`，简称`OOP`，是一种程序设计思想。`OOP`把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

在`Python`中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（`Class`）的概念。

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个`dict`表示：

```
std1 = { 'name': 'Michael', 'score': 98 }
std2 = { 'name': 'Bob', 'score': 81 }
```

而处理学生成绩可以通过函数实现，比如打印学生的成绩:

```
def print_score(std):
    print('%s: %s' % (std['name'], std['score']))
```

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是`Student`这种数据类型应该被视为一个对象，这个对象拥有`name`和`score`这两个属性（`Property`）。如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个`print_score`消息，让对象自己把自己的数据打印出来。

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（`Method`）。面向对象的程序写出来就像这样：

```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.print_score()
lisa.print_score()
```

面向对象的设计思想是从自然界中来的，因为在自然界中，类（ Class ）和对象（ Instance ）的概念是很自然的。 Class 是一种抽象概念，比如我们定义的 Class—Student ，是指学生这个概念，而对象（ Instance ）则是一个个具体的 Student ，比如， Bart Simpson 和 Lisa Simpson 是两个具体的 Student 。

所以，面向对象的设计思想是抽象出 Class ，根据 Class 创建 Instance 。

面向对象的抽象程度又比函数要高，因为一个 Class 既包含数据，又包含操作数据的方法。

类和对象

面向对象最重要的概念就是类（ Class ）和实例（ Instance ），必须牢记类是抽象的模板，比如 Student 类，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以 Student 类为例，在 Python 中，定义类是通过 class 关键字：

```
class Student(object):
    pass
```

class 后面紧接着是类名，即 Student ，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用 object 类，这是所有类最终都会继承的类。

定义好了 Student 类，就可以根据 Student 类创建出 Student 的实例，创建实例是通过类名+()实现的：

```
>>> bart = Student()
>>> bart
<__main__.Student object at 0x10a67a590>
>>> Student
<class '__main__.Student'>
```

可以看到，变量 bart 指向的就是一个 Student 的实例，后面的 0x10a67a590 是内存地址，每个 object 的地址都不一样，而 Student 本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例 bart 绑定一个 name 属性：

```
>>> bart.name = 'Bart Simpson'
>>> bart.name
'Bart Simpson'
```

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的 __init__ 方法，在创建实例的时候，就把 name ， score 等属性绑上去：

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
```

注意到 __init__ 方法的第一个参数永远是 self ，表示创建的实例本身，因此，在 __init__ 方法内部，就可以把各种属性绑定到 self ，因为 self 就指向创建的实例本身。

有了 `__init__` 方法，在创建实例的时候，就不能传入空的参数了，必须传入与 `__init__` 方法匹配的参数，但 `self` 不需要传， `Python` 解释器自己会把实例变量传进去：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.name
'Bart Simpson'
>>> bart.score
59
```

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量 `self` ，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

数据封装

面向对象编程的一个重要特点就是数据封装。在上面的 `Student` 类中，每个实例就拥有各自的 `name` 和 `score` 这些数据。我们可以通过函数来访问这些数据，比如打印一个学生的成绩：

```
>>> def print_score(std):
...     print('%s: %s' % (std.name, std.score))
...
>>> print_score(bart)
Bart Simpson: 59
```

但是，既然 `Student` 实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在 `Student` 类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和 `Student` 类本身是关联起来的，我们称之为类的方法：

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

要定义一个方法，除了第一个参数是 `self` 外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了 `self` 不用传递，其他参数正常传入：

```
>>> bart.print_score()
Bart Simpson: 59
```

这样一来，我们从外部看 `Student` 类，就只需要知道，创建实例需要给出 `name` 和 `score` ，而如何打印，都是在 `Student` 类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

封装的另一个好处是可以给 `Student` 类增加新的方法，比如 `get_grade` ：

```
class Student(object):
```

```
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

同样的， `get_grade` 方法可以直接在实例变量上调用，不需要知道内部实现细节。

继承和多态

在 `OOP` 程序设计中，当我们定义一个 `class` 的时候，可以从某个现有的 `class` 继承，新的 `class` 称为子类（ `Subclass` ），而被继承的 `class` 称为基类、父类或超类（ `Base class` 、 `Super class` ）。

比如，我们已经编写了一个名为 `Animal` 的 `class` ，有一个 `run()` 方法可以直接打印：

```
class Animal(object):
    def run(self):
        print('Animal is running...')
```

当我们需要编写 `Dog` 和 `Cat` 类时，就可以直接从 `Animal` 类继承：

```
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

对于 `Dog` 来说， `Animal` 就是它的父类，对于 `Animal` 来说， `Dog` 就是它的子类。 `Cat` 和 `Dog` 类似。

继承有什么好处？最大的好处是子类获得了父类的全部功能。由于 `Animial` 实现了 `run()` 方法，因此， `Dog` 和 `Cat` 作为它的子类，什么事也没干，就自动拥有了 `run()` 方法：

```
dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

运行结果如下：

```
Animal is running...
Animal is running...
```

当然，也可以对子类增加一些方法，比如 `Dog` 类:

```
class Dog(Animal):
    def run(self):
        print('Dog is running...')

    def eat(self):
        print('Eating meat...')
```

继承的第二个好处需要我们对代码做一点改进。你看到了，无论是 `Dog` 还是 `Cat` ，它们 `run()` 的时候，显示的都是 `Animal is running...` ，符合逻辑的做法是分别显示 `Dog is running...` 和 `Cat is running...` ，因此，对 `Dog` 和 `Cat` 类改进如下：

```
class Dog(Animal):
    def run(self):
        print('Dog is running...')

class Cat(Animal):
    def run(self):
        print('Cat is running...')
```

再次运行，结果如下：

```
Dog is running...
Cat is running...
```

当子类和父类都存在相同的 `run()` 方法时，我们说，子类的 `run()` 覆盖了父类的 `run()` ，在代码运行的时候，总是会调用子类的 `run()` 。这样，我们就获得了继承的另一个好处：多态。

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个 `class` 的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和 `Python` 自带的数据类型，比如 `str、list、dict` 没什么两样：

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型
```

判断一个变量是否是某个类型可以用 `isinstance()` 判断：

```
>>> isinstance(a, list)
True
>>> isinstance(b, Animal)
True
>>> isinstance(c, Dog)
True
```

看来 `a、b、c` 确实对应着 `list、Animal、Dog` 这 `3` 种类型。

但是等等，试试：

```
>>> isinstance(c, Animal)
True
```

看来 `c` 不仅仅是 `Dog` ， `c` 还是 `Animal` ！

不过仔细想想，这是有道理的，因为 `Dog` 是从 `Animal` 继承下来的，当我们创建了一个 `Dog` 的实例 `c` 时，我们认为 `c` 的数据类型是 `Dog` 没错，但 `c` 同时也是 `Animal` 也没错， `Dog` 本来就是 `Animal` 的一种！

所以，在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

```
>>> b = Animal()
>>> isinstance(b, Dog)
False
```

`Dog` 可以看成 `Animal` ，但 `Animal` 不可以看成 `Dog` 。

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个 `Animal` 类型的变量：

```
def run_twice(animal):
    animal.run()
    animal.run()
```

当我们传入 `Animal` 的实例时， `run_twice()` 就打印出：

```
>>> run_twice(Animal())
Animal is running...
Animal is running...
```

当我们传入 `Dog` 的实例时， `run_twice()` 就打印出：

```
>>> run_twice(Dog())
Dog is running...
Dog is running...
```

当我们传入 `Cat` 的实例时， `run_twice()` 就打印出：

```
>>> run_twice(Cat())
Cat is running...
Cat is running...
```

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个 `Tortoise` 类型，也从 `Animal` 派生：

```
class Tortoise(Animal):
    def run(self):
        print('Tortoise is running slowly...')
```

当我们调用 `run_twice()` 时，传入 `Tortoise` 的实例：

```
>>> run_twice(Tortoise())
Tortoise is running slowly...
Tortoise is running slowly...
```

你会发现，新增一个 `Animal` 的子类，不必对 `run_twice()` 做任何修改，实际上，任何依赖 `Animal` 作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

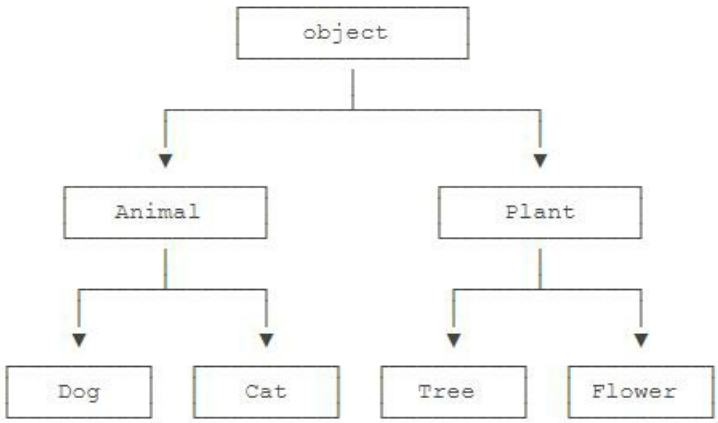
多态的好处就是，当我们需要传入 `Dog`、`Cat`、`Tortoise……` 时，我们只需要接收 `Animal` 类型就可以了，因为 `Dog`、`Cat`、`Tortoise……` 都是 `Animal` 类型，然后，按照 `Animal` 类型进行操作即可。由于 `Animal` 类型有 `run()` 方法，因此，传入的任意类型，只要是 `Animal` 类或者子类，就会自动调用实际类型的 `run()` 方法，这就是多态的意思：

对于一个变量，我们只需要知道它是 `Animal` 类型，无需确切地知道它的子类型，就可以放心地调用 `run()` 方法，而具体调用的 `run()` 方法是作用在 `Animal`、`Dog`、`Cat` 还是 `Tortoise` 对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种 `Animal` 的子类时，只要确保 `run()` 方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增 `Animal` 子类；

对修改封闭：不需要修改依赖 `Animal` 类型的 `run_twice()` 等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类 `object`，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



1.2.4 文件IO

我们现在生活在信息爆炸的时代，计算机中文本文件可存储的数据量多得难以置信，我们可以把各种信息都存储在文本文件中。每当我们需要利用程序去修改或分析存储在文本文件中的信息时，就先必须正确地读取文件。

要用 `Python` 程序去修改或分析文本文件中的信息，首先就需要将文本文件中的信息读取到内存中。而且我们既可以将文本文件中的内容一次性读取，也可以将信息按每次一行的方法逐步读取。

读取整个文件

一般我们读取的文件和编写的Python文件在同一目录下，例如在当前目录下我们已经创建了要处理的文件 `test.txt` ，里面包含的内容为：

```
Hello,world!
Hello,Python!
Hello,my brothers.
```

我们运行在同一目录下的 `Python` 文件 `test.py` ，代码如下：

```
with open('test.txt') as file_object:
    contents = file_object.read()
    print(contents)
```

程序运行结果：

```
Hello,world!
Hello,Python!
Hello,my brothers.
```

`test.py` 文件中的第一行代码中有函数 `open()` ，用于打开文件，这是我们处理文件的第一步。函数 `open()` 中的参数 `'test.txt'` 就是要打开的文件。函数 `open()` 返回的是打开文件的对象，第一行代码就是把文本文件 `test.txt` 打开，并将其对象保存在 `file_object` 变量中。

关键字 `with` 的功能是在不再需要访问文件后自动将文件关闭。所以我们在这里只是 `open()` 打开了文件，但是没有加入 `close()` 代码关闭文件，因为Python会在处理文件之后自动将文件关闭。

`test.py` 文件中的第二行代码是使用 `read()` 方法读取文本文件 `test.txt` 的全部内容，并将内容保存在字符串变量 `contents` 中，然后通过 `print()` 将结果都输出。

如果我们要处理的文本文件和Python程序文件不在同一目录下，那么我们就要在 `open()` 函数中传入文本文件的绝对路径。

逐行读取

我们也可以将文本文件中的数据进行逐行读取，我们要处理的文本文件还是上面的 `test.txt` ，这时我们可以逐行将 `test.txt` 的内容输出，代码如下：

```
with open('test.txt') as file_object:
    for line in file_object:
        print(line)
```

输出结果：

```
Hello,world!

Hello,Python!

Hello,my brothers.
```

我们会发现输出结果中每一行内容后面都多了一个空行，这时因为在文件中每一行的末尾都会有一个换行符，而每条 `print()` 语句也会加上一个换行符，所以每行末尾都有两个换行符，所以输出之后就会多一个空行。

我们可以采取 `rstrip()` 方法消除空行，代码如下：

我们会发现输出结果中每一行内容后面都多了一个空行，这时因为在文件中每一行的末尾都会有一个换行符，而每条 `print()` 语句也会加上一个换行符，所以每行末尾都有两个换行符，所以输出之后就会多一个空行。

我们可以采取 `rstrip()` 方法消除空行，代码如下：

```
with open('test.txt') as file_object:
    for line in file_object:
        print(line.rstrip())
```

这时输出的结果中就没有多余的空行了：

```
Hello,world!
Hello,Python!
Hello,my brothers.
```

写入空文件

要将信息写入文本文件中，我们依然用 `open()` 方法，只不过除了将文本文件名当作参数传入函数 `open()` 中去之外，还需要再传入写参数 `w` ，例如编写 `Python` 程序 `test2.py` ：

```
with open('test2.txt','w') as example:
    example.write('Hello world!')
```

程序运行结果就是在 `test2.py` 文件所在目录生成了一个名为 `test2.txt` 的文本文件，文本文件中的内容为：

```
Hello world!
```

在这个例子中，调用函数 `open()` ，传入两个参数，一个是我们要创建的文件为 `test2.txt` ，还有一个就是参数 `w` ，代表的是写入命令。

我们也可以往 `test2.txt` 中传入多行信息，例如：

```
with open('test2.txt','w') as example:
    example.write('Hello world!\n')
    example.write('Hello python!\n')
```

该段程序中第二行 `write()` 函数中还加入了换行符 `\n` ，这样就可以将加入的消息分行了。

程序运行之后生成的 `test2.txt` 的内容为：

```
Hello world!
Hello python!
```

附件到文件

但我们要注意的是使用 `w` 命令时如果写入的文本文件原来已经存在了， `Python` 将会在写入内容前将文本文件中原来的内容全部清空。所以我们如果要在已经存在的文本文件中添加信息内容，而不是将原来的信息内容都清除，就要采取附加模式打开文件。

我们用附加模式打开文件时， `Python` 会将我们写入的行直接加入到文本文件原有信息的末尾，如果指定的文本文件不存在，才会新建一个文本文件。

例如现在已经存在了文本文件 `test2.txt` ，里面的内容为：

```
Hello world!
Hello python!
```

我们现在想在文本文件 `test2.txt` 中再加入两句话：

`Hello my brothers!` `Hello my sisters!`

实现代码如下：

```
with open('test2.txt','a') as example:
    example.write('Hello my brothers!\n')
    example.write('Hello my sisters!\n')
```

我们在函数 `open()` 中传入了参数 `a` ,告诉 `Python` 程序我们是将信息加入到文本文件的末尾，而不是覆盖文件。

程序运行之后，文本文件 `test2.txt` 中的内容变为：

```
Hello world!
Hello python!
Hello my brothers!
Hello my sisters!
```

1.2.5 异常处理

异常的基本概念

究竟什么是异常呢？下面的例子可以让你茅塞顿开：

```
x, y = 12, 5
a = x / y
print(A)        #拼写错误，Python对大小写敏感， 并未定义变量A
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'A' is not defined
```

```
1/0
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

当 `Python` 检测到一个错误时，解释器就会指出当前程序流已经无法继续执行，这时候就出现了异常。当程序出现错误时 `Python` 会自动引发异常，程序员也可以通过 `raise` 语句显式地引发异常。

try...except...结构

异常处理结构中最常见也最基本的结构是 `try...except...` 结构。 `try` 子句中的代码块包含可能出现异常的语句，而 `except` 子句用来捕捉相应的异常， `except` 子句中的代码块用来处理异常。

```
try:
    try 块          #被监控的语句，可能引发异常
except Exception,e:
    except 块       #处理异常的代码
```

如果 `try` 中的代码块没有出现异常，则跳过 `except` 块执行后面的代码，如果出现异常并被 `except` 捕获，则执行 `except` 块中的代码，如果出现异常但未被 `except` 捕获，则继续向外层抛出。如果所有层都没有捕获并处理该异常，则程序终止并将异常抛给用户。

else与finally

另一种常用的结构是 `try...except...else...` 语句。如果 `try` 中的代码抛出异常，并被某个 `except` 捕捉则执行 `except` 块中的内容，这种情况下不执行 `else` 的内容，如果 `try` 中的代码没有发生异常，则执行 `else` 块。例如：

```
a_list=['abc','def','ghi']
print('请输入字符串序号')
```

```
while True:
    n=input()
    try:
         print a_list[n]
    except IndexError:
        print('输入不正确')
    else:
        break
```

还有一种异常处理结构是 `try...except...finally...` 结构。该结构中， `finally` 子句中的代码无论异常是否发生都会被执行。它常用来做一些清理工作以释放资源。

例如：

```
try:
    f=open('test.txt','r')
finally:
    f.close()
```

自定义异常的方法

`Python` 有许多内置异常类，如果需要的话也可以继承 `Python` 的内置异常类来实现自定义异常类，例如：

```
class ShortInputException(Exception):
     #自己定义的异常类
     def __init__(self,length,atleast):
         Exception.__init__(self)
         self.length=length
         self.atleast=atleast
     def __str__(self):
         return '输入的长度是 %d, 长度至少应是 %d' %(self.length, self.atleast)


try:
    s=raw_input('请输入: ')
    if len(s) < 3:
        raise ShortInputException(len(s), 3)
except ShortInputException as e:
    print(e)
else:
    print('无异常发生')
```

1.2.6 正则表达式

正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，来筛选出符合这个规则的内容。

可以简单理解为：一个强大的搜索工具中，正则表达式就是你要搜索内容的条件表达式。

正则表达式基础知识

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的 `Email` 地址，虽然可以编程提取@前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的。

所以我们判断一个字符串是否是合法的Email的方法是：

`1.` 创建一个匹配Email的正则表达式；

`2.` 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符来描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用 `\d` 可以匹配一个数字，`\w` 可以匹配一个字母或数字，所以：

- `'00\d'` 可以匹配 `'007'`，但无法匹配 `'00A'`
- `'\d\d\d'` 可以匹配 `'010'`
- `'\w\w\d'` 可以匹配 `'py3'`

`.` 可以匹配任意字符，所以：

- `'py.'` 可以匹配 `'pyc'`、`'pyo'`、`'py!'` 等等。

要匹配变长的字符，在正则表达式中，用 `*` 表示任意个字符（包括 `0` 个），用 `+` 表示至少一个字符，用 `?` 表示 `0` 个或 `1` 个字符，用 `{n}` 表示 `n` 个字符，用 `{n,m}` 表示 `n-m` 个字符：

来看一个复杂的例子：`\d{3}\s+\d{3,8}`。

我们来从左到右解读一下：

- `\d{3}` 表示匹配 `3` 个数字，例如 `'010'`。
- `\s` 可以匹配一个空格（也包括 `Tab` 等空白符），所以 `\s+` 表示至少有一个空格，例如匹配 `' '`, `'  '` 等。
- `\d{3,8}` 表示 `3-8` 个数字，例如 `'1234567'`。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配 `'010-12345'` 这样的号码呢？由于 `'-'` 是特殊字符，在正则表达式中，要用 `'\'` 转义，所以，上面的正则是 `\d{3}\-\d{3,8}` 。

但是，仍然无法匹配 `'010 - 12345'` ，因为带有空格。所以我们需要更复杂的匹配方式。

正则表达式进阶知识

要做更精确地匹配，可以用 `[]` 表示范围，比如：

- `[0-9a-zA-Z\_]` 可以匹配一个数字、字母或者下划线。
- `[0-9a-zA-Z\_]+` 可以匹配至少由一个数字、字母或者下划线组成的字符串，比如 `'a100','0_Z', 'Py3000'` 等等。
- `[a-zA-Z\_][0-9a-zA-Z\_]*` 可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是 `Python` 合法的变量。
- `[a-zA-Z\_][0-9a-zA-Z\_]{0, 19}` 更精确地限制了变量的长度是 `1-20` 个字符（前面 `1` 个字符 `+` 后面最多 `19` 个字符）。

`A|B` 可以匹配 `A` 或 `B` ，所以 `(P|p)ython` 可以匹配 `'Python'` 或者 `'python'` 。

`^` 表示行的开头， `^\d` 表示必须以数字开头。

`$` 表示行的结束， `\d$` 表示必须以数字结束。

你可能注意到了， `py` 也可以匹配 `'python'` ，但是加上 `^py$` 就变成了整行匹配，就只能匹配 `'py'` 了。

Python中的re模块

有了准备知识，我们就可以在 `Python` 中使用正则表达式了。 `Python` 提供 `re` 模块，包含所有正则表达式的功能。由于 `Python` 的字符串本身也用 `\` 转义，所以要特别注意：

```
s = 'ABC\\-001' # Python的字符串
# 对应的正则表达式字符串变成：
# 'ABC\-001'
```

因此我们强烈建议使用 `Python` 的 `r` 前缀，就不用考虑转义的问题了：

```
s = r'ABC\-001' # Python的字符串
# 对应的正则表达式字符串不变：
# 'ABC\-001'
```

先看看如何判断正则表达式是否匹配：

```
>>> import re
>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> re.match(r'^\d{3}\-\d{3,8}$', '010 12345')
>>>
```

`match()` 方法判断是否匹配，如果匹配成功，返回一个 `Match` 对象，否则返回 `None` 。常见的判断方法就是：

```
test = '用户输入的字符串'
if re.match(r'正则表达式', test):
    print('ok')
else:
    print('failed')
```

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用 `()` 表示的就是要提取的分组（ `Group` ）。比如：

`^(\d{3})-(\d{3,8})$` 分别定义了两个组，可以直接从匹配的字符串中提取出区号和本地号码：

```
>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')
>>> m
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> m.group(0)
'010-12345'
>>> m.group(1)
'010'
>>> m.group(2)
'12345'
```

如果正则表达式中定义了组，就可以在 `Match` 对象上用 `group()` 方法提取出子串来。注意到 `group(0)` 永远是原始字符串， `group(1)` 、 `group(2)` …… 表示第 `1`、`2`、…… 个子串。

提取子串非常有用。来看一个更凶残的例子：

```
>>> t = '19:05:30'
>>> m = re.match(r'^(0[0-9]|1[0-9]|2[0-3]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])\:(0[0-9]|
1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])$', t)
>>> m.groups()
('19', '05', '30')
```

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

```
'^(0[1-9]|1[0-2]|[0-9])-(0[1-9]|1[0-9]|2[0-9]|3[0-1]|[0-9])$'
```

对于 `'2-30'` ， `'4-31'` 这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合识别了。

最后需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的 `0` ：

```
>>> re.match(r'^(\d+)(0*)$', '102300').groups()
('102300', '')
```

由于 `\d+` 采用贪婪匹配，直接把后面的 `0` 全部匹配了，结果 `0*` 只能匹配空字符串了。必须让 `\d+` 采用非贪婪匹配（也就是尽可能少匹配），才能把后面的 `0` 匹配出来，加个 `?` 就可以让 `\d+` 采用非贪婪匹配：

```
>>> re.match(r'^(\d+?)(0*)$', '102300').groups()
('1023', '00')
```

当我们在 `Python` 中使用正则表达式时， `re` 模块内部会干两件事情：

- 编译正则表达式，如果正则表达式的字符串本身不合法，会报错。
- 用编译后的正则表达式去匹配字符串。

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

```
>>> import re
# 编译:
>>> re_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')
# 使用:
>>> re_telephone.match('010-12345').groups()
('010', '12345')
>>> re_telephone.match('010-8086').groups()
('010', '8086')
```

编译后生成 `Regular Expression` 对象，由于该对象自己包含了正则表达式，所以调用对应的方法时不用给出正则字符串。

Chapter2 Numpy

NumPy 是 Python 语言的一个扩充程序库。支持高级大量的维度数组与矩阵运算，此外也针对数组运算提供大量的数学函数库。Numpy 内部解除了 Python 的 PIL(全局解释器锁)，运算效率极好，是大量机器学习框架的基础库!

Numpy 相关实训已在 `educoder` 平台上提供，若感兴趣可以输入链接进行体验。

链接：https://www.educoder.net/paths/302

2.1.1：ndarray对象

怎样安装 NumPy

NumPy (Numerical Python) 是 Python 语言的一个扩展程序库，支持大量的维度数组与矩阵运算，此外也针对数组运算提供大量的数学函数库。是在学习机器学习、深度学习之前应该掌握的一个非常基本且实用的 python 库。

想要安装 NumPy 其实非常简单，进入命令行，输入如下代码即可：

```
pip install numpy
```

什么是 ndarray 对象

NumPy 为什么能够受到各个数据科学从业人员的青睐与追捧，其实很大程度上是因为 NumPy 在向量计算方面做了很多优化，接口也非常友好。而这些其实都是在围绕着 NumPy 的一个核心数据结构 ndarray 。

ndarray 的全称是 N-Dimension Arrary ，字面意义上其实已经表明了一个 ndarray 对象就是一个 N 维数组。但要注意的是，ndarray 是同质的。同质的意思就是说 **N** 维数组里的所有元素必须是属于同一种数据类型的。**(PS: python** 中的 **list** 是异质的**)**

ndarray 对象实例化好了之后，包含了一些基本的属性。比如 shape ， ndim ， size ， dtype 。其中：

- shape : ndarray 对象的形状，由一个 tuple 表示
- ndim : ndarray 对象的维度
- size : ndarray 对象中元素的数量
- dtype : ndarray 对象中元素的数据类型，例如 int64 ， float32 等。

来看个例子，假设现在有一个 3 行 5 列的矩阵（ ndarray ）如下：

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

那么该 ndarray 的 shape 是 (3, 5) 。(代表 3 行 5 列)；ndim 是 2 (因为矩阵有行和列两个维度)；size 是 15 (因为矩阵总共有 15 个元素)； dtype 是 int32 (因为矩阵中元素都是整数，并且用 32 位整型足够表示矩阵中的元素)；

打印这些属性的示例代码如下：

```
# 导入numpy并取别名为np
import numpy as np

# 构造ndarray
a = np.arange(15).reshape(3, 5)

# 打印a的shape，ndim，size，dtype
```

```
print(a.shape)
print(a.ndim)
print(a.size)
print(a.dtype)
```

输出如下：

```
(3, 5)
2
15
int32
```

如何实例化 **ndarray** 对象

实例化 ndarray 对象的函数有很多种，但最为常用的函数是 array ， zeros ， ones 以及 empty 。

使用 **array** 函数实例化 **ndarray** 对象

如果你手头上有一个 python 的 list ， 想要将这个 list 转成 ndarray ， 此时可以使用 NumPy 中的 array 函数将 list 中的值作为初始值，来实例化一个 ndarray 对象。代码如下：

```
import numpy as np

# 使用列表作为初始值，实例化ndarray对象a
a = np.array([2,3,4])

# 打印ndarray对象a
print(a)
```

输出如下：

```
[2, 3, 4]
```

使用 **zeros** ， **ones** ， **empty** 函数实例化 **ndarray** 对象

通常在写代码的时候，数组中元素的值一般都喜欢先初始化成 0 ， 如果使用 array 的方式实例化 ndarray 对象的话，虽然能实现功能，但显得很麻烦（首先要有一个全是 **0** 的 **list**）。那有没有简单粗暴的方式呢，有！！那就是 zeros 函数，你只需要把 ndarray 的 shape 作为参数传进去即可。代码如下：

```
import numpy as np

# 实例化ndarray对象a，a是一个3行4列的矩阵，矩阵中元素全为0
a = np.zeros((3, 4))

# 打印ndarray对象a
print(a)
```

输出如下:

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

如果想把数组中的元素全部初始化成 1 ，聪明的你应该能想到就是用 ones 函数，oens 的用法与 zeros 一致。代码如下：

```
import numpy as np

# 实例化ndarray对象a，a是一个3行4列的矩阵，矩阵中元素全为1
a = np.ones((3, 4))

# 打印ndarray对象a
print(a)
```

输出如下:

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

如果 01大法 满足不了你，想要用随机值作为初始值来实例化 ndarray 对象，empty 函数能够满足你。empty 的使用方式与 zeros 和 ones 如出一辙，代码如下：

```
import numpy as np

# 实例化ndarray对象a，a是一个2行3列的矩阵，矩阵中元素全为随机值
a = np.empty((2, 3))

# 打印ndarray对象a
print(a)
```

输出可能如下：

```
[[2.67276450e+185 1.69506143e+190 1.75184137e+190]
 [9.48819320e+077 1.63730399e-306 0.00000000e+000]]
```

2.1.2：形状操作

改变形状

上一节介绍了怎样实例化 ndarray 对象，比如想实例化一个 3 行 4 列的二维数组，并且数组中的值全为 0 。就可能会写如下代码：

```
import numpy as np

a = np.zeros((3, 4))
```

那如果想把 a 变成 4 行 3 列的二维数组，怎么办呢？机智的您可能会想到这样的代码：

```
import numpy as np

a = np.zeros((3, 4))

# 直接修改shape属性
a.shape = [4, 3]
```

最后您会发现，这样的代码可以完成功能，但是这种直接改属性的方式太粗暴了，不符合良好的编程规范。

更加优雅的解决方式是使用 NumPy 为我们提供了一个用来改变 ndarray 对象的 shape 的函数，叫 **reshape**。

NumPy 为了照顾偏向于面向对象或者这偏向于面向过程这两种不同风格的程序员，提供了两种调用**reshape**函数的方式(其实很多函数都提供了两种风格的接口)。

如果你更偏向于面向对象，那么你可以想象成 ndarray 对象中提供好了一个叫 reshape 成员函数。代码如下:

```
import numpy as np

a = np.zeros((3, 4))

# 调用a的成员函数reshape将3行4列改成4行3列
a = a.reshape((4, 3))
```

如果你更偏向于面向过程，NumPy 在它的作用域内实现了 reshape 函数。代码如下：

```
import numpy as np

a = np.zeros((3, 4))

# 调用reshape函数将a变形成4行3列
a = np.reshape(a, (4, 3))
```

不过需要注意的是，不管是哪种方式的 **reshape** ，都不会改变源 **ndarray** 的形状，而是将源 **ndarray** 进行深拷贝并进行变形操作，最后再将变形后的数组返回出去。也就是说如果代码是 `np.reshape(a, (4, 3))` 那么 **a** 的形状不会被修改！

如果想优雅的直接改变源 ndarray 的形状，可以使用 resize 函数。代码如下：

```
import numpy as np

a = np.zeros((3, 4))

# 将a从3行4列的二维数组变成一个有12个元素的一维数组
a.resize(12)
print(a)
```

输出如下：

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

有的时候懒得去算每个维度上的长度是多少，比如现在有一个 6 行 8 列的 ndarray ，然后想把它变形成有 2 列的 ndarray (行的数量我懒得去想)，此时我们可以在行的维度上传个 -1 即可，代码如下：

```
import numpy as np

a = np.zeros((6, 8))

# 行的维度上填-1，会让numpy自己去推算出行的数量，很明显，行的数量应该是24
a = a.reshape((-1, 2))
```

也就是说在变形操作时，如果某个维度上的值为 -1 ，那么该维度上的值会根据其他维度上的值自动推算。

-1 虽好，可不能贪杯！如果代码改成 `a = a.reshape((-1, -1))` ，**NumPy** 会认为你是在刁难他，并向你抛出异常 `ValueError: can only specify one unknown dimension` 。

2.1.3：基础操作

算术运算

如果想要对 ndarray 对象中的元素做 elementwise (逐个元素地)的算术运算非常简单，加减乘除即可。代码如下：

```
import numpy as np

a = np.array([0, 1, 2, 3])

# a中的所有元素都加2，结果为[2, 3, 4, 5]
b = a + 2

# a中的所有元素都减2，结果为[-2, -1, 0, 1]
c = a - 2

# a中的所有元素都乘以2，结果为[0, 2, 4, 6]
d = a * 2

# a中的所有元素都平方，结果为[0, 1, 4, 9]
e = a ** 2

# a中的所有元素都除以2，结果为[0, 0.5, 1, 1.5]
f = a / 2

# a中的所有元素都与2比，结果为[True, True, False, False]
g = a < 2
```

矩阵运算

相同 shape 的矩阵 A 与矩阵 B 之间想要做 elementwise 运算也很简单，加减乘除即可。代码如下：

```
import numpy as np

a = np.array([[0, 1], [2, 3]])
b = np.array([[1, 1], [3, 2]])

# a与b逐个元素相加，结果为[[1, 2], [5, 5]]
c = a + b

# a与b逐个元素相减，结果为[[-1, 0], [-1, 1]]
d = a - b

# a与b逐个元素相乘，结果为[[0, 1], [6, 6]]
e = a * b

# a的逐个元素除以b的逐个元素，结果为[[0., 1.], [0.66666667, 1.5]]
f = a / b

# a与b逐个元素做幂运算，结果为[[0, 1], [8, 9]]
g = a ** b
```

```
# a与b逐个元素相比较，结果为[[True, False], [True, False]]
h = a < b
```

细心的同学应该发现了， 只能做 *elementwise* 运算，如果想做真正的矩阵乘法运算显然不能用 。 NumPy 提供了 @ 和 dot 函数来实现矩阵乘法。代码如下：

```
import numpy as np

A = np.array([[1, 1], [0, 1]])
B = np.array([[2, 0], [3, 4]])

# @表示矩阵乘法，矩阵A乘以矩阵B，结果为[[5, 4], [3, 4]]
print(A @ B)

# 面向对象风格，矩阵A乘以矩阵B，结果为[[5, 4], [3, 4]]
print(A.dot(B))

# 面向过程风格，矩阵A乘以矩阵B，结果为[[5, 4], [3, 4]]
print(np.dot(A, B))
```

输出如下：

```
[[5 4]
 [3 4]]
[[5 4]
 [3 4]]
[[5 4]
 [3 4]]
```

简单统计

有的时候想要知道 ndarray 对象中元素的和是多少，最小值是多少，最小值在什么位置，最大值是多少，最大值在什么位置等信息。这个时候可能会想着写一个循环去遍历 ndarray 对象中的所有元素来进行统计。NumPy 为了解放我们的双手，提供了 sum ， min ， max ， argmin ， argmax 等函数来实现简单的统计功能，代码如下：

```
import numpy as np

a = np.array([[-1, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 13]])

# 计算a中所有元素的和，结果为67
print(a.sum())

# 找出a中最大的元素，结果为13
print(a.max())

# 找出a中最小的元素，结果为-1
print(a.min())

# 找出a中最大元素在a中的位置，由于a中有12个元素，位置从0开始计，所以结果为11
print(a.argmax())
```

```
# 找出a中最小元素在a中位置，结果为0
print(a.argmin())
```

输出如下:

```
67
13
-1
11
0
```

有的时候，我们在统计时需要根据轴来统计。举个例子，公司员工的基本工资，绩效工资，年终奖的信息如下：

工号	基本工资	绩效工资	年终奖
1	3000	4000	20000
2	2700	5500	25000
3	2800	3000	15000

这样一个表格很明显，可以用 ndarray 来存储。代码如下：

```
import numpy as np

info = np.array([[3000, 4000, 20000], [2700, 5500, 25000], [2800, 3000, 15000]])
```

info 实例化之后就有了维度和轴的概念，很明显 info 是个二维数组，所以它的维度是 **2** 。维度为 2 换句话来说就是 info 有两个轴：**0** 号轴与 **1** 号轴（轴的编号从 **0** 开始算）。轴所指的方向如下图所示：

如果想要统计下这 3 位员工中基本工资、绩效工资与年终奖的最小值与最大值（也就是说分别统计出每一列中的最小与最大值）。我们可以沿着 0 号轴来统计。想要实现沿着哪个轴来统计，只需要修改 axis 即可，代码如下：

```
import numpy as np

info = np.array([[3000, 4000, 20000], [2700, 5500, 25000], [2800, 3000, 15000]])

# 沿着0号轴统计，结果为[2700, 3000, 15000]
print(info.min(axis=0))

# 沿着0号轴统计，结果为[3000, 5500, 25000]
```

```
print(info.max(axis=0))
```

输出如下:

```
[ 2700  3000 15000]
[ 3000  5500 25000]
```

PS:当没有修改 **axis** 时，**axis** 的值默认为 **None** 。意思是在统计时会把 **ndarray** 对象中所有的元素都考虑在内。

2.1.4：随机数生成

简单随机数生成

NumPy 的 random 模块下提供了许多生成随机数的函数，如果对于随机数的概率分布没有什么要求，则通常可以使用 random_sample 、choice 、randint 等函数来实现生成随机数的功能。

random_sample

random_sample 用于生成区间为 [0, 1] 的随机数，需要填写的参数 size 表示生成的随机数的形状，比如 size=[2, 3] 那么则会生成一个 2 行 3 列的 ndarray ，并用随机值填充。示例代码如下：

```
import numpy as np

print(np.random.random_sample(size=[2, 3]))
```

输出可能如下：

```
[[0.32343809 0.38736262 0.42413616]
 [0.86190206 0.27183736 0.12824812]]
```

choice

如果想模拟像掷骰子、扔硬币等这种随机值是离散值，而且知道范围的可以使用 choice 实现。choice 的主要参数是 a 和 size 。 a 是个一维数组，代表你想从 a 中随机挑选；size 是随机数生成后的形状。假如模拟 5 次掷骰子，代码如下：

```
import numpy as np

'''
掷骰子时可能出现的点数为1, 2, 3, 4, 5, 6，所以a=[1,2,3,4,5,6]
模拟5此掷骰子所以size=5
结果可能为 [6, 4, 3, 1, 3]
'''
print(np.random.choice(a=[1, 2, 3, 4, 5, 6], size=5))
```

输出可能如下：

```
[6 4 3 1 3]
```

randint

randint 的功能和 choice 差不多，只不过 randint 只能生成整数，而 choice 生成的数与 a 有关，如果 a 中有浮点数，那么 choice 会有概率挑选到浮点数。

randint 的参数有 3 个，分别为 low ， high 和 size 。其中 low 表示随机数生成时能够生成的最小值， high 表示随机数生成时能够生成的最大值减 1 。也就是说 randint 生成的随机数的区间为 [low, high) 。假如模拟 5 次掷骰子，代码如下：

```
import numpy as np

'''
掷骰子时可能出现的点数为1, 2, 3, 4, 5, 6，所以low=1,high=7
模拟5此掷骰子所以size=5
结果可能为 [6, 4, 3, 1, 3]
'''
print(np.random.randint(low=1, high=7, size=5)
```

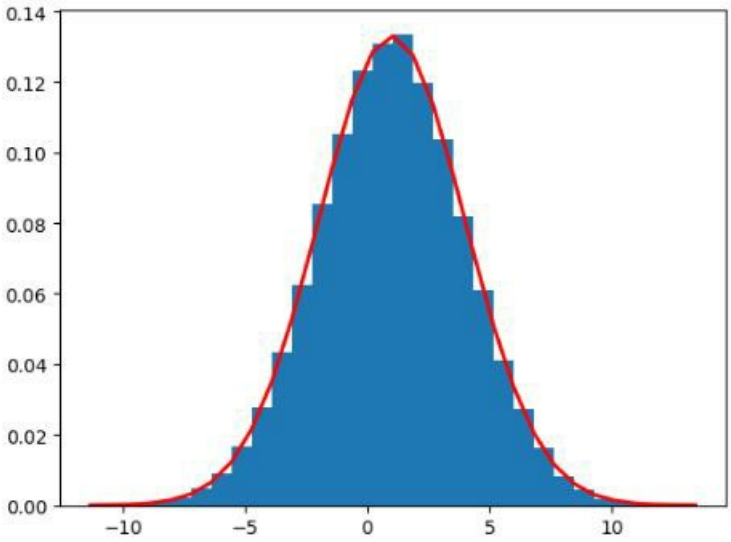
输出可能如下:

```
[6 4 3 1 3]
```

概率分布随机数生成

如果对于产生的随机数的概率分布有特别要求，NumPy 同样提供了从指定的概率分布中采样得到的随机值的接口。在这里主要介绍高斯分布。

高斯分布又称为正态分布，其分布图形如下：



上图中横轴为随机变量的值（在这里可以看成是产生的随机值），纵轴表示随机变量对应的概率（在这里可以看成是随机值被挑选到的概率）。

其实在日常生活中有很多现象或多或少都符合高斯分布。比如某个地方的高考分数，一般来说高考分数非常低和高考分数非常高的学生都比较少，而分数中规中矩的学生比较多，如果所统计的数据足够大，那么高考分数的概率分布也会和上图一样，中间高，两边低。

想要实现根据高斯分布来产生随机值，可以使用 normal 函数。示例代码如下：

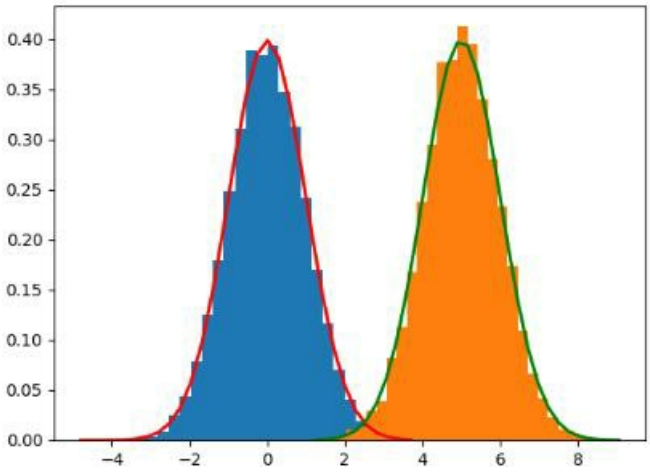
```
import numpy as np
```

```
'''
根据高斯分布生成5个随机数
结果可能为：[1.2315868, 0.45479902, 0.24923969, 0.42976352, -0.68786445]

从结果可以看出0.4左右得值出现的次数比较高，1和-0.7左右的值出现的次数比较低。
'''
print(np.random.normal(size=5)
```

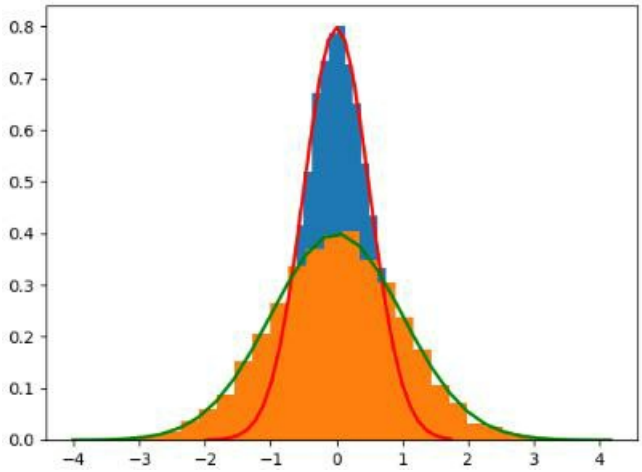
输出可能如下：

```
[1.2315868 0.45479902 0.24923969 0.42976352 -0.68786445]
```

其中 normal 函数除了 size 参数外，还有两个比较重要的参数就是 loc 和 scale ，它们分别代表高斯分布的均值和方差。 loc 影响的分布中概率最高的点的位置，假设 loc=2 ，那么分布中概率最高的点的位置就是 2 。下图体现了 loc 对分布的影响，其中蓝色f分布的 loc=0 ，红色分布的 loc=5 。



scale 影响的是分布图形的胖瘦，scale 越小，分布就越又高又瘦，scale 越大，分布就越又矮又胖。下图体现了 scale 对分布的影响，其中蓝色分布的scale=0.5 ，红色分布的 scale=1.0 。



所以，想要根据均值为 1 ，方差为 10 的高斯分布来生成 5 个随机值，代码如下：

```
import numpy as np

print(np.random.normal(loc=1, scale=10, size=5)
```

输出可能如下：

```
[1.52414964  7.08225325 13.26337651  4.29866004  9.89972241]
```

随机种子

前面说了这么多随机数生成的方法，那么随机数是怎样生成的呢？其实计算机产生的随机数是由随机种子根据一定的计算方法计算出来的数值。所以只要计算方法固定，随机种子固定，那么产生的随机数就不会变！

如果想要让每次生成的随机数不变，那么就需要设置随机种子（随机种子其实就是一个**0**到**4294967295**的整数）。设置随机种子很长简单，调用 seed 函数并设置随机种子即可，代码如下：

```
import numpy as np

# 设置随机种子为233
np.random.seed(seed=233)

data = [1, 2, 3, 4]

# 随机从data中挑选数字，结果为4
print(np.random.choice(data))

# 随机从data中挑选数字，结果为4
print(np.random.choice(data))
```

输出如下：

```
4
4
```

2.1.5：索引与切片

索引

ndarray 的索引其实和 python 的 list 的索引极为相似。元素的索引从 0 开始。代码如下：

```
import numpy as np

# a中有4个元素，那么这些元素的索引分别为0，1，2，3
a = np.array([2, 15, 3, 7])

# 打印第2个元素
# 索引1表示的是a中的第2个元素
# 结果为15
print(a[1])

# b是个2行3列的二维数组
b = np.array([[1, 2, 3], [4, 5, 6]])

# 打印b中的第1行
# 总共就2行，所以行的索引分别为0，1
# 结果为[1, 2, 3]
print(b[0])

# 打印b中的第2行第2列的元素
# 结果为5
print(b[1][1])
```

输出如下：

```
15
[1 2 3]
5
```

遍历

ndarray 的遍历方式与 python 的 list 的遍历方式也极为相似，示例代码如下：

```
import numpy as np

a = np.array([2, 15, 3, 7])

# 使用for循环将a中的元素取出来后打印
for element in a:
    print(element)

# 根据索引遍历a中的元素并打印
for idx in range(len(a)):
    print(a[idx])
```

```
# b是个2行3列的二维数组
b = np.array([[1, 2, 3], [4, 5, 6]])

# 将b展成一维数组后遍历并打印
for element in b.flat:
    print(element)

# 根据索引遍历b中的元素并打印
for i in range(len(b)):
    for j in range(len(b[0])):
        print(b[i][j])
```

输出如下：

```
2
15
3
7
2
15
3
7
1
2
3
4
5
6
1
2
3
4
5
6
```

切片

ndarray 的切片方式与 python 的 list 的遍历方式也极为相似，对切片不熟的同学也不用慌，套路很简单，就是用索引。

假设想要将下图中紫色部分切片出来，就需要确定行的范围和列的范围。由于紫色部分行的范围是 0 到 2，所以切片时行的索引范围是 0:3 (索引范围是左闭右开);又由于紫色部分列的范围也是 0 到 2，所以切片时列的索引范围也是 0:3 (索引范围是左闭右开)。最后把行和列的索引范围整合起来就是 [0:3, 0:3] (逗号左边是行的索引范围)。当然有时为了方便，0 可以省略，也就是 [:3, :3] 。

	0	1	2	3	4
0	1	10	11	20	21
1	2	9	12	19	22
2	3	8	13	18	23
3	4	7	14	17	24
4	5	6	15	16	25

切片示例代码如下：

```
import numpy as np

# a中有4个元素，那么这些元素的索引分别为0，1，2，3
a = np.array([2, 15, 3, 7])

'''
将索引从1开始到最后的所有元素切片出来并打印
结果为[15  3  7]
'''
print(a[1:])

'''
将从倒数第2个开始到最后的所有元素切片出来并打印
结果为[3  7]
'''
print(a[-2:])

'''
将所有元素倒序切片并打印
结果为[ 7  3 15  2]
'''
print(a[::-1])

# b是个2行3列的二维数组
b = np.array([[1, 2, 3], [4, 5, 6]])

'''
将第2行的第2列到第3列的所有元素切片并打印
结果为[[5 6]]
'''
print(b[1:, 1:3])

'''
将第2列到第3列的所有元素切片并打印
结果为[[2 3]
      [5 6]]
'''
print(b[:, 1:3])
```

输出如下：

```
[15 3 7]
[3 7]
[7 3 15 2]
[[5 6]]
[[2 3]
 [5 6]]
```

2.2.1：堆叠操作

stack

stack 的意思是堆叠的意思，所谓的堆叠就是将两个 ndarray 对象堆叠在一起组合成一个新的 ndarray 对象。根据堆叠的方向不同分为 hstack 以及 vstack 两种。

hstack

假如你是某公司的 HR ，需要记录公司员工的一些基本信息。可能你现在已经记录了如下信息：

工号	姓名	出生年月	联系电话
1	张三	1988.12	13323332333
2	李四	1987.2	15966666666
3	王五	1990.1	13777777777
4	周六	1996.4	13069699696

世界上没有不变的需求，你的老板让你现在记录一下公司所有员工的居住地址和户籍地址。此时你只好屁颠屁颠的记录这些附加信息。然后可能会有这样的结果：

居住地址	户籍地址
江苏省南京市禄口机场宿舍202	江西省南昌市红谷滩新区天月家园A座2201
江苏省南京市禄口机场宿舍203	湖南省株洲市天元区新天华府11栋303
江苏省南京市禄口机场宿舍204	四川省成都市武侯祠安置小区1栋701
江苏省南京市禄口机场宿舍205	浙江省杭州市西湖区兴天世家B座1204

接下来你需要把之前记录的信息和刚刚记录好的附加信息整合起来，变成这样：

工号	姓名	出生年月	联系电话	居住地址	户籍地址
1	张三	1988.12	13323332333	江苏省南京市禄口机场宿舍202	江西省南昌市红谷滩新区天月家园A座2201
2	李四	1987.2	15966666666	江苏省南京市禄口机场宿舍203	湖南省株洲市天元区新天华府11栋303
3	王五	1990.1	13777777777	江苏省南京市禄口机场宿舍204	四川省成都市武侯祠安置小区1栋701
4	周六	1996.4	13069699696	江苏省南京市禄口机场宿舍205	浙江省杭州市西湖区兴天世家B座1204

看得出来，你在整合的时候是将两个表格(二维数组)在水平方向上堆叠在一起组合起来，拼接成一个新的表格(二维数组)。像这种行为称之为 hstack (horizontal stack)。

NumPy 提供了实现 hstack 功能的函数叫 hstack ， hstack 的使用套路代码如下：

```
import numpy as np
```

```
a = np.array([[8, 8], [0, 0]])
b = np.array([[1, 8], [0, 4]])

'''
将a和b按元组中的顺序横向拼接
结果为：[[8, 8, 1, 8],
        [0, 0, 0, 4]]
'''
print(np.hstack((a,b)))

c = np.array([[1, 2], [3, 4]])

'''
将a,b,c按元组中的顺序横向拼接
结果为：[[8, 8, 1, 8, 1, 2],
        [0, 0, 0, 4, 3, 4]]
'''
print(np.hstack((a,b,c)))
```

输出如下：

```
[[8 8 1 8]
 [0 0 0 4]]
[[8 8 1 8 1 2]
 [0 0 0 4 3 4]]
```

vstack

你还是某公司的 HR ，你记录了公司员工的一些信息，如下：

工号	姓名	出生年月	联系电话
1	张三	1988.12	13323332333
2	李四	1987.2	15966666666
3	王五	1990.1	13777777777
4	周六	1996.4	13069699696

今天有两位新同事入职，你需要记录他们的信息，如下：

工号	姓名	出生年月	联系电话
5	刘七	1986.5	13323332331
6	胡八	1997.3	15966696669

然后你需要将新入职同事的信息和已经入职的员工信息整合在一起。

工号	姓名	出生年月	联系电话
1	张三	1988.12	13323332333
2	李四	1987.2	15966666666
3	王五	1990.1	13777777777
4	周六	1996.4	13069699696

5	刘七	1986.5	13323332331
6	胡八	1997.3	15966696669

在这种情况下，你在整合的时候是将两个表格(二维数组)在竖直方向上堆叠在一起组合起来，拼接成一个新的表格(二维数组)。像这种行为称之为 vstack (vertical stack)。

NumPy 提供了实现 vstack 功能的函数叫 vstack， vstack 的示例代码如下：

```
import numpy as np

a = np.array([[8, 8], [0, 0]])
b = np.array([[1, 8], [0, 4]])

'''
将a和b按元组中的顺序纵向拼接
结果为：[[8, 8]
        [0, 0]
        [1, 8]
        [0, 4]]
'''
print(np.vstack((a,b)))

c = np.array([[1, 2], [3, 4]])

'''
将a,b,c按元组中的顺序纵向拼接
结果为：[[8 8]
        [0 0]
        [1 8]
        [0 4]
        [1 2]
        [3 4]]
'''
print(np.vstack((a,b,c)))
```

输出如下：

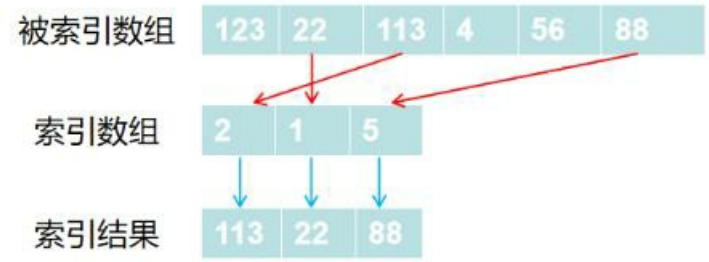
```
[[8 8]
 [0 0]
 [1 8]
 [0 4]]
[[8 8]
 [0 0]
 [1 8]
 [0 4]
 [1 2]
 [3 4]]
```

2.2.2：花式索引与布尔索引

花式索引

花式索引(Fancy Indexing)是 NumPy 用来描述使用整型数组（这里的数组，可以是 **NumPy** 的 **ndarray**，也可以是 **python** 的 **list**）作为索引。

Fancy Indexing 的意义是根据索引数组的值作为被索引数组(ndarray)的某个轴的下标来取值。对于使用一维整型数组作为索引来说，如果被索引数组(ndarray)是一维数组，那么索引的结果就是对应位置的元素；如果被索引数组(ndarray)是二维数组，那么就是对应下标的行。如下图所示：



示例代码如下：

```
import numpy as np

arr = np.array(['zero','one','two','three','four'])

'''
打印arr中索引为1和4的元素
结果为：['one', 'four']
'''
print(arr[[1,4]])

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

'''
打印arr中索引为1和0的行
```

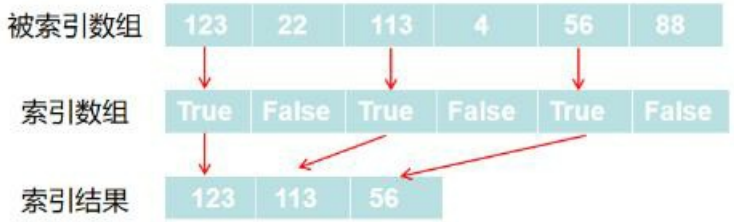
```
结果为：[[4, 5, 6],
        [1, 2, 3]]
'''
print(arr[[1, 0]])

'''
打印arr中第2行第1列与第3行第2列的元素
结果为：[4, 8]
'''
print(arr[[1, 2], [0, 1]])
```

输出如下：

```
['one' 'four']
[[4 5 6]
 [1 2 3]]
[4 8]
```

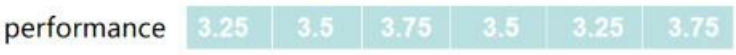
布尔索引

我们可以通过一个布尔数组来索引目标数组，以此找出与布尔数组中值为 True 的对应的目标数组中的数据，从而达到筛选出想要的数据的功能。如下图所示：(**PS**：需要注意的是，布尔数组的长度必须与被索引数组对应的轴的长度一致)



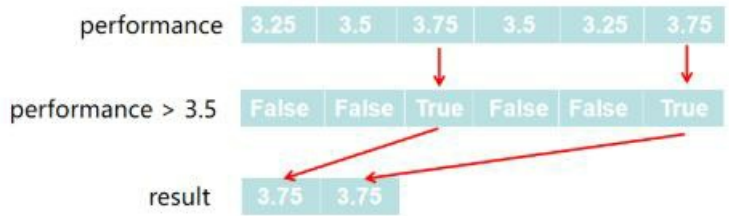
不过单纯的传入布尔数组进去有点蠢，有没有更加优雅的方式来使用布尔索引来达到筛选数据的效果呢？当然有！我们可以想办法根据我们的需求，构造出布尔数组，然后再通过布尔索引实现筛选数据的功能。

假设有公司员工绩效指数的数据如下(用一个一维的 ndarray 表示)，现在想要把绩效指数大于 3.5 的筛选出来进行股权激励。



那首先就要构造出布尔数组，构造布尔数组很简单，`performance > 3.5` 即可。此时会生成想要的布尔数组。

有了布尔数组就可以使用布尔索引来实现筛选数据的功能了。



示例代码如下：

```
import numpy as np

performance = np.array([3.25, 3.5, 3.75, 3.5, 3.25, 3.75])

'''
筛选出绩效高于3.5的数据
结果为:[3.75, 3.75]
'''
print(performance[performance > 3.5])

'''
筛选出绩效高于3.25并且低于4的数据
注意：&表示并且的意思，可以看成是and。&左右两边必须加上()
结果为:[3.5  3.75 3.5  3.75]
'''
print(performance[(performance > 3.25) & (performance < 4)])
```

输出如下：

```
[3.75 3.75]
[3.5  3.75 3.5  3.75]
```

2.2.3：广播机制

什么是广播

两个 `ndarray` 对象的相加、相减以及相乘都是对应元素之间的操作。代码如下：

```
import numpy as np

x = np.array([[2,2,3],[1,2,3]])
y = np.array([[1,1,3],[2,2,4]])

print(x*y)

'''
输出结果如下：
[[ 2 2 9]
[ 2 4 12]]
'''
```

当两个 `ndarray` 对象的形状并不相同的时候，我们可以通过扩展数组的方法来实现相加、相减、相乘等操作，这种机制叫做广播(`broadcasting`)。 比如，一个二维的 `ndarray` 对象减去列平均值，来对数组的每一列进行取均值化处理：

```
import numpy as np

# arr为4行3列的ndarray对象
arr = np.random.randn(4,3)
# arr_mean为有3个元素的一维ndarray对象
arr_mean = arr.mean(axis=0)
# 对arr的每一列进行
demeaned = arr - arr_mean
```

很明显上面代码中的 `arr` 和 `arr_mean` 维度并不相同，但是它们可以进行相减操作，这就是通过广播机制来实现的。

广播的原则

如果两个数组的后缘维度(trailing dimension，即从末尾开始算起的维度)的轴长度相符，或其中的一方的长度为 `1` ，则认为它们是广播兼容的。广播会在缺失或长度为 `1` 的维度上进行。

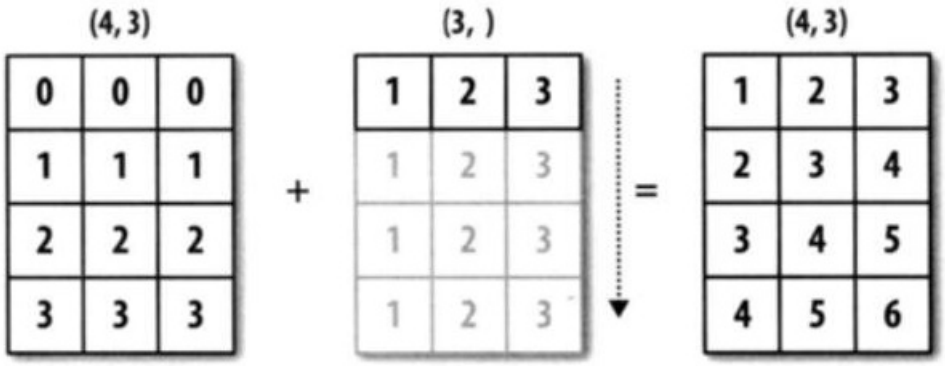
这句话是理解广播的核心。广播主要发生在两种情况，一种是两个数组的维数不相等，但是它们的后缘维度的轴长相符，另外一种是有一方的长度为 `1` 。

我们来看一个例子：

```
import numpy as np

arr1 = np.array([[0,  0,  0],[1,  1,  1],[2,  2,  2],  [3,  3,  3]])
arr2 = np.array([1,  2,  3])
```

```
arr_sum = arr1 + arr2
print(arr_sum)

'''
输入结果如下：
[[1 2 3]
[2 3 4]
[3 4 5]
[4 5 6]]
'''
```

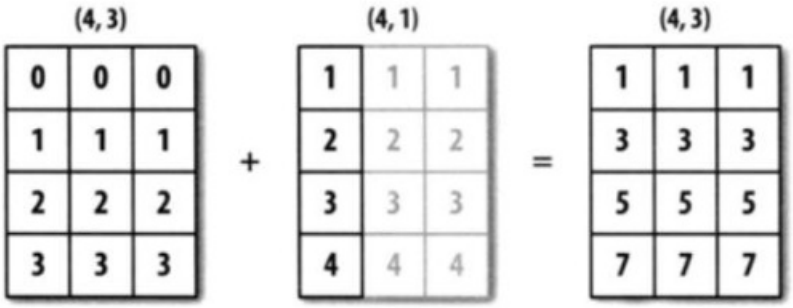
`arr1` 的 `shape` 为 `(4,3)`， `arr2` 的 `shape` 为 `(3,)`。可以说前者是二维的，而后者是一维的。但是它们的后缘维度相等， `arr1` 的第二维长度为 `3`，和 `arr2` 的维度相同。 `arr1` 和 `arr2` 的 `shape` 并不一样，但是它们可以执行相加操作，这就是通过广播完成的，在这个例子当中是将 `arr2` 沿着 `0` 轴进行扩展。



我们再看一个例子：

```
import numpy as np

arr1 = np.array([[0, 0, 0],[1, 1, 1],[2, 2, 2], [3, 3, 3]])  #arr1.shape = (4,3)
arr2 = np.array([[1],[2],[3],[4]])    #arr2.shape = (4, 1)

arr_sum = arr1 + arr2
print(arr_sum)

'''
输出结果如下：
[[1 1 1]
 [3 3 3]
 [5 5 5]
 [7 7 7]]
'''
```

`arr1` 的 `shape` 为 `(4,3)`， `arr2` 的 `shape` 为 `(4,1)`，它们都是二维的，但是第二个数组在 `1` 轴上的长度为 `1`，所以，可以在 `1` 轴上面进行广播。


(4, 3)
0 0 0
1 1 1
2 2 2
3 3 3
+
(4, 1)
1 1 1
2 2 2
3 3 3
4 4 4
=
(4, 3)
1 1 1
3 3 3
5 5 5
7 7 7

2.2.4：线性代数

numpy的线性代数

线性代数（如矩阵乘法、矩阵分解、行列式以及其他方阵数学等）是任何数组库的重要组成部分，一般我们使用 `*` 对两个二维数组相乘得到的是一个元素级的积，而不是一个矩阵点积。因此 `numpy` 提供了线性代数函数库 `linalg`，该库包含了线性代数所需的所有功能。

常用的 `numpy.linalg` 函数：

函数	说明
dot	矩阵乘法
vdot	两个向量的点积
det	计算矩阵的行列式
inv	计算方阵的逆
svd	计算奇异值分解(SVD)
solve	解线性方程组Ax=b，A是一个方阵
matmul	两个数组的矩阵积

常用函数

dot() 该函数返回俩个数组的点积。对于二维向量，效果等于矩阵乘法。对于一维数组，它是向量的内积。对于 `N` 维数组，它是 `a` 的最后一个轴上的和与 `b` 的倒数第二个轴的乘积。

```
 a=np.array([[1,2],[3,4]])

 a1=np.array([[5,6],[7,8]])

 np.dot(a,a1)
'''
输出：array([[19, 22],
       [43, 50]])
'''
```

det() 该函数用于计算输入矩阵的行列式。

```
a = np.array([[14, 1], [6, 2]])

a=linalg.det(a)

print(a)
'''
输出：21.999999999999996
'''
```

inv() 该函数用于计算方阵的逆矩阵。逆矩阵的定义维如果两个方阵A、B，使得AB=BA=E，则A称为可逆矩阵，B为A的逆矩阵，E为单位矩阵。

```
a=np.array([[1,2],[3,4]])

b=linalg.inv(a)

print(np.dot(a,b))
'''
输出：array([[1.0000000e+00, 0.0000000e+00],
       [8.8817842e-16, 1.0000000e+00]])
'''
```

solve() 该函数用于计算线性方程的解。假设有如下方程组： `3x+2y=7 x+4y=14` 写成矩阵的形式： `[[3,2] [1,4]] * [[x],[y]] = [[7],[14]]` 解如上方程组代码如下：

```
a=np.array([[3,2], [1,4]])

b=np.array([[7],[14]])

linalg.solve(a,b)
'''
输出:array([[0. ],
       [3.5]])
最后解出x=0，y=3.5
'''
```

matmul()

函数返回两个数组的矩阵乘积。如果参数中有一维数组，则通过在其维度上附加 `1` 来提升为矩阵，并在乘法之后去除。

```
a=[[3,4],[5,6]]

b=[[7,8],[9,10]]

np.matmul(a,b)
'''
输出：array([[ 57,  64],
       [ 89, 100]])
'''

b=[7,8]

np.matmul(a,b)
'''
输出：array([53, 83])
'''
```

svd() 奇异值分解是一种矩阵分解的方法，该函数用来求解 `SVD` 。

```
a=[[0,1],[1,1],[1,0]]

linalg.svd(a)
```

```
'''
输出:(array([[-4.08248290e-01,  7.07106781e-01,  5.77350269e-01],
       [-8.16496581e-01,  2.64811510e-17, -5.77350269e-01],
       [-4.08248290e-01, -7.07106781e-01,  5.77350269e-01]]), array([1.73205081, 1.        ]), array([[-0.707
10678, -0.70710678],
       [-0.70710678,  0.70710678]]))
'''
```

2.2.5：排序和条件筛选

numpy中的快速排序

`numpy` 中的排序相对 Python 的更加高效，默认情况下 `np.sort` 的排序算法是快速排序，也可以选择归并排序和堆排序。

类型	速度	最坏情况	工作空间	稳定性
快速排序	`1`	`O(n^2)`	`0`	否
归并排序	`2`	`O(n*log(n))`	`~n/2`	是
堆排序	`3`	`O(n*log(n))`	`0`	否

- `np.sort()` 函数返回排序后的数组副本，只能是升序。

```
a=np.array([5,9,1,15,3,10])

np.sort(a) #升序排序
'''
输出：array([ 1,  3,  5,  9, 10, 15])
'''
```

- `np.argsort()` 函数返回排序后数组值从小到大的索引，可以通过这些索引值创建有序的数组。

```
a=np.array([4,5,9,1,3])

b=np.argsort(a) #对a使用argsort函数

print(b)
'''
输出：array([3, 4, 0, 1, 2], dtype=int64)
第一个数是最小的值的索引，第二个数是第二小的值的索引,以此类推
'''

b1=[]
for i in b: #循环获取索引对应的值
    b1.append(a[i])
print(b1)
'''
输出:[1, 3, 4, 5, 9]
'''
```

- 沿行或列进行排序，通过 `axis` 参数实现对数组的行、列进行排序，这种处理是将行或列当作独立的数组，任何行或列的值之间的关系将会丢失。

```
a=np.array([[8,1,5,9],[5,4,9,6],[7,1,5,3]])

np.sort(a,axis=1) #沿行排序
'''
输出：array([[1, 5, 8, 9],
```

```
            [4, 5, 6, 9],
            [1, 3, 5, 7]])
'''

np.sort(a,axis=0) #沿列排序
'''
输出:array([[5, 1, 5, 3],
           [7, 1, 5, 6],
           [8, 4, 9, 9]])
'''
```

- `np.partition()` 函数为给定一个数，对数组进行分区，区间中的元素任意排序。

```
a=np.array([8,9,2,3,1,6,4])

np.partition(a,5) #比5小的在左边，比5大的在右边
'''
输出:array([1, 3, 2, 4, 6, 8, 9])
'''
```

其他排序函数：

函数	描述
`msort()`	数组按第一个轴排序，返回排序后的数组副本
`sort_complex()`	对复数按先实部后虚部的顺序进行排序
`argpartition()`	通过关键字指定算法沿指定轴进行分区

where函数

- `np.where()` 函数返回输入数组中满足给定条件的元素的索引，可以利用该函数进行 条件筛选。

```
a=np.array([19,5,16,22,17])

np.where(a>15) #应用where函数
'''
输出：(array([0, 2, 3, 4], dtype=int64),)
'''

a[np.where(a>15)] #获取满足条件索引的元素

'''
输出:array([19, 16, 22, 17])
'''
```

2.1.6：结构化数组

结构化数组

结构化数组其实就是 `ndarrays` ，其数据类型是由组成一系列命名字段的简单数据类型组成的，在定义结构化数据时需要指定数据类型。

构建结构化数组的数据类型有多种制定方式，如字典、元组列表：

```
a=np.array([(b'Rex', 9, 81.), (b'Fido', 7, 77.)],
      dtype=[('name', '<S10'), ('age', '<i4'), ('score', '<f4')]) #通过字典定义数据类型

a=np.array([(b'Rex', 9, 81.), (b'Fido', 7, 77.)],dtype={'names':("name","age","score"),"formats":("<S10","<i4"
,"<f4")}) #通过元组定义数据类型

print(a)
'''
两种方式结果都是一样的
输出:array([(b'Rex', 9, 81.), (b'Fido', 7, 77.)],
      dtype=[('name', '<S10'), ('age', '<i4'), ('score', '<f4')])
'''
```

上面案例中的 `S10` 表示“长度不超过 `10` 的字符串”， `i4` 表示“ `4` 个字节整型”， `f4` 表示“ `4` 字节浮点型”。

`numpy` 的数据类型如下：

数据类型	描述	示例
`b`	字节型	`np.dtype('b')`
`i`	有符号整型	`np.dtype('i4')==np.int32`
`u`	无符号整型	`np.dtype('u')==np.uint8`
`f`	浮点型	`np.dtype('f8')==np.int64`
`c`	复数浮点型	`np.dtype('c16')==np.complex128`
`S` 、 `a`	字符串	`np.dtype('SS')`
`U`	`Unicode` 字符串	`np.dtype('U')==np.str\_`
`V`	原生数据	`np.dtype('V')==np.void`

通过文件构造结构化数组

`numpy` 通过 `loadtxt()` 函数读取文件内容，假设有以下文件内容，需要读取文件构造结构化数组：

name	**age**	**score**
amy	11	70
kasa	10	80

baron	9	66

```
a=np.loadtxt("data.txt",skiprows=1,dtype=[("name","S10"),("age","int"),("score","float")]) #读取文件并设置数据类型，其中skiprows为跳过第一行

print(a)
'''
输出：[(b'amy', 11, 70.) (b'kasa',  10, 80.) (b'baron', 9, 66.)]
'''
```

结构化数组常规操作

结构化数组的方便之处在于，你可以通过索引或名称查看相应的值，并且可以进行快速的数据处理。

```
a=np.array([(b'Rex', 10, 81.), (b'Fido', 7, 77.),(b'kasr', 9, 55.)],
      dtype=[('name', 'S10'), ('age', '<i4'), ('score', '<f4')]) #构造结构化数组

print(a[0]) #查看第一行
'''
输出:(b'Rex', 10, 81.)
'''

print(a["age"]>=10) #查看是否有大于等于10岁的
'''
输出:[True False False]
'''

print(a["score"]<60) #查看是否有小于60的
'''
输出:[False False  True]
'''

print(a[a["age"]>=10]) #查看大于等于10岁的信息
'''
输出:[(b'Rex', 10, 81.)]
'''

print(a[a["score"]<60]["name"]) #查看小于60的人的姓名
'''
输出:[b'kasr']
'''
```

注意：尽管这里列举的模式对于简单的操作非常有用，但是这些操作场景也可以用 `pandas` 的 `DataFrame` 来实现，并且 `DataFrame` 更加强大。

Chapter3 Pandas

Pandas 是基于 NumPy 的一种工具，该工具是为了解决数据分析任务而创建的。Pandas 纳入了大量库和一些标准的数据模型，提供了高效地操作大型数据集所需的工具。Pandas 提供了大量能使我们快速便捷地处理数据的函数和方法。你很快就会发现，它是使 Python 成为强大而高效的数据分析环境的重要因素之一。

Pandas 相关实训已在 `educoder` 平台上提供，若感兴趣可以输入链接进行体验。

链接：https://www.educoder.net/paths/302

3.1.1：Series对象

创建Series对象

创建 `Pandas` 的 `Series` 对象的方法： `pd.Series(data,index=index)`

其中， `index` 是一个可选参数，默认为 `np.arange(n)` ， `data` 参数支持多种数据类型。

Series是通用的Numpy数组

`Series` 对象和一维 `Numpy` 数组基本可以等价交换，但两者间的本质差异其实是索引： `NumPy` 数组通过隐式定义的整数索引获取数值，而 `Pandas` 的 `Series` 对象用一种显式定义的索引与数值关联。 显式索引的定义让 `Series` 对象拥有了更强的能力。例如，索引不再仅仅是整数，还可以是任意想要的类型。

```
In: data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
Out: a 0.25
     b 0.50
     c 0.75
     d 1.00
     dtype: float64
In: data["b"]
Out: 0.5
```

Series是特殊的字典

你可以把 `Pandas` 的 `Series` 对象看成一种特殊的 `Python` 字典。字典是一种将任意键映射到一组任意值的数据结构，而 `Series` 对象其实是一种将类型键映射到一组类型值的数据结构。我们可以直接用 `Python` 的字典创建一个 `Series` 对象，让 `Series` 对象与字典的类比更加清晰。

```
In: population_dict = {'California': 38332521,'Texas': 26448193, 'New York': 19651127, 'Florida': 19552860, '
Illinois': 12882135}
In: population = pd.Series(population_dict)
In: population
Out: California 38332521
     Florida    19552860
     Illinois   12882135
     New York   19651127
     Texas      26448193
     dtype: int64
```

用字典创建 `Series` 对象时，其索引默认按照顺序排列。典型的字典数值获取方式仍然有效，而且还支持数组形式的切片操作等。

```
In: population['California']
Out: 38332521
In: population['California':'Illinois']
Out: California 38332521
```

```
Florida    19552860
Illinois   12882135
dtype: int64
```

3.1.2：DateFrame对象

创建DataFrame对象

`Pandas` 的 `DataFrame` 对象可以通过许多方式创建，这里举几个常用的例子。

- 通过数组创建
  > pd.DateFrame(array, index=list0, columns=list1) #list表示一个列表
- 通过单个Series对象创建
  > pd.DateFrame(Series,columns=list)
- 通过字典列表创建
  > data = [{'a': i, 'b': 2 * i} for i in range(3)] pd.DataFrame(data)

DataFrame是通用的NumPy数组

如果将 `Series` 类比为带灵活索引的一维数组，那么 `DataFrame` 就可以看作是一种既有灵活的行索引，又有灵活列名的二维数组，你也可以把 `DataFrame` 看成是有序排列的若干 `Series` 对象。这里的“排列”指的是它们拥有共同的索引 `index` 。

```
# 创建Series对象
In: area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297, 'Florida': 170312, 'Illinois': 14
9995}
In: population_dict = {'California': 38332521,'Texas': 26448193, 'New York': 19651127, 'Florida': 19552860, '
Illinois': 12882135}
In: population = pd.Series(population_dict)
In: area = pd.Series(area_dict)
# 创建DataFrame对象
In: states = pd.DataFrame({'population': population, 'area': area})
In: states
Out:          area      population
 California  423967     38332521
 Florida     170312     19552860
 Illinois    149995     12882135
 New York    141297     19651127
 Texas       695662     26448193
```

和 `Series` 对象一样， `DataFrame` 也有一个 `index` 属性可以获取索引标签。另外， `DataFrame` 还有一个 `columns` 属性，是存放列标签的 `Index` 对象。

```
In: states.columns
Out: Index(['area', 'population'], dtype='object')
```

DataFrame是特殊的字典

与 `Series` 类似，我们也可以把 `DataFrame` 看成一种特殊的字典。字典是一个键映射一个值，而 `DataFrame` 是一列映射一个 `Series` 的数据。例如，通过 `area` 的列属性可以返回包含面积数据的 `Series` 对象。

```
In: states['area']
Out: California  423967
     Florida     170312
     Illinois    149995
     New York    141297
     Texas       695662
     Name: area, dtype: int64
```

注意：在 **`NumPy`** 的二维数组里， **`data[0]`** 返回第一行；而在 **`DataFrame`** 中， **`data['列名']`** 返回与列名相匹配的那一列。

3.1.3：Index对象

将Index看作不可变数组

`Index` 对象得许多操作都像数组。可以通过标准 `Python` 的取值方法获取数值，也可以通过切片获取数值。

```
In: ind[1]
Out: 3
In: ind[::2]
Out: Int64Index([2, 5, 11], dtype='int64')
```

`Index` 对象还有许多与 `NumPy` 数组相似的属性。

```
In: print(ind.size, ind.shape, ind.ndim, ind.dtype)
Out: 5 (5,) 1 int64
```

`Index` 对象与 `NumPy` 数组之间的不同在于， `Index` 对象的索引是不可变的，也就是说不能通过通常的方式进行调整。

> ind[1] = 0 # 这种操作是不可取的，会报错

将Index看作有序集合

`Pandas` 对象被设计用于实现许多操作，如连接（ `join` ）数据集，其中会涉及许多集合操作。 `Index` 对象遵循 `Python` 标准库的集合（ `set` ）数据结构的许多习惯用法，包括并集、交集、差集等。

```
In: indA = pd.Index([1, 3, 5, 7, 9])
In: indB = pd.Index([2, 3, 5, 7, 11])
In: indA & indB      # 交集
Out: Int64Index([3, 5, 7], dtype='int64')
In: indA | indB      # 并集
Out: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
In: indA ^ indB      # 异或
Out: Int64Index([1, 2, 9, 11], dtype='int64')
```

这些操作还可以通过调用对象方法来实现，例如 `indA.intersection(indB)` 等。

3.1.4：Series数据选择

Series数据选择方法

与 `Python` 中的字典一样， `Series` 对象提供了键值对的映射。

```
In: import pandas as pd
In: data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
In: data["b"]
Out: 0.5
```

还可以用 `Python` 字典的表达式和方法来检测键/索引和值，也可以像字典一样来修改 `Series` 对象的值。

```
In: "a" in data
Out: True
In: data.keys()
Out: Index(['a', 'b', 'c', 'd'], dtype='object')
In: list(data.items())
Out: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
In: data["b"] = 0.05    # 也可以通过此方法来扩展Series
In: data
Out:  a 0.25
      b 0.05
      c 0.75
      d 1.00
      dtype: float64
```

`Series` 对象的可变性是一个非常方便的特性： `Pandas` 在底层已经为可能发生的内存布局和数据复制自动决策，用户不需要担心这些问题。

将Series看作一堆数组

`Series` 对象还具备和 `Numpy` 数组一样的数组数据选择功能，包括索引、掩码、花哨索引等操作，具体示例如下所示：

- 将显示索引作为切片 *(*注意：显示索引切片结果包含最后一个索引，也就是能取到*“c”*的值*)*

```
In: data['a':'c']
Out: a 0.25
       b 0.50
       c 0.75
       dtype: float64
```

- 将隐式整数索引作为切片 *(*注意：隐式索引切片结果不包含最后一个索引*)*

```
In: data[0:2]
Out: a 0.25
       b 0.50
```

```
       dtype: float64
```

- 掩码

```
In: data[(data > 0.3) & (data < 0.8)]
Out: b 0.50
       c 0.75
       dtype: float64
```

- 花哨的索引

```
In: data[["a","e"]]
Out: a 0.25
       e 1.25
       dtype: float64
```

索引器：loc和iloc

这些切片和取值操作非常混乱，假如 `Series` 对象索引序列为整数时， `data[2]` 不会取第三行，而是取索引序列为 `2` 的那一行，也就是说会优先使用显示索引，而 `data[1:3]` 这样的切片操作会优先使用隐式索引。

```
In: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
In: data[1]
Out: "a"
In: data[0:2]
Out: 1 a
     3 b
     dtype: object
```

正是应为这些整数索引很容易造成混淆，所以 `Pandas` 提供了一些索引器**(indexer)**属性来作为取值的方法。它们不是 `Series` 对象的函数方法，而是暴露切片接口的属性。

- `loc` 属性：表示取值和切片都是显示的

```
In: data.loc[1]
Out: "a"
In: data.loc[1:3]
Out: 1 a
       3 b
       dtype: object
```

- `iloc` 属性：表示取值和切片都是 `Python` 形式的隐式索引

```
In: data.iloc[1]
Out: "b"
In: data.iloc[1:3]
Out: 3 b
     5 c
     dtype: object
```

`Python`代码的设计原则之一式“显示优于隐式”。使用 `loc` 和 `iloc` 可以让代码更容易维护，可读性更高。特别是在处理整数索引的对象时，我强烈推荐使用这两种索引器。它们既可以让代码阅读和理解起来更容易，也能避免因误用索引或者切片而产生的小 `bug` 。

3.1.5：DataFrame数据选择方法

将DataFrame看作字典

`DataFrame` 可以看作一个由若干 `Series` 对象构成的字典，可以通过对列名进行字典形式的取值获取数据。

```
In: area = pd.Series({'California': 423967, 'Texas': 695662, 'New York': 141297, 'Florida': 170312, 'Illinois'
: 149995})
In: pop = pd.Series({'California': 38332521, 'Texas': 26448193, 'New York': 19651127, 'Florida': 19552860, 'I
llinois': 12882135})
In: data = pd.DataFrame({'area':area, 'pop':pop})
In: data["area"]    # data.area 这种属性形式也可以获取到相同的结果
Out: California 423967
     Florida 170312
     Illinois 149995
     New York 141297
     Texas 695662
     Name: area, dtype: int64
```

虽然属性形式的数据选择方法很方便，但是它并不是通用的。如果列名不是纯字符串，或者列名与 `DataFrame` 的方法同名，那么就不能用属性索引。例如， `DataFrame` 有一个 `pop()` 方法，如果用 `data.pop` 就不会获取 `'pop'` 列，而是显示为方法。

```
In: data.pop is data['pop']
Out: False
```

所以，尽量避免用属性形式选择的列直接赋值，即避免 `data.pop=z` 这种方式赋值。

和 `Series` 对象一样，可以用字典形式的语法调整对象，例如增加一列数据。

```
In: data["density"] = data['pop']/data['area']
In: data
Out:              area     pop     density
      California  423967 38332521 90.413926
      Florida     170312 19552860 114.806121
      Illinois    149995 12882135 85.883763
      New York    141297 19651127 139.076746
      Texas       695662 26448193 38.018740
```

将DataFrame看作二维数组

`DataFrame` 可以看成是一个增强版的二维数组，许多数组操作方式都可以用在 `DataFrame` 对象上，例如，用 `value` 属性按行查看数组数据，对 `DataFrame` 进行转置等等。

```
In: data.value
Out:
array(
```

```
[[ 4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
 [ 1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
 [ 1.49995000e+05, 1.28821350e+07, 8.58837628e+01],
 [ 1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
 [ 6.95662000e+05, 2.64481930e+07, 3.80187404e+01]])
In: data.T
Out:
```

	California	Florida	Illinois	New York	Texas
area	4.239670e+05	1.703120e+05	1.499950e+05	1.412970e+05	6.956620e+05
pop	3.833252e+07	1.955286e+07	1.288214e+07	1.965113e+07	2.644819e+07
density	9.041393e+01	1.148061e+02	8.588376e+01	1.390767e+02	3.801874e+01

将 `DataFrame` 看作数组时，我们可以是使用单个行索引获取一行数据。

```
In: data.values[0]    #取一行数据
Out: array([ 4.23967000e+05, 3.83325210e+07, 9.04139261e+01])
```

而获取一列数据就需要向 `DataFrame` 传递单个列索引，与字典形式的操作相同，都是用 `data["列名"]` 获取。

因此，在进行数组形式的取值时，我们就需要用另一种方法——前面介绍过的 `Pandas` 索引器 `loc、iloc` 和 `ix` 了。通过 `iloc` 索引器，我们就可以像对待 `NumPy` 数组一样索引 `Pandas` 的底层数组（ `Python` 的隐式索引）， `DataFrame` 的行列标签会自动保留在结果中。

```
In: data.iloc[1:3, :2]    # 隐式
Out:       area    pop
 Florida  170312 19552860
 Illinois 149995 12882135
In: data.loc[:'Illinois', :'pop']    #显式
Out:         area     pop
 California 423967 38332521
 Florida    170312 19552860
 Illinois   149995 12882135
```

使用 `ix` 索引器可以实现一种混合效果， `Series` 中也可以使用这种索引器，但是作用不明显。需要注意的是， `ix` 索引器处理整数索引时和 `Series` 对象一样容易让人混淆。

```
In: data.ix[:3, :"pop"]
Out:         area    pop
 California 423967 38332521
 Florida    170312 19552860
 Illinois   149995 12882135
```

任何用于处理 `NumPy` 形式数据的方法都可以用于这些索引器，例如在 `loc` 索引器中结合使用掩码与花哨索引方法。

```
In: data.loc[data.density > 100, ['pop', 'density']]
Out:         pop     density
 Florida  19552860 114.806121
```

```
  New York 19651127 139.076746
```

其他取值方法

还有一些取值方法和前面介绍过的方法不太一样，但是在实际应用中非常实用。

```
data["列名"]
data["A列":"B列"]
data[0:3]   # 取第一到第四行
data[data.density > 100]  #取density列值大于100的行
```

3.1.6：数值运算方法

通用函数：保留索引

因为 `Pandas` 是建立在 `NumPy` 基础之上的，所以 `NumPy` 的通用函数同样适用于 `Pandas` 的 `Series` 和 `DataFrame` 对象。

```
import numpy as np
import pandas as pd

rng = np.random.RandomState(42)  #创建随机数种子
ser = pd.Series(rng.randint(0,10,4))
df = pd.DataFrame(rng.randint(0,10,(3,4)), columns=['A','B','C','D'])
# 对Series对象使用Numpy通用函数，生成的结果是另一个保留索引的Pands对象
print(np.exp(ser))
Out: 0  403.428793
     1  20.085537
     2  1096.633158
     3  54.598150
     dtype: float64
# 对DataFrame使用Numpy通用函数
print(np.sin(df*np.pi/4))
Out:
          A             B             C             D
 0 -1.000000  7.071068e-01  1.000000 -1.000000e+00
 1 -0.707107  1.224647e-16  0.707107 -7.071068e-01
 2 -0.707107  1.000000e+00 -0.707107  1.224647e-16
```

通用函数：索引对齐

Series索引对齐

假如你要整合两个数据源的数据，其中一个是美国面积最大的三个州的面积数据，另一个是美国人口最多的三个州的人口数据：

```
# 面积
area=pd.Series({'Alaska':1723337,'Texas':695662,'California':423967},name='area')
# 人口
population=pd.Series({'California':38332521,'Texas':26448193,'New York': 19651127}, name='population'})
```

人口除以面积的结果：

```
print(population/area)
Out:
 Alaska         NaN
 California     90.413926
 New York       NaN
 Texas          38.018740
 dtype: float64
```

对于缺失的数据，`Pandas`会用`NaN`填充，表示空值。这是`Pandas`表示缺失值的方法。这种索引对齐方式是通过`Python`内置的集合运算规则实现的，任何缺失值默认都用`NaN`填充。

DataFrame索引对齐

在计算两个`DataFrame`时，类似的索引对齐规则也同样会出现在共同列中：

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)), columns=list('AB'))
"""
A:
    A  B
 0  1  11
 1  5  1
"""

B = pd.DataFrame(rng.randint(0, 10, (3, 3)), columns=list('BAC'))

"""
B:
    B A C
 0  4 0 9
 1  5 8 0
 2  9 2 6
"""
print(A + B)
Out::   A        B        C
 0     1.0      15.0     NaN
 1     13.0     6.0      NaN
 2     NaN      NaN      NaN
```

从上面的例子可以发现，两个对象的行列索引可以是不同顺序的，结果的索引会自动按顺序排列。
在`Series`中，我们可以通过运算符方法的`fill_value`参数自定义缺失值；这里我们将用`A`中所有值的均值来填充缺失值。

```
fill = A.stack().mean()  # stack()能将二维数组压缩成多个一维数组
print(A.add(B,fill_value=fill))
Out:
    A     B     C
 0 1.0  15.0  13.5       #NaN值都变成了均值
 1 13.0  6.0  4.5
 2 6.5  13.5  10.5
```

下表中列举了与`Python`运算符相对应的`Pandas`对象方法。

Python运算符	**Pandas**方法
+	add()
-	sub()、substract()
*	mul()、multiply()
/	truediv()、div()、divide()
//	floordiv()
%	mod()

**	pow()

通用函数：DataFrame与Series的运算

`DataFrame` 和 `Series` 的运算规则与 `Numpy` 中二维数组与一维数组的运算规则是一样的。来看一个常见运算，让一个二维数组减去自身的一行数据。

```
A = rng.randint(10, size=(3, 4))
A - A[0]
Out:
array([[ 0,  0, 0, 0],
       [-1, -2, 2, 4],
       [ 3, -7, 1, 4]])     # 根据Numpy的广播规则，默认是按行运算的
```

在 `Pandas` 里默认也是按行运算的，如果想按列计算，那么就需要利用前面介绍过的运算符方法，通过设置 `axis(轴)` 实现。

```
df = pd.DataFrame(A, columns=list('QRST'))
print(df - df.iloc[0])
Out:
    Q  R  S  T
 0  0  0  0  0
 1 -1 -2  2  4
 2  3 -7  1  4

print(df.subtract(df['R'],axis=0))
Out:
    Q  R  S  T
 0 -5  0 -6 -4
 1 -4  0 -2  2
 2  5  0  2  7
```

`DataFrame/Series` 的运算与前面介绍的运算一样，结果的索引都会自动对齐。

3.1.7：数值运算与缺失值处理

选择处理缺失值的方法

一般情况下可以分为两种：一种方法是通过一个覆盖全局的掩码表示缺失值，另一种方法是用一个标签值（`sentinel value`）表示缺失值。

- 掩码方法中掩码可能是一个与原数组维度相同的完整布尔类型数组，也可能是用一个比特（`0` 或 `1`）表示有缺失值的局部状态。
- 标签方法中，标签值可能是具体的数据（例如用 `-9999` 表示缺失的整数），也可能是些极少出现的形式。

Pandas缺失值

综合考虑各种方法的优缺点，`Pandas` 最终选择用标签方法表示缺失值，包括两种 `Python` 原有的缺失值：浮点数据类型的 `NaN` 值，以及 `Python` 的 `None` 对象。

- **`None`**：`Python` 对象类型的缺失值 `Pandas` 可以使用的第一种缺失值标签是 `None`，它是一个 `Python` 单体对象，由于 `None` 是一个 `Python` 对象，所以不能作为任何 `NumPy / Pandas` 数组类型的缺失值，只能用于 `'object'` 数组类型（即由 `Python` 对象构成的数组）

```
np.array([1, None, 3, 4])
Out: array([1, None, 3, 4], dtype=object)
```

- **`NaN`**：数值类型的缺失值 另一种缺失值的标签是 `NaN`（全称 `Not a Number`），是一种按照 `IEEE` 浮点数标准设计、在任何系统中都兼容的特殊浮点数：

```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
Out:  dtype('float64')
```

注意：`NumPy` 会为这个数组选择一个原生浮点类型，这意味着和之前的 `object` 类型数组不同，这个数组会被编译成 `C` 代码从而实现快速操作。你可以把 `NaN` 看作是一个数据类病毒——它会将与它接触过的数据同化。无论和 **`NaN`** 进行何种操作，最终结果都是 **`NaN`**：

```
1 + np.nan
0 * np.nan  #这两个的结果都为nan
```

虽然这些累计操作的结果定义是合理的（即不会抛出异常），但是并非总是有效的：

```
vals2 = np.array([1, np.nan, 3, 4])
vals2.sum(), vals2.min(), vals2.max()
Out:(nan, nan, nan)
```

`NumPy` 也提供了一些特殊的累计函数，它们可以忽略缺失值的影响：

```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
Out: (8.0, 1.0, 4.0)
```

谨记， *NaN* 是一种特殊的浮点数，不是整数、字符串以及其他数据类型。

- **Pandas** 中 **NaN** 与 **None** 的差异 虽然 `NaN` 与 `None` 各有各的用处，但是 `Pandas` 把它们看成是可以等价交换的：

```
pd.Series([1, np.nan, 2, None])
Out:
 0 1.0
 1 NaN
 2 2.0
 3 NaN
 dtype: float64
```

`Pandas` 会将没有标签值的数据类型自动转换为 `NA` 。例如我们将整形数组中的一个值设置为 `np.nan` 时，这个值就会强制转换成浮点数缺失值 `NA` ，下表表示 `Pandas` 对不同类型缺失值的转换规则：

类型	缺失值转换规则	**NA**标签值
floating 浮点型	无变化	np.nan
object 对象类型	无变化	np.nan或None
integer 整数类型	强制转换为 float64	np.nan
boolean 布尔类型	强制转换为 object	np.nan或None

发现缺失值

`Pandas` 有两种方法可以发现缺失值： `isnull()` 和 `notnull()` ，这俩个中方法皆可用于 `Series` 和 `DataFrame` 。每种方法都返回布尔类型的掩码数据。

```
data=pd.Series([1,np.nan,'hello',None])
data.isnull()
Out:
 0 False
 1 True
 2 False
 3 True
 dtype: bool
```

布尔类型掩码数组可以直接作为 `Series` 或 `DataFrame` 的索引使用。

```
data[data.notnull()]
Out:
 0 1
 2 hello
 dtype: object
```

处理缺失值

- dropna()删除缺失值 作用在 `Series` 对象上时，它的作用和 `data[data.notnull()]` 一样，而在 `DataFrame` 上使用它们时需要设置一些参数：

```
df = pd.DataFrame([[1,      np.nan,  2],
                   [2,      3,       5],
                   [np.nan, 4,       6]])
# 如果不传任何参数时，dropna会删除有缺失值的所有行
df.dropna()
Out:
     0   1  2
 1 2.0 3.0 5
# 传入axis=1(或者axis='columns')时会删除所有包含缺失值的列
df.dropna(axis=1)
Out:
    2
 0 2
 1 5
 2 6
```

但是这么做也会把非缺失值一并剔除，因为可能有时候只需要剔除全部是缺失值的行或列，或者绝大多数是缺失值的行或列。这些需求可以通过设置 `how` 或 `thresh` 参数来满足，它们可以设置剔除行或列缺失值的数量阈值。

```
df[3] = np.nan
Out:
     0   1  2  3
0 1.0 NaN 2 NaN
1 2.0 3.0 5 NaN
2 NaN 4.0 6 NaN
# 删除值全部为缺失值的列
df.dropna(axis=1,how="all")
Out:
     0   1  2
 0 1.0 NaN 2
 1 2.0 3.0 5
 2 NaN 4.0 6
#通过 thresh 参数设置行或列中非缺失值的最小数量
df.dropna(axis='rows', thresh=3) #非缺失值至少有3个
Out:
     0   1  2  3
 1 2.0 3.0 5 NaN
```

- fillna()填充缺失值 有时候你可能并不想移除缺失值，而是想把它们替换成有效的数值。虽然你可以通过 `isnull()` 方法建立掩码来填充缺失值，但是 `Pandas` 为此专门提供了一个 `fillna()` 方法，它将返回填充了缺失值后的数组副本。

```
data=pd.Series([1, np.nan, 2, None, 3],index=list('abcd')
data.fillna(0)  # 将缺失值填充为0
Out:
 a 1.0
 b 0.0
 c 2.0
 d 0.0
```

```
 e 3.0
 dtype: float64
```

可以用缺失值前面的有效值来从前往后填充（forward-fill），也可以用缺失值后面的有效值来从后往前填充（back-fill）：

```
data.fillna(method="ffill")
Out:
 a 1.0
 b 1.0
 c 2.0
 d 2.0
 e 3.0
 dtype: float64
data.fillna(method='bfill')
Out:
 a 1.0
 b 2.0
 c 2.0
 d 3.0
 e 3.0
 dtype: float64
```

DataFrame 的操作方法与 Series 类似，只是在填充时需要设置坐标轴参数 axis。

```
df.fillna(method='ffill', axis=1)  # bfill同样适用
Out:
     0   1   2   3
 0 1.0 1.0 2.0 2.0
 1 2.0 3.0 5.0 5.0
 2 NaN 4.0 6.0 6.0
```

3.2.1：多级索引的取值与切片

创建多级索引

1. 通过 `MultiIndex` 构建多级索引

```
index = [('California', 2000), ('California', 2010), ('New York', 2000), ('New York', 2010), ('Texas', 2
000), ('Texas', 2010)]
populations = [33871648, 37253956, 18976457, 19378102, 20851820, 25145561]
pop = pd.Series(populations, index=index)
# 1.基于元组创建
index1 = pd.MultiIndex.from_tuples(index)
index1
Out:
MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]], codes=[[0, 0, 1, 1, 2, 2], [0, 1,
0, 1, 0, 1]])
```

`MultiIndex` 里面有一个 `levels` 属性表示索引的等级——这样做可以将州名和年份作为每个数据点的不同标签。如果将前面创建的 `pop` 的索引重置 （`reindex`） 为 `MultiIndex`，就会看到层级索引。其中前两列表示 `Series` 的多级索引值，第三列式数据。

```
pop1 = pop.reindex(index1)
pop1
Out:
 California 2000 33871648
            2010 37253956
 New York   2000 18976457
            2010 19378102
 Texas      2000 20851820
            2010 25145561
 dtype: int64
```

查询 `2010` 年的数据。

```
pop[:, 2010]  # 得到的是一个单索引数组
Out:
 California 37253956
 New York   19378102
 Texas      25145561
 dtype: int64
```

以上的例子都是 `Series` 创建多级行索引，而每个 `DataFrame` 的行与列都是对称的，也就是说既然有多级行索引，那么同样可以有多级列索引。只需要在创建 `DataFrame` 时将 `columns` 的参数传入一个 `MultiIndex` 。

1. 通过二维索引数组创建多级索引 `Series` 或 `DataFrame` 创建多级索引最直接的办法就是将 `index` 参数设置为至少二维的索引数组。

```
df = pd.DataFrame(np.random.rand(4, 2), index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]], columns=['data1', 'data2'
])
```

```
df
Out:
      data1    data2
 a 1 0.554233 0.356072
   2 0.925244 0.219474
 b 1 0.441759 0.610054
   2 0.171495 0.886688
```

`MultiIndex` 的创建工作将在后台完成。同理，如果你把将元组作为键的字典传递给 `Pandas`， `Pandas` 也会默认转换为 `MultiIndex` 。

1. 显示的创建多级索引 你可以用 `pd.MultiIndex` 中的类方法更加灵活地构建多级索引。

```
# 由不同等级的若干简单数组组成的列表来构建
pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
# 包含多个索引值的元组构成的列表创建
pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
# 由两个索引的笛卡尔积（Cartesian product）创建
pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
# 三种创建方法的结果都一致
Out:
MultiIndex(levels=[['a', 'b'], [1, 2]],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

在创建 *`Series`* 或 *`DataFrame`* 时，可以将这些对象作为 *`index`* 参数，或者通过 *`reindex`* 方法更新 *`Series`* 或 *`DataFrame`* 的索引。

1. 多级索引的等级名称 你可以在前面任何一个 `MultiIndex` 构造器中通过 `names` 参数设置等级名称，也可以在创建之后通过索引的 `names` 属性来修改名称。

   ```
   pop.index.names = ['state', 'year']
   Out:
   state      year
   California 2000 33871648
              2010 37253956
   New York   2000 18976457
              2010 19378102
   Texas      2000 20851820
              2010 25145561
   dtype: int64
   ```

2. 多级列索引 每个 `DataFrame` 的行与列都是对称的，也就是说既然有多级行索引，那么同样可以有多级列索引。

3. 多级索引与 `DataFrame` 的相互转换 `unstack()` 方法可以快速将一个多级索引的 `Series` 转化为普通索引的 `DataFrame` 。

   ```
   pop.unstack()   # stack()可以将DataFrame转换为多级索引
   Out:
               2000     2010
   California 33871648 37253956
   New York   18976457 19378102
   Texas      20851820 25145561
   ```

多级索引的取值与切片

1. `Series` 多级索引 以各州历年人口数量创建的多级索引 `Series` 为例：

```
pop
Out:
state      year
California 2000 33871648
           2010 37253956
New York   2000 18976457
           2010 19378102
Texas      2000 20851820
           2010 25145561
dtype: int64
```

获取单个元素：

```
pop['California',2000]
Out:
33871648
```

`MultiIndex` 也支持局部取值 （`partial indexing`），即只取索引的某一个层级。假如只取最高级的索引，获得的结果是一个新的 `Series`，未被选中的低层索引值会被保留：

```
pop['California']
Out:
year
2000 33871648
2010 37253956
dtype: int640
```

1. `DataFrame` 多级索引

```
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]], names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']], names=['subject', 'type'
])
# 模拟数据
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37
# 创建一个包含多级列索引的DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
Out:
subject     Bob      Guido     Sue
type        HR  Temp  HR  Temp  HR  Temp
year  visit
2013  1     31.0 38.7 32.0 36.7 35.0 37.2
2 44. 0     37.7 50.0 35.0 29.0 36.7
2014  1     30.0 37.4 39.0 37.8 61.0 36.9
2 47. 0     37.8 48.0 37.3 51.0 36.5
```

由于 `DataFrame` 的基本索引是列索引，因此 `Series` 中多级索引的用法到了 `DataFrame` 中就应用在列上了。

```
health_data['Guido', 'HR']
```

```
Out:
year visit
2013 1         32.0
     2         50.0
2014 1         39.0
     2         48.0
Name: (Guido, HR), dtype: float64
```

`loc` 、 `iloc` 、 `ix` 三个索引器都可以使用，虽然这些索引器将多维数据当作二维数据处理，但是在 `loc` 和 `iloc` 中可以传递多个层级的索引元组。

```
health_data.loc[:, ('Bob', 'HR')]
Out:
year visit
2013  1 31.0
      2 44.0
2014  1 30.0
      2 47.0
Name: (Bob, HR), dtype: float64
```

这种索引元组的用法不是很方便，如果在元组中使用切片还会导致语法错误。

```
health_data.loc[(:, 1), (:, 'HR')]
# 这种切片方式会报错
```

`Pandas` 的 `IndexSlice` 对象可以解决上述问题。

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
Out:
subject     Bob  Guido Sue
type        HR   HR    HR
year visit
2013 1      31.0 32.0  35.0
2014 1      30.0 39.0  61.0
```

3.2.2：多级索引的数据转换与累计方法

多级索引行列转换

使用多级索引的关键是掌握有效数据转换的方法，`Pandas`提供了许多操作，可以让数据在内容保持不变的同时，按照需要进行行列转换。上一关我们用`stack()`和`unstack()`演示过简单的行列转换，但其实还有许多合理控制层级行列索引的方法，让我们来一探究竟。

1. 有序和无序的索引

如果`MultiIndex`不是有序的索引，那么大多数切片操作都会失败，如下例：

```
# 首先创建一个不按字典顺序排列的多级索引Series
index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
data = pd.Series(np.random.rand(6), index=index)
data.index.names = ['char', 'int']
data
Out:
char int
 a    1    0.003001
        2    0.164974
 c    1    0.741650
        2    0.569264
 b    1    0.001693
        2    0.526226
 dtype:    float64
# 如果对该多级索引使用局部切片，就会报错
try:
        data['a':'b']
except KeyError as e:
        print(type(e))
        print(e)
Out:
<class 'KeyError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

问题是出在`MultiIndex`无序排列上，局部切片和许多其他相似的操作都要求`MultiIndex`的各级索引是有序的。为此，`Pandas`提供了许多便捷的操作完成排序，如`sort_index()`和`sortlevel()`方法。

```
data = data.sort_index()
data["a":"b"]
Out:
char int
a    1     0.003001
        2     0.164974
b    1     0.001693
        2     0.526226
 dtype: float64
```

1. 索引stack与unstack

上一节提过，我们可以将一个多级索引数据集转换成简单的二维形式，可以通过 `level` 参数设置转换的索引层级。

```
pop
Out:
state      year
California 2000     33871648
              2010     37253956
New York   2000     18976457
              2010     19378102
Texas      2000     20851820
              2010     25145561
dtype: int64
# level=0
pop.unstack(level=0)
Out:
state California New York  Texas
year
2000  33871648   18976457  20851820
2010  37253956   19378102  2514556
# level=1
pop.unstack(level=1)
Out:
 year        2000      2010
 state
 California 33871648 37253956
 New York   18976457 19378102
 Texas      20851820 25145561
```

`unstack()` 是 `stack()` 的逆操作，同时使用这两种方法 （`pop.unstack().stack()`） 让数据保持不变。

1. 索引的设置与重置

层级数据维度转换的另一种方法是行列标签转换，可以通过 `reset_index` 方法实现。也可以用数据的 `name` 属性为列设置名称：

```
pop_flat = pop.reset_index(name='population')
pop_flat
Out:
         state  year population
 0 California 2000 33871648
 1 California 2010 37253956
 2 New York   2000 18976457
 3 New York   2010 19378102
 4 Texas      2000 20851820
 5 Texas      2010 25145561
```

在解决实际问题的时候，可以使用 `DataFrame` 的 `set_index` 方法将类似这样的原始输入数据的列直接转换成 `MultiIndex` ，返回结果就会是一个带多级索引的 `DataFrame` 。

```
pop_flat.set_index(['state', 'year'])
Out:
                    population
 state      year
 California 2000  33871648
              2010  37253956
```

```
New York   2000  18976457
             2010  19378102
Texas      2000  20851820
             2010  25145561
```

多级索引的数据累计方法

`Pandas` 有一些自带的数据累计方法，比如 `mean()` 、 `sum()` 和 `max()` 。而对于层级索引数据，可以设置参数 `level` 实现对数据子集的累计操作。以体检数据为例：

```
health_data
Out:
 subject     Bob      Guido     Sue
 type         HR Temp  HR  Temp  HR  Temp
 year visit
 2013 1     31.0 38.7 32.0 36.7 35.0 37.2
        2     44.0 37.7 50.0 35.0 29.0 36.7
 2014 1     30.0 37.4 39.0 37.8 61.0 36.9
        2     47.0 37.8 48.0 37.3 51.0 36.5
```

如果你需要计算每一年各项指标的平均值，那么可以将参数 `level` 设置为索引 `year` ：

```
data_mean = health_data.mean(level='year')
data_mean
Out:
 subject Bob       Guido      Sue
 type     HR Temp  HR   Temp  HR   Temp
 year
 2013    37.5 38.2 41.0 35.85 32.0 36.95
 2014    38.5 37.6 43.5 37.55 56.0 36.70
```

如果再设置 `axis` 参数，就可以对列索引进行类似的累计操作了：

```
data_mean.mean(axis=1, level='type')
Out:
type       HR     Temp
 year
 2013 36.833333 37.000000
 2014 46.000000 37.283333
```

3.2.3：Concat与Append操作

相关知识

在 `Numpy` 中，我们介绍过可以用 `np.concatenate` 、 `np.stack` 、 `np.vstack` 和 `np.hstack` 实现合并功能。 `Pandas` 中有一个 `pd.concat()` 函数与 `concatenate` 语法类似，但是配置参数更多，功能也更强大，主要参数如下。

参数名	说明
objs	参与连接的对象，必要参数
axis	指定轴，默认为0
join	inner或者outer，默认为outer，指明其他轴的索引按哪种方式进行合并,inner表示取交集，outer表示取并集
join_axes	指明用于其他n-1条轴的索引，不执行并集/交集运算
keys	与连接对象有关的值，用于形成连接轴向上的层次化索引。可以是任意值的列表或数组
levels	指定用作层次化索引各级别上的索引
names	用于创建分层级别的名称，如果设置了keys和levels
verify_integrity	检查结果对象新轴上的重复情况，如果发现则引发异常。默认False允许重复
ignore_index	不保留连接轴上的索引，产生一组新索引

`pd.concat()` 可以简单地合并一维的 `Series` 或 `DataFrame` 对象。

```
# Series合并
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1,ser2])
Out:
 1 A
 2 B
 3 C
 4 D
 5 E
 6 F
 dtype: object

# DataFrame合并，将concat的axis参数设置为1即可横向合并
df1 = pd.DataFrame([["A1","B1"],["A2","B2"]],index=[1,2],columns=["A","B"])
df2 = pd.DataFrame([["A3","B3"],["A4","B4"]],index=[3,4],columns=["A","B"])
pd.concat([df1,df2])
Out:
    A  B
 1 A1 B1
 2 A2 B2
 3 A3 B3
 4 A4 B4
```

合并时索引的处理

`np.concatenate` 与 `pd.concat` 最主要的差异之一就是 `Pandas` 在合并时会保留索引，即使索引是重复的！

```
df3 = pd.DataFrame([["A1","B1"],["A2","B2"]],index=[1,2],columns=["A","B"])
df4 = pd.DataFrame([["A3","B3"],["A4","B4"]],index=[1,2],columns=["A","B"])
pd.concat([df3,df4])
Out:
    A  B
 1 A1 B1
 2 A2 B2
 1 A3 B3
 2 A4 B4
```

1. 如果你想要检测 `pd.concat()` 合并的结果中是否出现了重复的索引，可以设置 `verify_integrity` 参数。将参数设置为 `True` ，合并时若有索引重复就会触发异常。

   ```
   try:
    pd.concat([df3, df4], verify_integrity=True)
   except ValueError as e:
    print("ValueError:", e)
   Out:
   ValueError: Indexes have overlapping values: [0, 1]
   ```

2. 有时索引无关紧要，那么合并时就可以忽略它们，可以通过设置 `ignore_index` 参数为 `True` 来实现。

   ```
   pd.concat([df3,df4],ignore_index=True)
   Out:
      A B
   0 A0 B0
   1 A1 B1
   2 A2 B2
   3 A3 B3
   ```

3. 另一种处理索引重复的方法是通过 `keys` 参数为数据源设置多级索引标签，这样结果数据就会带上多级索引。

   ```
   pd.concat([df3, df4], keys=['x', 'y'])
   Out:
        A B
   x 0 A0 B0
     1 A1 B1
   y 0 A2 B2
     1 A3 B3
   ```

join和join_axes参数

前面介绍的简单示例都有一个共同特点，那就是合并的 `DataFrame` 都是同样的列名。而在实际工作中，需要合并的数据往往带有不同的列名，而 `pd.concat` 提供了一些参数来解决这类合并问题。

```
df5 = pd.DataFrame([["A1","B1","C1"],["A2","B2","C2"]],index=[1,2],columns=["A","B","C"])
df6 = pd.DataFrame([["B3","C3","D3"],["B4","C4","D4"]],index=[3,4],columns=["B","C","D"])
pd.concat([df5,df6])
```

```
Out:

    A  B  C  D
1 A1  B1 C1 NaN
2 A2  B2 C2 NaN
3 NaN B3 C3 D3
4 NaN B4 C4 D4
```

可以看到，结果中出现了缺失值，如果不想出现缺失值，可以使用 `join` 和 `join_axes` 参数。

```
pd.concat([df5,df6],join="inner") # 合并取交集
Out:
    B C
1 B1 C1
2 B2 C2
3 B3 C3
4 B4 C4

# join_axes的参数需为一个列表索引对象
pd.concat([df5,df6],join_axes=[pd.Index(["B","C"])])
Out:
    B C
1 B1 C1
2 B2 C2
3 B3 C3
4 B4 C4
```

append()方法

因为直接进行数组合并的需求非常普遍，所以 `Series` 和 `DataFrame` 对象都支持 `append` 方法，让你通过最少的代码实现合并功能。例如， `df1.append(df2)` 效果与 `pd.concat([df1,df2])` 一样。但是它和 `Python` 中的 `append` 不一样，每次使用 `Pandas` 中的 `append()` 都需要重新创建索引和数据缓存。

3.2.4：合并与连接

相关知识

`merge()` 可根据一个或者多个键将不同的 `DataFrame` 连接在一起，类似于 `SQL` 数据库中的合并操作。

参数名	说明
left	拼接左侧DataFrame对象
right	拼接右侧DataFrame对象
on	列(名称)连接，必须在左和右DataFrame对象中存在(找到)。
left_on	左侧DataFrame中的列或索引级别用作键，可以是列名、索引级名，也可以是长度等于DataFrame长度的数组。
right_on	右侧DataFrame中的列或索引级别用作键。可以是列名，索引级名称，也可以是长度等于DataFrame长度的数组。
left_index	如果为True，则使用左侧DataFrame中的索引(行标签)作为其连接键。
right_index	与left_index相似
how	它可以等于left, right, outer, inner. 默认inner。inner是取交集，outer取并集。
sort	按照字典顺序通过连接键对结果DataFrame进行排序。
suffixes	用于追加到重叠列名的末尾。例如，左右两个DataFrame都有data列，则结果中就会出现data_x和data_y.
copy	默认总是复制

数据连接的类型

- 一对一的连接

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'], 'group': ['Accounting', 'Engineering', 'Engin
eering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'], 'hire_date': [2004, 2008, 2012, 2014]})
df3 = pd.merge(df1,df2)
df3
```

输出：

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

- 多对一的连接

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'], 'supervisor': ['Carly', 'Guido', 'Steve']})
pd.merge(df3,df4)
```

输出：

df3

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

df4

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

pd.merge(df3, df4)

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

- 多对多连接

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting', 'Engineering', 'Engineering', 'HR', 'HR'], 'skills': ['math', 'spreadsheets', 'coding', 'linux', 'spreadsheets', 'organization']})
pd.merge(df1,df5)
```

输出：

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df5

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

```
pd.merge(df1, df5)
```

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6	Sue	HR	spreadsheets
7	Sue	HR	organization

merge()的主要参数

1. `on` 可以是列名字符串或者一个包含多列名称的列表

```
pd.merge(df1, df2, on='employee')
```

输出：

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

```
pd.merge(df1, df2, on='employee')
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

这个参数只能在两个 *DataFrame* 有共同列名的时候才可以使用。

2. `left_on` 与 `right_on` 参数

有时你也需要合并两个列名不同的数据集，例如前面的员工信息表中有一个字段不是 `employee` 而是 `name` 。在这种情况下，就可以用 `left_on` 和 `right_on` 参数来指定列名。

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'], 'salary': [70000, 80000, 120000, 90000]})
dfx = pd.merge(df1,df3,left_on="employee",right_on="name")
```

输出：

```
df1                          df2
  employee        group          employee  hire_date
0      Bob   Accounting        0     Lisa       2004
1     Jake  Engineering        1      Bob       2008
2     Lisa  Engineering        2     Jake       2012
3      Sue           HR        3      Sue       2014

pd.merge(df1, df2, on='employee')
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

如果出现重复列，但是列名不同时，可以使用 `drop` 方法将这列去掉

```
dfx.drop("name",axis=1)
```

输出：

```
  employee        group  salary
0      Bob   Accounting   70000
1     Jake  Engineering   80000
2     Lisa  Engineering  120000
3      Sue           HR   90000
```

3. left_index与right_index参数 用于合并索引

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
pd.merge(df1a,df2a,left_index=True,right_index=True)
```

输出：

```
df1a                         df2a
               group                      hire_date
employee                    employee
Bob         Accounting      Lisa               2004
Jake       Engineering      Bob                2008
Lisa       Engineering      Jake               2012
Sue                HR       Sue               2014

pd.merge(df1a, df2a, left_index=True, right_index=True)
               group  hire_date
employee
Lisa       Engineering       2004
Bob         Accounting       2008
Jake       Engineering       2012
Sue                HR       2014
```

用 `join()` 方法也可以实现该功能：

```
df1a.join(df2a)
```

输出：

```
df1a.join(df2a)
                group  hire_date
employee
Bob         Accounting       2008
Jake       Engineering       2012
Lisa       Engineering       2004
Sue                HR       2014
```

如果想将索引与列混合使用，那么可以通过结合 `left_index` 与 `right_on` ，或者结合 `left_on` 与 `right_index` 来实现。

```
pd.merge(df1a, df3, left_index=True, right_on='name')
```

输出：

```
pd.merge(df1a, df3, left_index=True, right_on='name')
          group  name  salary
0   Accounting   Bob   70000
1  Engineering  Jake   80000
2  Engineering  Lisa  120000
3           HR   Sue   90000
```

---

4. how参数

`how` 参数默认情况下是 `inner` ，也就是取交集。 `how` 参数支持的数据连接方式还有 `outer` 、 `left` 和 `right` 。 `outer` 表示外连接，取并集。

```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'], 'food': ['fish', 'beans', 'bread']}, columns=['name',
'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'], 'drink': ['wine', 'beer']}, columns=['name', 'drink'])
pd.merge(df6, df7, how='outer')
```

输出：

```
df6                 df7                 pd.merge(df6, df7, how='outer')
     name   food         name drink          name   food drink
0   Peter   fish     0    Mary  wine     0   Peter   fish   NaN
1    Paul  beans     1  Joseph  beer     1    Paul  beans   NaN
2    Mary  bread                       2    Mary  bread  wine
                                       3  Joseph    NaN  beer
```

左连接和右连接返回的结果分别只包含左列和右列。

```
pd.merge(df6, df7, how='left')
```

输出：

```
df6                df7                pd.merge(df6, df7, how='left')
     name   food         name drink         name   food drink
0  Peter   fish     0    Mary  wine    0   Peter   fish   NaN
1   Paul  beans     1  Joseph  beer    1    Paul  beans   NaN
2   Mary  bread                      2    Mary  bread  wine
```

5. suffixes参数

如果输出结果中有两个重复的列名，因此 `pd.merge()` 函数会自动为它们增加后缀 `_x` 或 `_y`，当然也可以通过 `suffixes` 参数自定义后缀名。

```
df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'], 'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'], 'rank': [3, 1, 4, 2]})
pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
```

输出：

```
df8                df9
    name  rank         name  rank
0   Bob     1      0   Bob     3
1  Jake     2      1  Jake     1
2  Lisa     3      2  Lisa     4
3   Sue     4      3   Sue     2
```

```
pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
   name  rank_L  rank_R
0   Bob       1       3
1  Jake       2       1
2  Lisa       3       4
3   Sue       4       2
```

`suffixes` 参数同样适用于任何连接方式，即使有三个及三个以上的重复列名时也同样适用。

3.2.5：分组聚合

分组

通常我们将数据分成多个集合的操作称之为分组，`Pandas` 中使用 `groupby()` 函数来实现分组操作。

单列和多列分组

对分组后的子集进行数值运算时，不是数值的列会自动过滤

```
import pandas as pd
data = {'A': [1, 2, 2, 3, 2, 4],
        'B': [2014, 2015, 2014, 2014, 2015, 2017],
        'C': ["a", "b", "c", "d", "e", "f"],
        'D': [0.5, 0.9, 2.1, 1.5, 0.5, 0.1]
        }
df = pd.DataFrame(data)
df.groupby("B")   #单列分组   返回的是一个groupby对象
df.groupby(["B","C"])    #多列分组
```

`Series` 系列分组

选取数据帧中的一列作为 `index` 进行分组：

```
df["A"].groupby(df["B"])   #df的 A 列根据 B 进行分组
```

通过数据类型或者字典分组

数据类型分组：

```
df.groupby(df.dtypes,axis=1)   # axis=1表示按列分组，以数据类型为列名
```

传入字典分组:

```
dic = {"A": "number", "B": "number", "C": "str", "D": "number"}
df.groupby(dic, axis=1)   #按列分组，列名是字典的值
```

获取单个分组

使用 `get_group()` 方法可以选择一个组。

```
df.groupby("A").get_group(2)
```

输出：

```
   A     B  C    D
1  2  2015  b  0.9
2  2  2014  c  2.1
4  2  2015  e  0.5
```

对分组进行迭代

`GroupBy` 对象支持迭代，可以产生一组二元元组（由分组名和数据块组成）。

```
for name,data in df.groupby("A"):
        print(name)
        print(data)
```

输出：

```
1
   A     B  C    D
0  1  2014  a  0.5
2
   A     B  C    D
1  2  2015  b  0.9
2  2  2014  c  2.1
4  2  2015  e  0.5
3
   A     B  C    D
3  3  2014  d  1.5
4
   A     B  C    D
5  4  2017  f  0.1
```

聚合

聚合函数为每个组返回单个聚合值。当创建了 `groupby` 对象，就可以对分组数据执行多个聚合操作。比较常用的是通过聚合函数或等效的 `agg` 方法聚合。常用的聚合函数如下表：

函数名	说明
count	分组中非空值的数量
sum	非空值的和
mean	非空值的平均值
median	非空值的中位数
std、var	无偏标准差和方差
min、max	非空值的最小和最大值
prod	非空值的积
first、last	第一个和最后一个非空值

应用单个聚合函数

对分组后的子集进行数值运算时，不是数值的列会自动过滤

```
import pandas as pd
import numpy as np
data = {'A': [1, 2, 2, 3, 2, 4],
        'B': [2014, 2015, 2014, 2014, 2015, 2017],
        'C': ["a", "b", "c", "d", "e", "f"],
        'D': [0.5, 0.9, 2.1, 1.5, 0.5, 0.1]
        }
df = pd.DataFrame(data)
df.groupby("B").sum()       #对分组进行求和
```

输出：

```
      A    D
B
2014  6    4.1
2015  4    1.4
2017  4    0.1
```

```
df.groupby("B").describe()
```

输出：

	level								value							
	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%	max
year																
2014	3.0	2.0	1.0	1.0	1.5	2.0	2.5	3.0	3.0	1.366667	0.808290	0.5	1.0	1.5	1.8	2.1
2015	2.0	2.0	0.0	2.0	2.0	2.0	2.0	2.0	2.0	0.700000	0.282843	0.5	0.6	0.7	0.8	0.9
2017	1.0	4.0	NaN	4.0	4.0	4.0	4.0	4.0	1.0	0.100000	NaN	0.1	0.1	0.1	0.1	0.1

应用多个聚合函数

```
df.groupby("B").agg([np.sum,np.mean,np.std])
```

输出：

```
      A                   D
      sum    mean    std    sum     mean        std
B
2014    6     2      1.0    4.1    1.366667    0.808290
2015    4     2      0.0    1.4    0.700000    0.282843
2017    4     4      NaN    0.1    0.100000    NaN
```

自定义函数传入**agg()**中

```
def result(df):
    return df.max() - df.min()
df.groupby("B").agg(result)  #求每一组最大值与最小值的差
```

输出：

```
        A    D
B
2014    2    1.6
2015    0    0.4
2017    0    0.0
```

对不同的列使用不同的聚合函数

```
mapping = {"A":np.sum,"D":np.mean}
df.groupby("B").agg(mapping)
```

输出：

```
        A    D
B
2014    6    1.366667
2015    4    0.700000
2017    4    0.100000
```

3.2.6：创建透视表和交叉表

透视表

透视表是各种电子表格程序和其他数据分析软件中一种常见的数据汇总工具。它根据一个或多个键对数据进行聚合，并根据行和列上得分组建将数据分配到各个矩形区域中。在 `pandas` 中，可以通过 `pivot_table` 函数创建透视表。

`pivot_talbe` 函数的参数：

```
DataFrame.pivot_table(self, values=None, index=None, columns=None,ggfunc='mean', fill_value=None, .margins=False,dropna=True, margins_name='All')
```

参数名	说明
values	待聚合的列的名称。默认聚合所有数值列
index	用于分组的列名或其他分组键，出现在结果透视表的行
columns	用于分组的列名或其他分组键，出现在结果透视表的列
aggfunc	聚合函数或函数列表，默认为 `mean` ，可以是任何对groupby有效的函数
fill_value	用于替换结果表中的缺失值
dropna	boolean值，默认为True，是否删除空值
margins	boolean值，当需要计算每一组的总数时可以设为True
margins_name	string，默认为‘ALL’，当参数margins为True时，ALL行和列的名字

示例：

```
data = {'A': [1, 2, 2, 3, 2, 4],
        'B': [2014, 2015, 2014, 2014, 2015, 2017],
        'C': ["a", "b", "c", "d", "e", "f"],
        'D': [0.5, 0.9, 2.1, 1.5, 0.5, 0.1]
        }
df = pd.DataFrame(data)
df.pivot_table(index=["B"], columns=["C"], values=["A"], aggfunc=sum, margins=True)
```

输出：

```
        A
C       a    b    c    d    e    f All
B
2014  1.0  NaN  2.0  3.0  NaN  NaN   6
2015  NaN  2.0  NaN  NaN  2.0  NaN   4
2017  NaN  NaN  NaN  NaN  NaN  4.0   4
All   1.0  2.0  2.0  3.0  2.0  4.0  14
```

交叉表

交叉表是一种用于计算分组频率的特殊透视表。通常使用 `crosstab` 函数来创建交叉表。

`crosstab` 的参数

```
pd.crosstab(index,columns,values=None,rownames=None
,colnames=None,aggfunc=None,margins=False,dropna=True,normalize=False)
```

其中 `rownames` 可以设置行名， `colnames` 可以设置列名，而且前两个参数可以是数组、 `Series` 或数组列表。

示例：

```
data = {'A': [1, 2, 2, 3, 2, 4],
        'B': [2014, 2015, 2014, 2014, 2015, 2017],
        'C': ["a", "b", "c", "d", "e", "f"],
        'D': [0.5, 0.9, 2.1, 1.5, 0.5, 0.1]
        }
df = pd.DataFrame(data)
pd.crosstab(index=[df["B"],df["A"]], columns=df["C"], values=df["A"], aggfunc=sum, margins=True)
```

输出：

```
C         a    b    c    d    e    f  All
B    A
2014 1  1.0  NaN  NaN  NaN  NaN  NaN    1
     2  NaN  NaN  2.0  NaN  NaN  NaN    2
     3  NaN  NaN  NaN  3.0  NaN  NaN    3
2015 2  NaN  2.0  NaN  NaN  2.0  NaN    4
2017 4  NaN  NaN  NaN  NaN  NaN  4.0    4
All     1.0  2.0  2.0  3.0  2.0  4.0   14
```

3.2.7：字符串操作方法

字符串方法

如果你对 `Python` 字符串方法十分了解，那么下面的知识对你来说如瓮中捉鳖，几乎所有的 `Python` 内置的字符串方法都被复制到 `Pandas` 的向量化字符串方法中。下图列举了 `Pandas` 的 `str` 方法借鉴 `Python` 字符串方法的内容：

len()	lower()	translate()	islower()
ljust()	upper()	startswith()	isupper()
rjust()	find()	endswith()	isnumeric()
center()	rfind()	isalnum()	isdecimal()
zfill()	index()	isalpha()	split()
strip()	rindex()	isdigit()	rsplit()
rstrip()	capitalize()	isspace()	partition()
lstrip()	swapcase()	istitle()	rpartition()

它们的作用与 `Python` 字符串的基本一致，但是需要注意这些方法的返回值不同。举两个例子：

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam', 'Eric Idle', 'Terry Jones', 'Michael Pal
in'])
monte.str.lower()  # 返回字符串
```

输出：

```
0 graham chapman
1    john cleese
2  terry gilliam
3      eric idle
4    terry jones
5  michael palin
dtype: object
```

```
monte.str.split()  # 返回列表
```

输出：

```
0   [Graham, Chapman]
1   [John, Cleese]
2   [Terry, Gilliam]
3   [Eric, Idle]
4   [Terry, Jones]
5   [Michael, Palin]
dtype: object
```

pandas 中还有一些自带的字符串方法，如下图所示：

方法	描述
get()	获取元素索引位置上的值，索引从 0 开始
slice()	对元素进行切片取值
slice_replace()	对元素进行切片替换
cat()	连接字符串（此功能比较复杂，建议阅读文档）
repeat()	重复元素
normalize()	将字符串转换为 Unicode 规范形式
pad()	在字符串的左边、右边或两边增加空格
wrap()	将字符串按照指定的宽度换行
join()	用分隔符连接 Series 的每个元素
get_dummies()	按照分隔符提取每个元素的 dummy 变量，转换为独热（one-hot）编码的 DataFrame

其中 get_dummies() 方法有点难以理解，给大家举个例子，假设有一个包含了某种编码信息的数据集，如 A = 出生在美国、 B = 出生在英国、 C = 喜欢奶酪、 D = 喜欢午餐肉：

```
full_monte = pd.DataFrame({
 'name': monte,
 'info': ['B|C|D', 'B|D', 'A|C', 'B|D', 'B|C', 'B|C|D']})
print(full_monte)
```

输出：

```
    info    name
0 B|C|D   Graham Chapman
1 B|D     John Cleese
2 A|C     Terry Gilliam
3 B|D     Eric Idle
4 B|C     Terry Jones
5 B|C|D   Michael Palin
```

get_dummies() 方法可以让你快速将这些指标变量分割成一个独热编码的 DataFrame （每个元素都是 0 或 1 ）：

```
full_monte['info'].str.get_dummies('|')
```

输出：

```
   A B C D
0 0 1 1 1
1 0 1 0 1
2 1 0 1 0
3 0 1 0 1
4 0 1 1 0
5 0 1 1 1
```

正则表达式方法

还有一些支持正则表达式的方法可以用来处理每个字符串元素。如下图所示：

方法	描述
match()	对每个元素调用 re.match()，返回布尔类型值
extract()	对每个元素调用 re.match()，返回匹配的字符串组（groups）
findall()	对每个元素调用 re.findall()
replace()	用正则模式替换字符串
contains()	对每个元素调用 re.search()，返回布尔类型值
count()	计算符合正则模式的字符串的数量
split()	等价于 str.split()，支持正则表达式
rsplit()	等价于 str.rsplit()，支持正则表达式

众所周知，正则表达式“无所不能”，我们可以利用正则实现一些独特的操作，例如提取每个人的 `first name` :

```
monte.str.extract('([A-Za-z]+)')
```

输出：

```
0  Graham
1  John
2  Terry
3  Eric
4  Terry
5  Michael
dtype: object
```

或者找出所有开头和结尾都是辅音字符的名字：

```
monte.str.findall(r'^[^AEIOU].*[^aeiou]$')
```

输出：

```
0 [Graham Chapman]
1 []
2 [Terry Gilliam]
3 []
4 [Terry Jones]
5 [Michael Palin]
dtype: object
```

3.2.8：日期与时间工具

相关知识

Pandas 是为金融模型而创建的，所以拥有一些功能非常强大的日期、时间、带时间索引数据的处理工具。本关卡介绍的日期与时间数据主要包含三类：

- 时间戳：表示某个具体的时间点（例如 2015 年 7 月 4 日上午 7 点）
- 时间间隔与周期：期表示开始时间点与结束时间点之间的时间长度，例如 2015 年（指的是 2015 年 1 月 1 日至 2015 年 12 月 31 日这段时间间隔）。周期通常是指一种特殊形式的时间间隔，每个间隔长度相同，彼此之间不会重叠（例如，以 24 小时为周期构成每一天）
- 时间增量或持续时间：表示精确的时间长度（例如，某程序运行持续时间 22.56 秒）

Python 的日期与时间工具

原生 Python 中也有处理日期与时间的工具，它与 Pandas 中处理时间的工具有着千丝万缕的联系。 Python 的日期与时间功能都在标准库的 datetime 模块和第三方库 dateutil 模块。如果你处理的时间数据量比较大，那么速度就会比较慢，这时就需要使用到 NumPy 中已经被编码的日期类型数组了。

NumPy中的datetime64类型

Python 原生日期格式的性能弱点促使 NumPy 团队为 NumPy 增加了自己的时间序列类型。 datetime64 类型将日期编码为 64 位整数，这样可以让日期数组非常紧凑（节省内存）。

```
In[3]: import numpy as np
       date = np.array('2015-07-04', dtype=np.datetime64)
       date
Out[3]: array(datetime.date(2015, 7, 4), dtype='datetime64[D]')
```

有了这个日期格式，既可以进行快速的向量化运算：

```
In[4]: date + np.arange(12) #由于date是datetime类型，所以向量化运算也是datetime类型的运算
Out[4]:
array(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10', '201
5-07-11', '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'], dtype='datetime64[D]')
```

由于 datetime64 对象是 64 位精度，所以可编码的时间范围可以是基本单元的 264 倍。也就是说， datetime64 在时间精度与最大时间跨度之间达成了一种平衡，也就是说， NumPy 会自动判断输入时间所需要使用的时间单位。

```
In[5]: np.datetime64('2015-07-04') # 天为单位
Out[5]: numpy.datetime64('2015-07-04')
In[6]: np.datetime64('2015-07-04 12:00') # 分钟为单位
Out[6]: numpy.datetime64('2015-07-04T12:00')
In[7]: np.datetime64('2015-07-04 12:59:59.50', 'ns') # 手动设置时间单位
Out[7]: numpy.datetime64('2015-07-04T12:59:59.500000000')
```

日期与时间单位格式代码表如下：

代码	含义	时间跨度（相对）	时间跨度（绝对）
Y	年（year）	± 9.2e18 年	[9.2e18 BC, 9.2e18 AD]
M	月（month）	± 7.6e17 年	[7.6e17 BC, 7.6e17 AD]
W	周（week）	± 1.7e17 年	[1.7e17 BC, 1.7e17 AD]
D	日（day）	± 2.5e16 年	[2.5e16 BC, 2.5e16 AD]
h	时（hour）	± 1.0e15 年	[1.0e15 BC, 1.0e15 AD]
m	分（minute）	± 1.7e13 年	[1.7e13 BC, 1.7e13 AD]
s	秒（second）	± 2.9e12 年	[2.9e9 BC, 2.9e9 AD]
ms	毫秒（millisecond）	± 2.9e9 年	[2.9e6 BC, 2.9e6 AD]
us	微秒（microsecond）	± 2.9e6 年	[290301 BC, 294241 AD]
ns	纳秒（nanosecond）	± 292 年	[1678 AD, 2262 AD]
ps	皮秒（picosecond）	± 106 天	[1969 AD, 1970 AD]
fs	飞秒（femtosecond）	± 2.6 小时	[1969 AD, 1970 AD]
as	原秒（attosecond）	± 9.2 秒	[1969 AD, 1970 AD

Pandas的日期与时间工具

`Pandas` 中的 `datetime` 是结合了原生 `Python` 和 `NumPy` 的 `datetime`，用来处理时间序列的基础数据类型如下：

- 针对时间戳数据， `Pandas` 提供了 `Timestamp` 类型。它本质上是 `Python` 的原生 `datetime` 类型的替代品，但是在性能更好的 `numpy.datetime64` 类型的基础上创建。对应的索引数据结构是 **`DatetimeIndex`** 。
- 针对时间周期数据， `Pandas` 提供了 `Period` 类型。这是利用 `numpy.datetime64` 类型将固定频率的时间间隔进行编码。对应的索引数据结构是 **`PeriodIndex`** 。
- 针对时间增量或持续时间， `Pandas` 提供了 `Timedelta` 类型。 `Timedelta` 是一种代替 `Python` 原生 `datetime.timedelta` 类型的高性能数据结构，同样是基于 `numpy.timedelta64` 类型。对应的索引数据结构是 **`TimedeltaIndex`** 。

最基础的日期 / 时间对象是 `Timestamp` 和 `DatetimeIndex` 。这两种对象可以直接使用，最常用的方法是 `pd.to_datetime()` 函数， 。对 `pd.to_datetime()` 传递一个日期会返回一个 `Timestamp` 类型，传递一个时间序列会返回一个 `DatetimeIndex` 类型

- **`DatetimeIndex`** 类型

```
In[8]: dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015', '2015-Jul-6', '07-07-2015', '2
0150708'])
       dates
Out[8]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07', '2015-07-08'], dtype='dat
etime64[ns]', freq=None)
```

- **`PeriodIndex`** 类型 任何 `DatetimeIndex` 类型都可以通过 `to_period()` 方法和一个频率代码转换成 `PeriodIndex` 类型， `PeridoIndex` 类型可以通过 `to_timestamp()` 方法转换为 `DatetimeIndex` 类型。下面用 `D` 将数据转换成单日的时间序列：

```
In[10]: dates.to_period('D')
Out[10]: PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07', '2015-07-08'], dtype='int6
```

```
4', freq='D')
```

- **`TimedeltaIndex`** 类型 当用一个日期减去另一个日期时，返回的结果是 `TimedeltaIndex` 类型：

```
In[11]: dates - dates[0]
Out[11]:
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'], dtype='timedelta64[ns]', freq=None)
```

为了能更简便地创建有规律的时间序列， `Pandas` 提供了一些方法： `pd.date_range()` 可以处理时间戳、 `pd.period_range()` 可以处理周期、 `pd.timedelta_range()` 可以处理时间间隔。

- **`pd.date_range()`** 通过开始日期、结束日期和频率代码（可选的）创建一个有规律的日期序列，默认的频率是天：

```
In[12]: pd.date_range('2015-07-03', '2015-07-10')
Out[12]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07', '2015-07-0
8', '2015-07-09', '2015-07-10'], dtype='datetime64[ns]', freq='D')
```

范围不一定非是开始时间和结束时间，也可以设置周期数 `periods` 来达到改目的：

```
In[13]: pd.date_range('2015-07-03', periods=8)
Out[13]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07', '2015-07-0
8', '2015-07-09', '2015-07-10'], dtype='datetime64[ns]', freq='D')
```

`freq` 表示时间间隔，默认是 `D` ，可以通过修改它来 `periods` 参数的意义：

```
In[14]: pd.date_range('2015-07-03', periods=8, freq='H')
Out[14]:
DatetimeIndex(['2015-07-03 00:00:00', '2015-07-03 01:00:00',
               '2015-07-03 02:00:00', '2015-07-03 03:00:00',
               '2015-07-03 04:00:00', '2015-07-03 05:00:00',
               '2015-07-03 06:00:00', '2015-07-03 07:00:00'],
               dtype='datetime64[ns]', freq='H')
```

- **`pd.timedelta_range()`** 如果要创建一个有规律的周期或时间间隔序列， `pd.timedelta_range()` 可以实现该功能：

```
In[15]: pd.period_range('2015-07', periods=8, freq='M')
Out[15]:
PeriodIndex(['2015-07', '2015-08', '2015-09', '2015-10', '2015-11', '2015-12', '2016-01', '2016-02'], dt
ype='int64', freq='M')
```

也可以通过修改 `freq` 参数实现各种频率的时间间隔序列。

3.2.9 时间序列的高级应用

相关知识

Pandas 时间序列工具的基础是时间频率或偏移量代码。就像之前见过的 `D（day）` 和 `H（hour）` 代码，我们可以用这些代码设置任意需要的时间间隔。

Pandas频率代码表如下：

代码	描述
D	天（calendar day，按日历算，含双休日）
W	周（weekly） M 月末（month end）
Q	季末（quarter end）
A	年末（year end）
H	小时（hours）
T	分钟（minutes）
S	秒（seconds）
L	毫秒（milliseonds）
U	微秒（microseconds）
N	纳秒（nanoseconds）
B	天（business day，仅含工作日）
BM	月末（business month end，仅含工作日）
BQ	季末（business quarter end，仅含工作日）
BA	年末（business year end，仅含工作日）
BH	小时（business hours，工作时间）
MS	月初（month start）
BMS	月初（business month start，仅含工作日）
QS	季初（quarter start）
BQS	季初（business quarter start，仅含工作日）
AS	年初（year start）
BAS	年初（business year start，仅含工作日）

时间频率与偏移量

我们可以在频率代码后面加三位月份缩写字母来改变季、年频率的开始时间，也可以再后面加三位星期缩写字母来改变一周的开始时间：

- `Q-JAN、BQ-FEB、QS-MAR、BQS-APR` 等
- `W-SUN、W-MON、W-TUE、W-WED` 等

时间频率组合使用：

```
In[0]: pd.timedelta_range(0,periods=9,freq="2H30T")
Out[0]: TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00', '12:30:00', '15:00:00', '
17:30:00', '20:00:00'], dtype='timedelta64[ns]', freq='150T')
```

比如直接创建一个工作日偏移序列：

```
In[1]: from pandas.tseries.offsets import BDay
       pd.date_range('2015-07-01', periods=5, freq=BDay())
Out[1]: DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03', '2015-07-06', '2015-07-07'], dtype='datetime
64[ns]', freq='B')
```

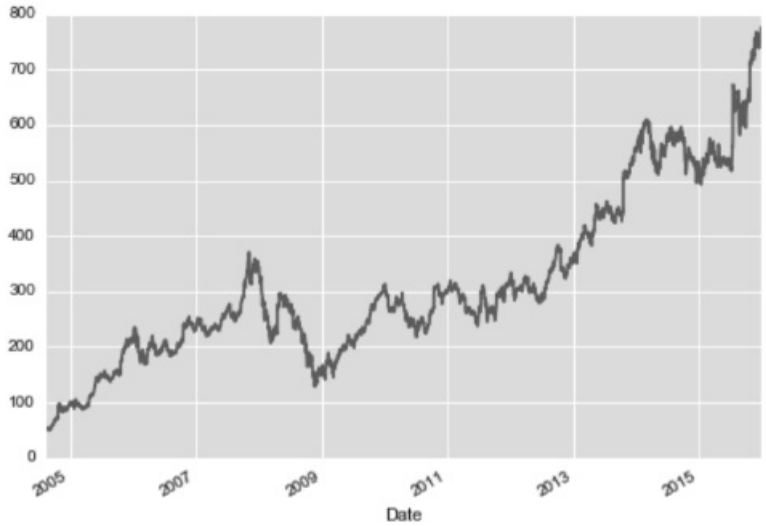
重新取样、迁移和窗口

重新取样

处理时间序列数据时，经常需要按照新的频率（更高频率、更低频率）对数据进行重新取样。举个例子，首先通过 `pandas-datareader` 程序包(需要手动安装)导入 Google 的历史股票价格，只获取它的收盘价：

```
In[2]:
from pandas_datareader import data
goog = data.DataReader('GOOG',start="2014",end="2016",data_source="google")['Close']   # Close表示收盘价的列
In[3]:
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
goog.plot();   #数据可视化
```

输出：

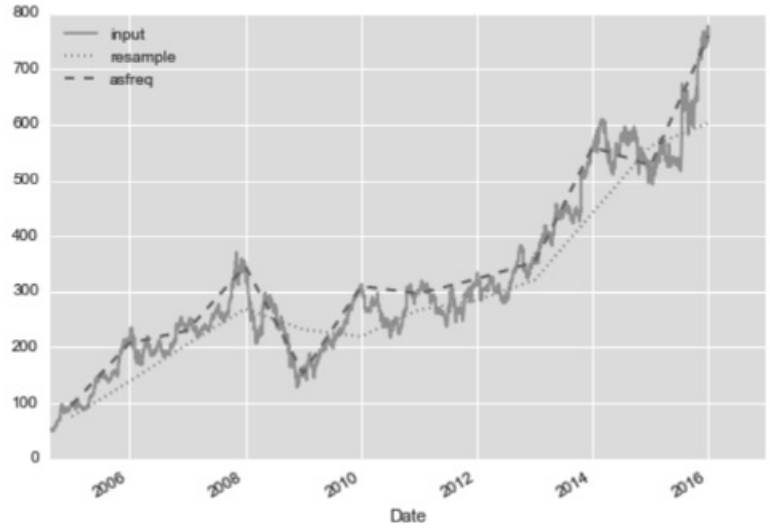


我们可以通过 `resample()` 方法和 `asfreq()` 方法解决这个问题， `resample()` 方法是以数据累计为基础，而 `asfreq()` 方法是以数据选择为基础。

```
In[4]:
goog.plot(alpha=0.5, style='-')
```

```
goog.resample('BA').mean().plot(style=':')
goog.asfreq('BA').plot(style='--');
plt.legend(['input', 'resample', 'asfreq'], loc='upper left');
```

输出：

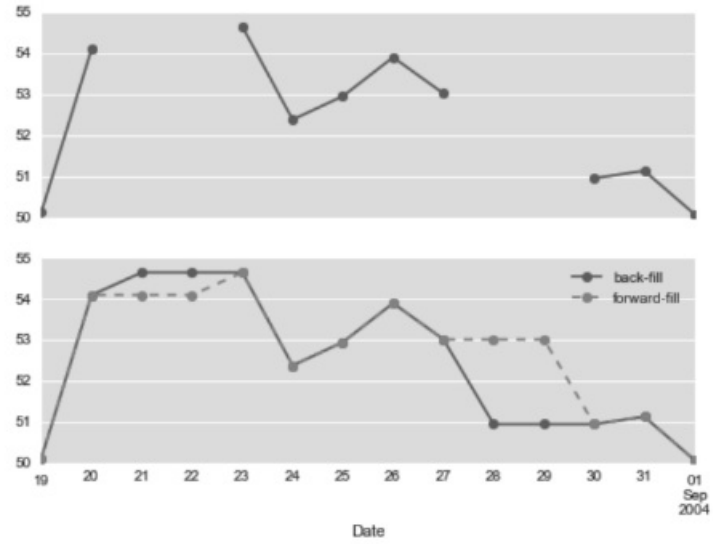


请注意这两种取样方法的差异：在每个数据点上，`resample` 反映的是上一年的均值，而 `asfreq` 反映的是上一年最后一个工作日的收盘价。

数据集中经常会出现缺失值，从上面的例子来看，由于周末和节假日股市休市，周末和节假日就会产生缺失值，上面介绍的两种方法默认使用的是向前取样作为缺失值处理。与前面介绍过的 `pd.fillna()` 函数类似，`asfreq()` 有一个 `method` 参数可以设置填充缺失值的方式。

```
In[5]: fig, ax = plt.subplots(2, sharex=True)
       data = goog.iloc[:10]
       data.asfreq('D').plot(ax=ax[0], marker='o')
       data.asfreq('D', method='bfill').plot(ax=ax[1], style='-o')
       data.asfreq('D', method='ffill').plot(ax=ax[1], style='--o')
       ax[1].legend(["back-fill", "forward-fill"]);
```

输出：



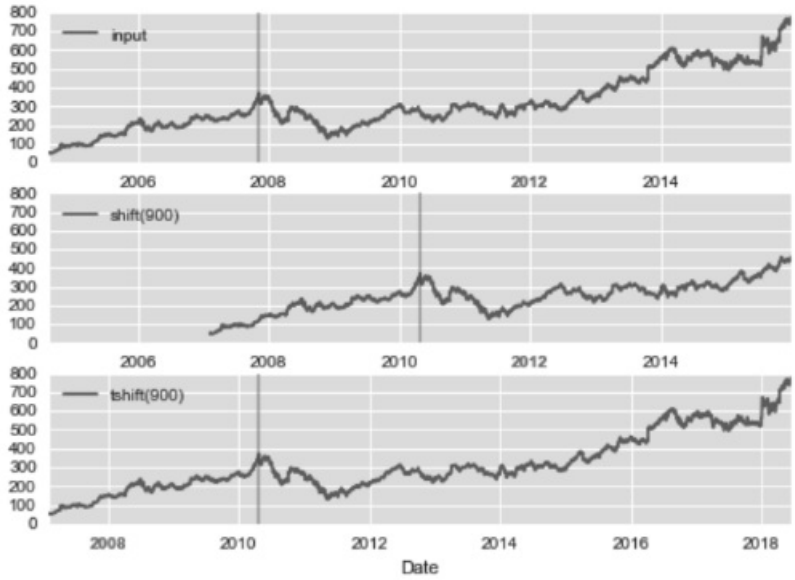
时间迁移

另一种常用的时间序列操作是对数据按时间进行迁移，`Pandas`有两种解决这类问题的方法：`shift()`和`tshift()`。简单来说，`shift()`就是迁移数据，而`tshift()`就是迁移索引。两种方法都是按照频率代码进行迁移。

下面我们将用`shift()`和`tshift()`这两种方法让数据迁移`900`天：

```
In[6]: fig, ax = plt.subplots(3, sharey=True)
       # 对数据应用时间频率，用向后填充解决缺失值
       goog = goog.asfreq('D', method='pad')
       goog.plot(ax=ax[0])
       goog.shift(900).plot(ax=ax[1])
       goog.tshift(900).plot(ax=ax[2])
       # 设置图例与标签
       local_max = pd.to_datetime('2007-11-05')
       offset = pd.Timedelta(900, 'D')
       ax[0].legend(['input'], loc=2)
       ax[0].get_xticklabels()[4].set(weight='heavy', color='red')
       ax[0].axvline(local_max, alpha=0.3, color='red')
       ax[1].legend(['shift(900)'], loc=2)
       ax[1].get_xticklabels()[4].set(weight='heavy', color='red')
       ax[1].axvline(local_max + offset, alpha=0.3, color='red')
       ax[2].legend(['tshift(900)'], loc=2)
       ax[2].get_xticklabels()[1].set(weight='heavy', color='red')
        ax[2].axvline(local_max + offset, alpha=0.3, color='red');
```

输出：



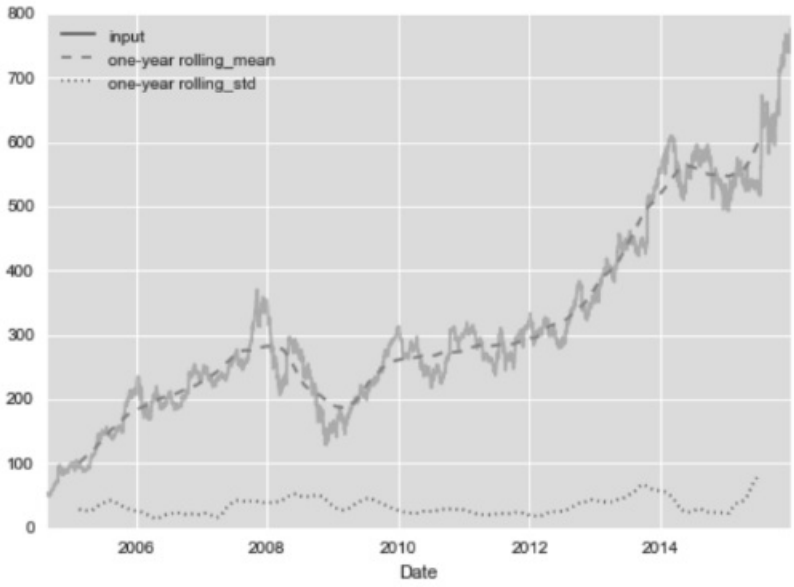
`shift(900)`将数据向前推进了`900`天，这样图形中的一段就消失了（最左侧就变成了缺失值），而`tshift(900)`方法是将时间索引值向前推进了`900`天。

移动时间窗口

`Pandas` 处理时间序列数据的第 `3` 种操作是移动统计值。这些指标可以通过 `Series` 和 `DataFrame` 的 `rolling()` 属性来实现，它会返回与 `groupby` 操作类似的结果，移动视图使得许多累计操作成为可能。

```
In[7]:  rolling = goog.rolling(365, center=True)
         data = pd.DataFrame({'input': goog, 'one-year rolling_mean': rolling.mean(), 'one-year rolling_std':
 rolling.std()})
         ax = data.plot(style=['-', '--', ':'])
         ax.lines[0].set_alpha(0.3)
```

输出：



Chapter4 Matplotlib

Matplotlib 的功能和 matlab 中的画图的功能十分类似。因为 matlab 进行画图相对来说比较复杂，所以使用 Python 中的 Matplotlib 来画图比较方便。

Matplotlib 是 Python 中的一个包，主要用于绘制 2D 图形（当然也可以绘制 3D，但是需要额外安装支持的工具包）。在数据分析领域它有很大的地位，而且具有丰富的扩展，能实现更强大的功能。

Matplotlib 相关实训已在 `educoder` 平台上提供，若感兴趣可以输入链接进行体验。

链接：https://www.educoder.net/paths/302

4.1.1：画图接口

导入matplotlib

和 `numpy` , `pandas` 一样，在导入 `matplotlib` 时我们也可以用一些常用的简写形式:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

`pyplot` 是最常用的画图模块接口，功能非常强大。

显示图像

开发环境的不同，显示图像的方式也就不一样，一般有三种开发环境，分别是脚本、 `IPython shell` 、 `IPython Notebook` 。

在脚本中使用 `matplotlib` 进行可视化时显示图像可以使用 `plt.show()` 。

在 `IPython shell` 中使用 `matplotlib` 可视化非常方便，使用 `%matplotlib` 命令启动 `matplotlib` 模式。之后的任何 `plt` 命令都会自动打开一个图像窗口，当有新的命令，图像就会更新。但对已经画好的图像不会自动实时更新。对于这种可以使用 `plt.draw()` 强制更新。

在 `IPython Notebook` 中画图和 `IPthon shell` 类似，也需要使用 `%matplotlib` 命令。图像的显示是嵌在 `IPython Notebook` 页面中。有两种展示形式: `%matplotlib notebook` 交互式图形； `%matplotlib inline` 静态图形。

`matplotlib` 还可以直接将图像保存文件，通过 `plt.savefig("test.jpg")` 命令保存文件。

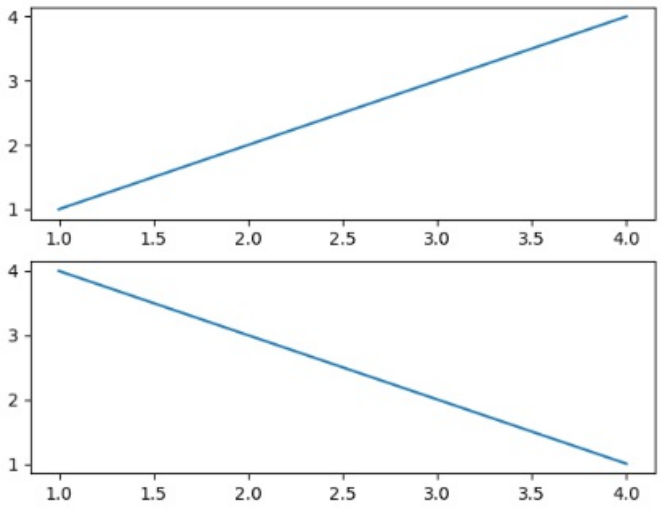
```
plt.savefig("test.jpg")
```

画图接口

`matplotlib` 有两个画图接口：一个是便捷的 `matlab` 风格接口，另一个是功能更强大的面向对象接口。 `matplotlib` 的 `matlab` 接口许多语法都和 `MATLAB` 类似，所以使用过 `MATLAB` 的朋友们想必很快就能上手 `matplotlib` 。

```
import matplotlib.pyplot as plt#导入模块

plt.figure(figsize=(10,10))#创建图形,并设置大小为10 x 10

plt.subplot(2,1,1)#创建子图1（行，列，子图编号）

plt.plot([1,2,3,4], [1,2,3,4])

plt.subplot(2,1,2)#创建子图2（行，列，子图编号）

plt.plot([4,3,2,1], [1,2,3,4])
```

```
plt.show()
```



面向对象接口可以适应更加复杂的场景，更好地控制图形，在画比较复杂的图形市，面向对象方法会更方便。通过下面的代码，可以用面向对象接口重新创建之前的图形。

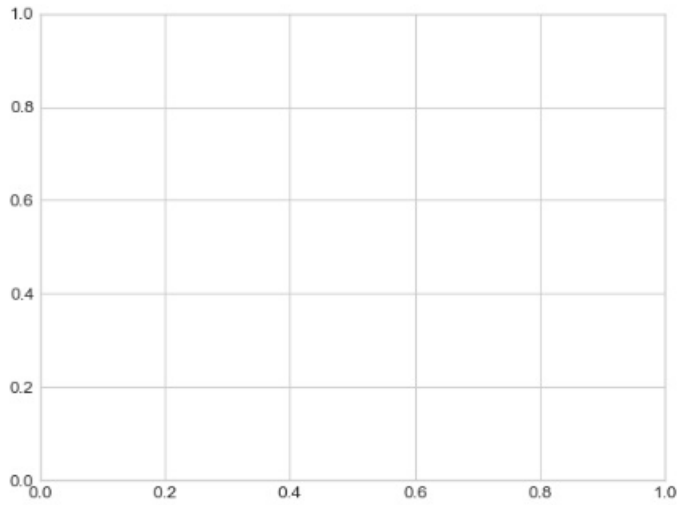
```
fig,ax=plt.subplots(2)#ax是一个包含2个axes对象的数组

ax[0].plot([1,2,3,4], [1,2,3,4])

ax[1].plot([4,3,2,1], [1,2,3,4])

plt.show()
```

4.1.2：线形图

绘制线形图

在所有图形中，最简单的应该就是线性方程 `y = f (x)` 的可视化了。来看看如何创建这个简单的线形图。要画 `Matplotlib` 图形时，都需要先创建一个图形 `fig` 和一个坐标轴 `ax` 。创建图形与坐标轴的最简单做法是：

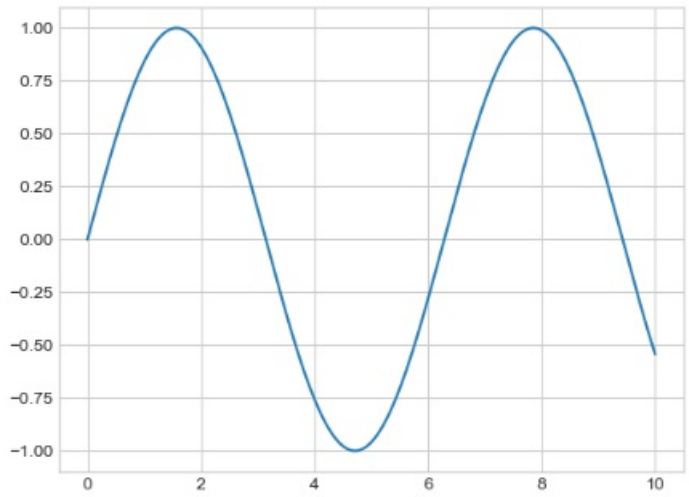
```
import matplotlib.pyplot as plt#导入模块
plt.style.use('seaborn-whitegrid')#设置matplotlib画图样式

fig = plt.figure()
ax = plt.axes()
```



在 `Matplotlib` 中， `figure` (`plt.Figure` 类的一个实例)可以被看成是个能够容纳各种坐标轴、图形、文字和标签的容器。就像你在图中看到的那样， `axes` (`plt.Axes` 类的一个实例)是一个带有刻度和标签的矩形，最终会包含所有可视化的图形元素。在这里我们一般使用变量 `fig` 表示一个图形实例，用变量 `ax` 表示一个坐标轴实例。

接下来使用 `ax.plot` 画图，从简单的正弦曲线开始：

```
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x))
```

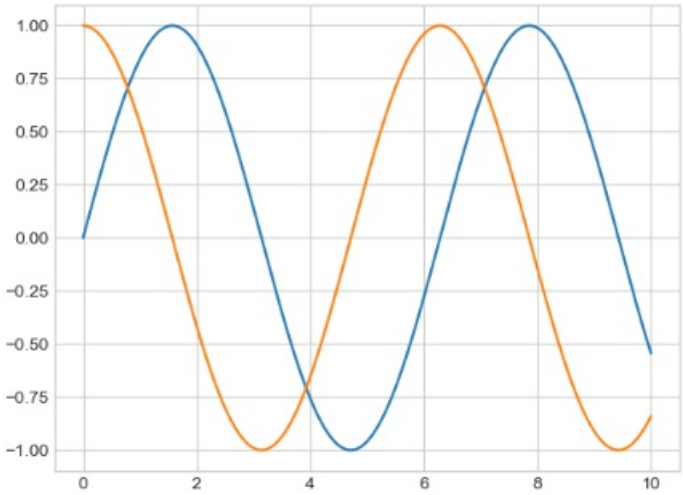


也可以使用 `pylab` 接口画图，这时图形与坐标轴都在底层执行，执行结果和上图一样：

```
plt.plot(x, np.sin(x))
```

试想下如果我们重复调用 `plot` 命令会发生什么，它会在一张图中创建多条线：

```
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
```

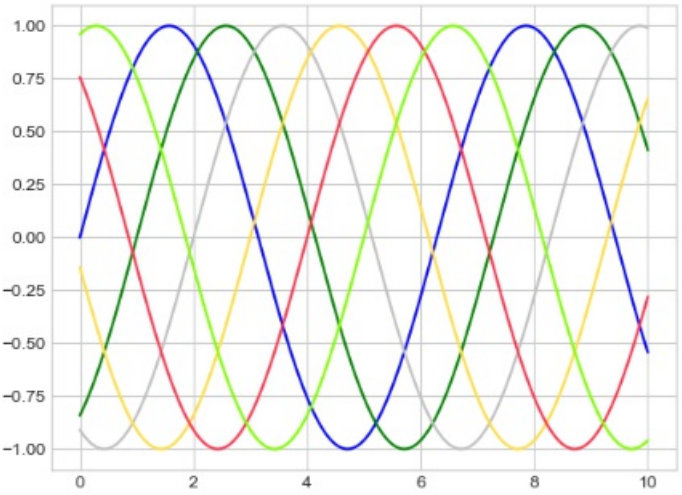


设置颜色和风格

在画图的过程中通常对图形的第一次调整是调整它线条的颜色与风格。 `plt.plot()` 函数可以通过相应的参数设置颜色和风格，修改颜色使用 `color` 参数，它支持各种颜色值的字符串，具体使用如下：

```
plt.plot(x, np.sin(x - 0), color='blue') # 标准颜色名称
plt.plot(x, np.sin(x - 1), color='g') # 缩写颜色代码（rgbcmyk）
plt.plot(x, np.sin(x - 2), color='0.75') # 范围在0~1的灰度值
plt.plot(x, np.sin(x - 3), color='#FFDD44') # 十六进制（RRGGBB，00~FF）
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB元组，范围在0~1
```

```
plt.plot(x, np.sin(x - 5), color='chartreuse') # HTML颜色名称
```

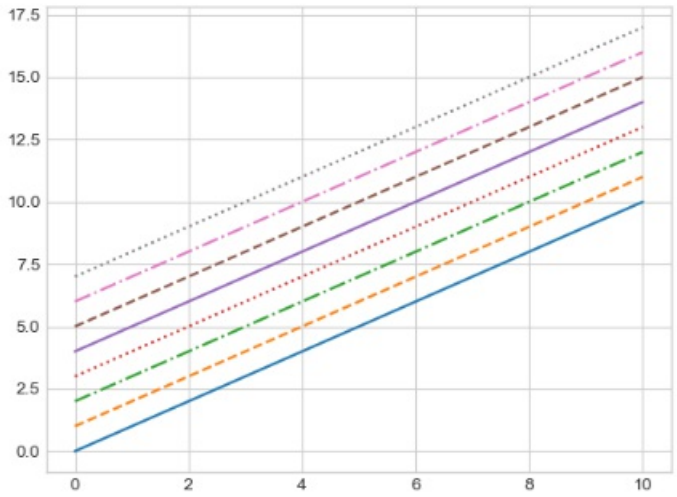


常用颜色对应值：

取值	颜色
‘b’	blue
‘g’	green
‘r’	red
‘c’	cyan
‘m’	magenta
‘y’	yellow
‘k’	black
‘w’	white

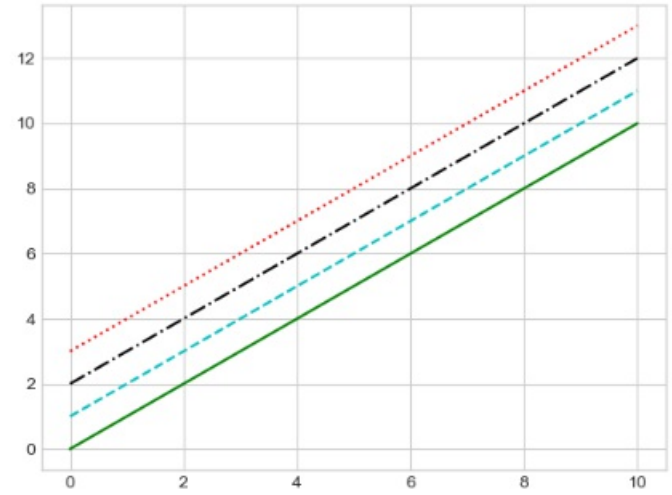
如果不指定颜色，`matplotlib`会为多条线自动循环使用一组默认的颜色。设置样式使用`linestyle`参数：

```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted')
#也可以用下面的简写形式
plt.plot(x, x + 4, linestyle='-') # 实线
plt.plot(x, x + 5, linestyle='--') # 虚线
plt.plot(x, x + 6, linestyle='-.') # 点划线
plt.plot(x, x + 7, linestyle=':') # 实点线
```



还可以将 `linestyle` 和 `color` 编码组合起来，作为 `plt.plot()` 函数的一个非关键字参数使用：

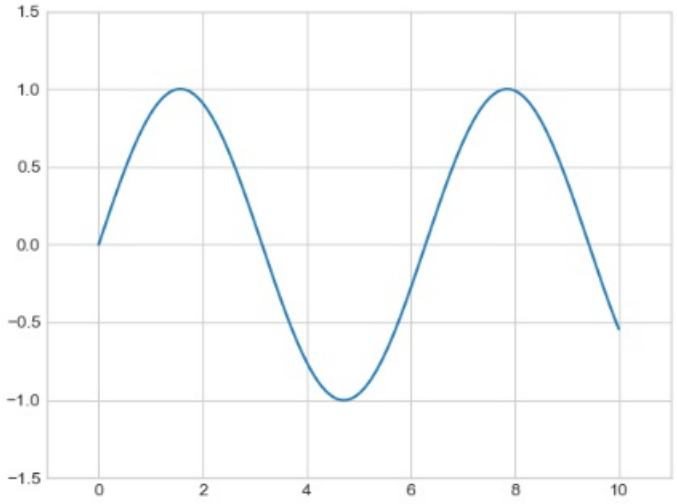
```
plt.plot(x, x + 0, '-g') # 绿色实线
plt.plot(x, x + 1, '--c') # 青色虚线
plt.plot(x, x + 2, '-.k') # 黑色点划线
plt.plot(x, x + 3, ':r'); # 红色实点线
```



设置坐标轴上下限

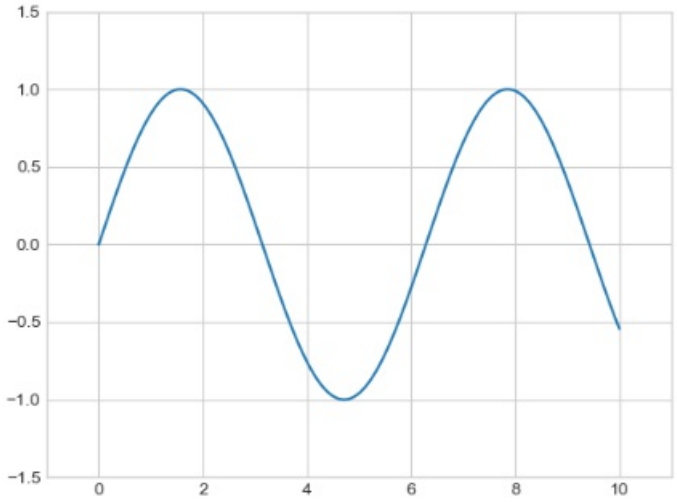
虽然 `matplotlib` 会自动为你的图形选择最合适的坐标轴上下限，但是有时自定义坐标轴上下线可能会更好。调整坐标轴上下限最基础的方式是 `plt.xlim()` 和 `plt.ylim()`：

```
plt.plot(x, np.sin(x))

plt.xlim(-1, 11)

plt.ylim(-1.5, 1.5)
```



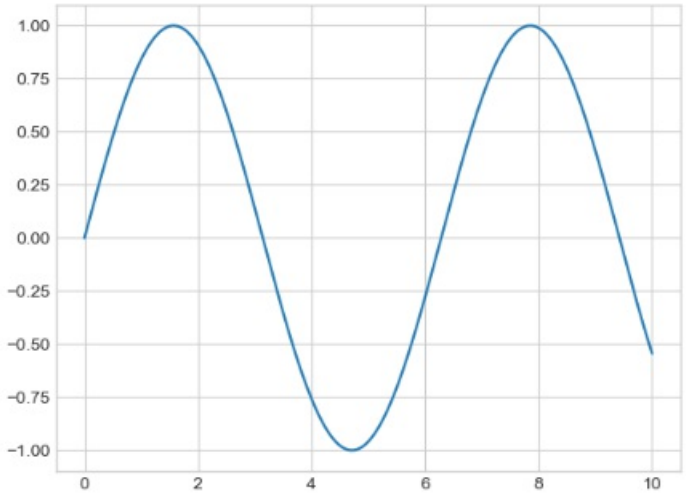
如果你想要让坐标轴逆序显示，那么只需要逆序设置坐标轴刻度值就可以了。`matplotlib` 还有一个方法是 `plt.axis()` 。通过传入[`xmin` ， `xmax` ， `ymin` ， `ymax`]对应的值，这样就可以用一行代码设置 `x` 和 `y` 的限值：

```
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5])
```



还支持按照图形的内容自动收紧坐标轴，不留空白区域：

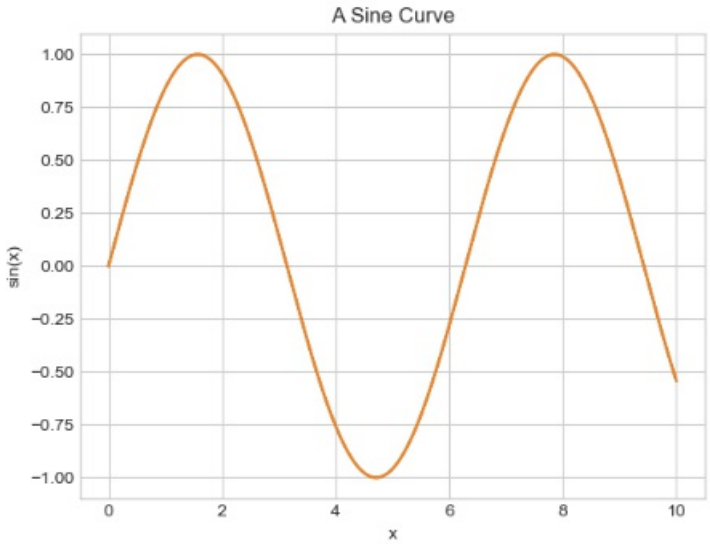
```
plt.plot(x, np.sin(x))
plt.axis('tight')
```



设置图形标签

图形标签与坐标轴标题是最简单的标签，设置方法如下：

```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```

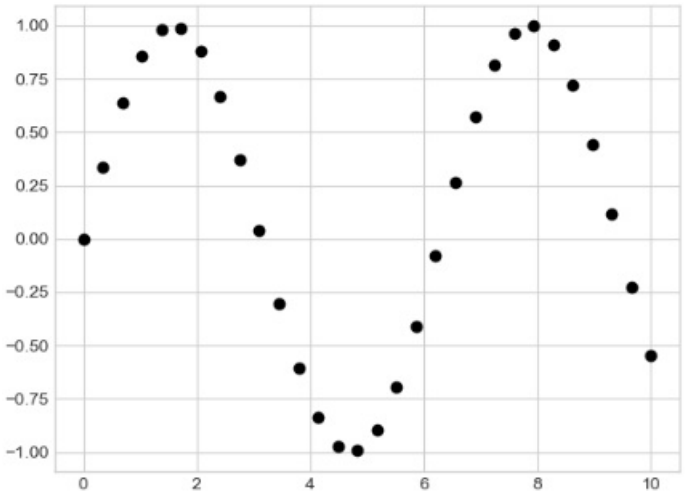


4.1.3：线形图

plot绘制散点图

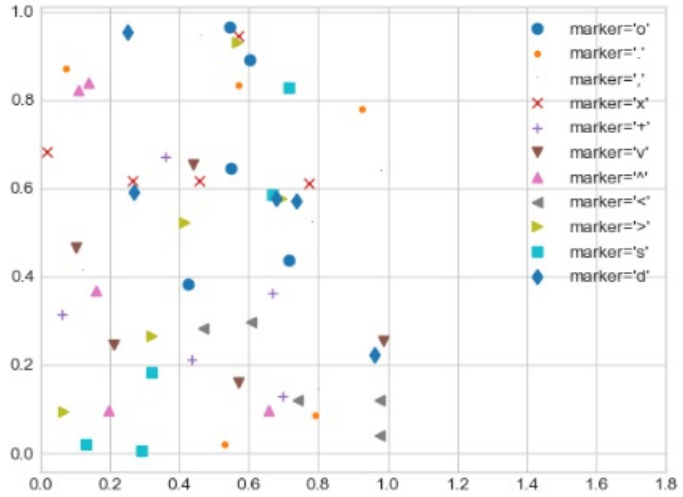
散点图也是在数据科学中常用图之一，前面我们学习了使用 `plt.plot/ax.plot` 画线形图的方法。同样的，现在用这些函数来画散点图：

```
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.plot(x, y, 'o', color='black')
```



函数的第三个参数是一个字符，表示图形符号的类型。与我们之前用 `-` 和 `--` 设置线条属性类似，对应的图形标记也有缩写形式。

```
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker,label="marker='{0}'".format(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
plt.savefig("T1.png")
plt.show()
```



常用标记如下：

取值	含义
‘.’	point marker
‘,’	pixel marker
‘o’	circle marker
‘v’	triangle_down marker
‘^’	triangle_up marker
‘<’	triangle_left marker
‘>’	triangle_right marker
‘1’	tri_down marker
‘2’	tri_up marker
‘3’	tri_left marker
‘4’	tri_right marker
‘s’	square marker
‘p’	pentagon marker
‘*’	star marker
‘h’	hexagon1 marker
‘H’	hexagon2 marker
‘+’	plus marker
‘x’	x marker
‘D’	diamond marker
‘d’	thin_diamond marker
‘|’	vline marker
‘_’	hline marker

`plt.plot` 函数非常灵活，可以满足各种不同的可视化配置需求。关于具体配置的完整描述，由于篇幅有限，请参考 `plt.plot` 文档，这里就不多介绍了，大家多多尝试就好。

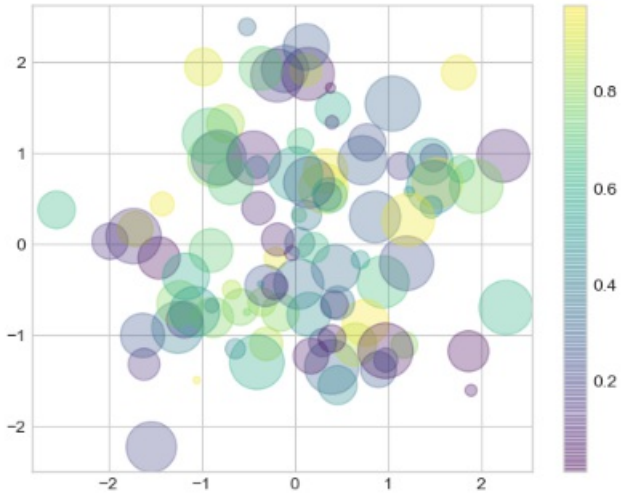
scatter画散点图

另一个可以创建散点图的函数是 `plt.scatter` 。它的功能非常强大，其用法与 `plt.plot` 函数类似：

```
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.scatter(x, y, marker='o');
```

结果和前面 `plt.plot` 画的一样。 `plt.scatter` 和 `plt.plot` 的主要差别在于，前者在创建散点图时具有更高的灵活性，可以单独控制每个散点与数据匹配，也可以让每个散点具有不同的属性(大小、颜色等)。接下来画一个随机散点图，里面有各种颜色和大小的散点。为了能更好的显示重叠部分，用 `alpha` 参数来调整透明度：

```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
cmap='viridis')
plt.colorbar() # 显示颜色条
```

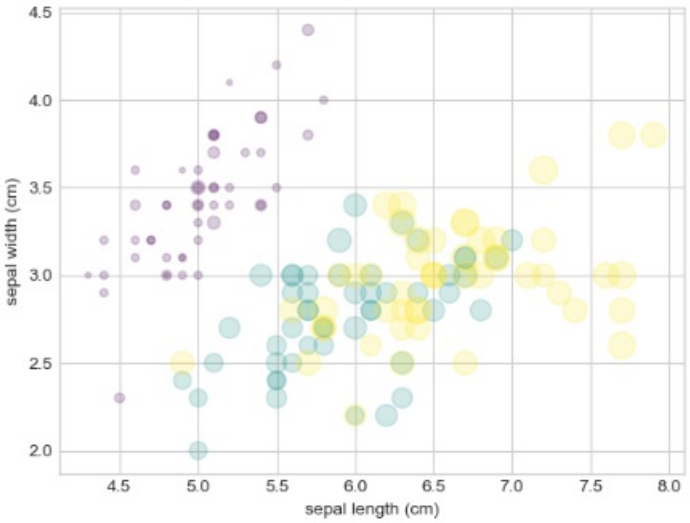


这里散点的大小以像素为单位。颜色为浮点数，自动映射成颜色条(`color scale` ，通过 `colorbar` ()显示)。当取值为浮点数时，它所对应的颜色则是对应的 `colormap` 上对应长度的取值。 `colormap` 就像以下这样的条带：

这样，散点的颜色与大小就可以在可视化图中显示多维数据的信息了。例如，可以使用 `sklearn` 程序库中的鸢尾花数据来演示。它里面有三种花，每个样本是一种花，其花瓣与花萼的长度与宽度都经过了测量：

```
from sklearn.datasets import load_iris
```

```
iris = load_iris()
features = iris.data.T#加载数据
plt.scatter(features[0], features[1], alpha=0.2,
s=100*features[3], c=iris.target, cmap='viridis')
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1]);
```



散点图可以让我们同时看到不同维度的数据：每个点的坐标值(`x` , `y`) 分别表示花萼的长度和宽度，而点的大小表示花瓣的宽度，三种颜色对应三种不同类型的鸢尾花。这类多颜色与多特征的散点图在探索与演示数据时非常有用。

plot与scatter效率对比

`plot` 与 `scatter` 除了特征上的差异之外，在数据量较大时， `plot` 的效率将大大高于 `scatter` 。这时由于 `scatter` 会对每个散点进行单独的大小与颜色的渲染，因此渲染器会消耗更多的资源。而在 `plot` 中，散点基本都彼此复制，因此整个数据集中的所有点的颜色、大小只需要配置一次。所以面对大型数据集时， `plot` 方法比 `scatter` 方法好。

4.1.4：直方图

什么是直方图

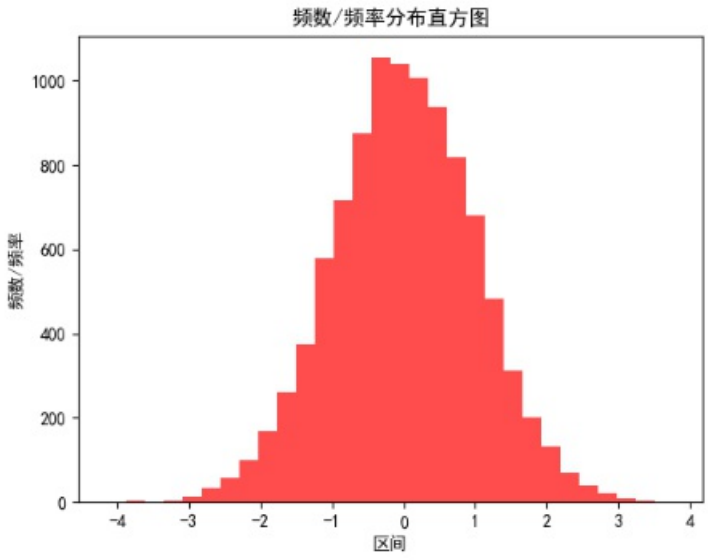
单从外表上看直方图和条形图非常相似。首先需要区分清楚概念：直方图和条形图。

- 条形图用长条形表示每一个类别，长条形的长度表示类别的频数，宽度表示表示类别。
- 直方图是一种统计报告图，形式上也是一个个的长条形，但是直方图用长条形的面积表示频数，所以长条形的高度表示频数组距，宽度表示组距，其长度和宽度均有意义。当宽度相同时，一般就用长条形长度表示频数。

绘制直方图

直方图一般用来描述等距数据。直观上，直方图各个长条形是衔接在一起的，表示数据间的数学关系。

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib

# 设置matplotlib正常显示中文和负号
matplotlib.rcParams['font.sans-serif']=['SimHei']   # 用黑体显示中文
matplotlib.rcParams['axes.unicode_minus']=False     # 正常显示负号
# 随机生成（10000,）服从正态分布的数据
data = np.random.randn(10000)
plt.hist(data,bins=30, normed=0, facecolor="red", alpha=0.7)
# 显示横轴标签
plt.xlabel("区间")
# 显示纵轴标签
plt.ylabel("频数/频率")
# 显示图标题
plt.title("频数/频率分布直方图")
```



参数	作用
data	必选参数，绘图数据
bins	直方图的长条形数目，可选项，默认为10
normed	是否将得到的直方图向量归一化，可选项，默认为0，代表不归一化，显示频数。 normed=1，表示归一化，显示频率。
facecolor	长条形的颜色
edgecolor	长条形边框的颜色
alpha	透明度

4.1.5：饼图

饼图

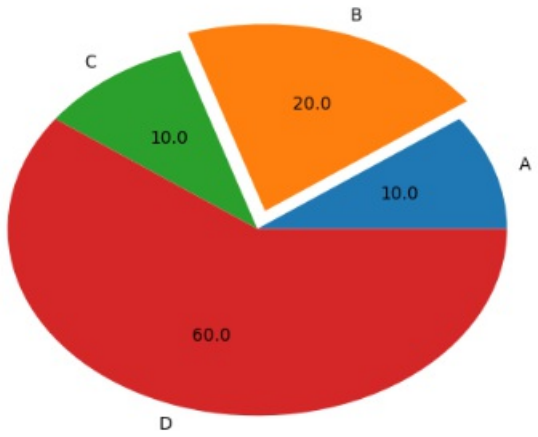
饼图又称圆饼图、圆形图等，它是利用圆形及圆内扇形面积来表示数值大小的图形。饼图主要用于总体中各组成部分所占比重的研究。

绘制饼图

`matplotlib` 用来绘制饼图的函数是 `pie()` 。常用参数如下：

参数	作用
x	(每一块的比例，如果sum(x) > 1会使用sum(x)归一化
labels	饼图外侧显示的说明文字
explode	(每一块)离开中心距离
startangle	起始绘制角度,默认图是从x轴正方向逆时针画起,如设定=90则从y轴正方向画起
shadow	在饼图下面画一个阴影。默认值：False，即不画阴影
autopct	控制饼图内百分比设置
radius	控制饼图半径，默认值为1
wedgeprops	字典类型，可选参数，默认值：None。参数字典传递给wedge对象用来画一个饼图。

```
labels = 'A','B','C','D'
sizes = [10,20,10,60]
plt.pie(sizes,labels=labels,explode = (0,0.1,0,0),autopct='%1.1f')

plt.show()
```



嵌套饼图

嵌套饼图通常被称为空心饼图图表。空心饼图形状的效果是通过 `wedgeprops` 参数设置馅饼楔形的宽度来实现的:

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

size = 0.3
vals = np.array([[60., 32.], [37., 40.], [29., 10.]])
cmap = plt.get_cmap("tab20c")
outer_colors = cmap(np.arange(3)*4)
inner_colors = cmap(np.array([1, 2, 5, 6, 9, 10]))

ax.pie(vals.sum(axis=1), radius=1, colors=outer_colors,
       wedgeprops=dict(width=size, edgecolor='w') )

ax.pie(vals.flatten(), radius=1-size, colors=inner_colors,
       wedgeprops=dict(width=size, edgecolor='w'))
ax.set(aspect="equal" )
```

4.1.6：手动创建子图

plt.axes创建子图

前面已经介绍过 `plt.axes` 函数，这个函数默认配置是创建一个标准的坐标轴，填满整张图。它还有一个可选的参数，由图形坐标系统的四个值构成。这四个值表示为坐标系的[底坐标、左坐标、宽度、高度]，数值的取值范围为左下角为 `0` ，右上角为 `1` 。

下面演示在右上角创建一个画中画：

```
x1 = plt.axes() # 默认坐标轴
ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
```

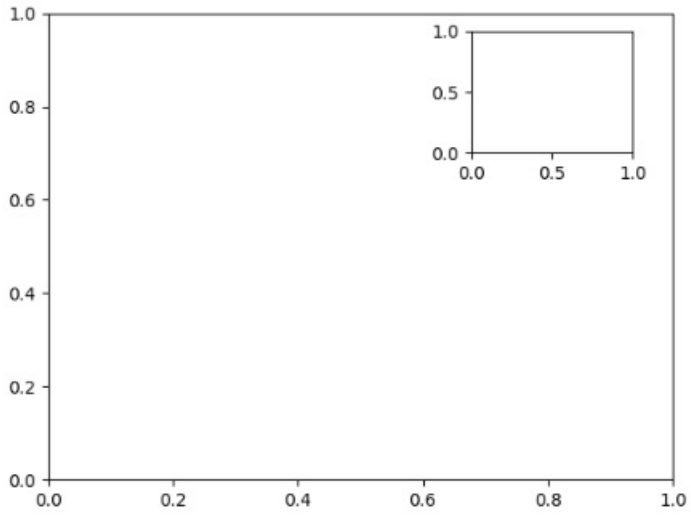
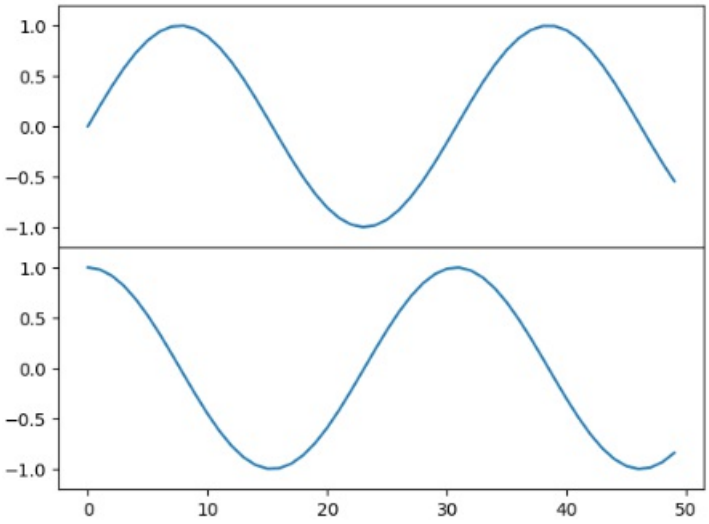


fig.add_axes()创建子图

面向对象画图接口中类似的命令由 `fig.add_axes()` 。用这个命令创建两个竖直排列的坐标轴：

```
fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.5, 0.8, 0.4],
xticklabels=[], ylim=(-1.2, 1.2))
ax2 = fig.add_axes([0.1, 0.1, 0.8, 0.4],
ylim=(-1.2, 1.2))
x = np.linspace(0, 10)
ax1.plot(np.sin(x))
ax2.plot(np.cos(x))
```



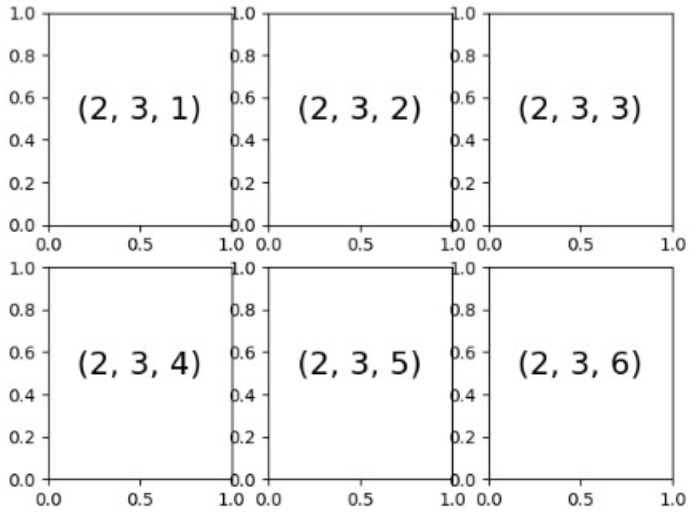
可以看到两个紧挨着的坐标轴：上子图的(起点 `y` 坐标为 `0.5` 位置)与下子图 `x` 轴刻度是对应的（起点 y 坐标为 0.1，高度为 0.4）。

4.1.7：网格子图

plt.subplot()绘制子图

若干彼此对齐的行列子图是常见的可视化任务，`matplotlib`拥有一些可以轻松创建它们的简便方法。最底层且最常用的方法是`plt.subplot()`。这个函数在一个网格中创建一个子图，该函数由三个整型参数，依次为将要创建的网格子图行数、列数和索引值，索引值从1开始，从左上到右下递增。

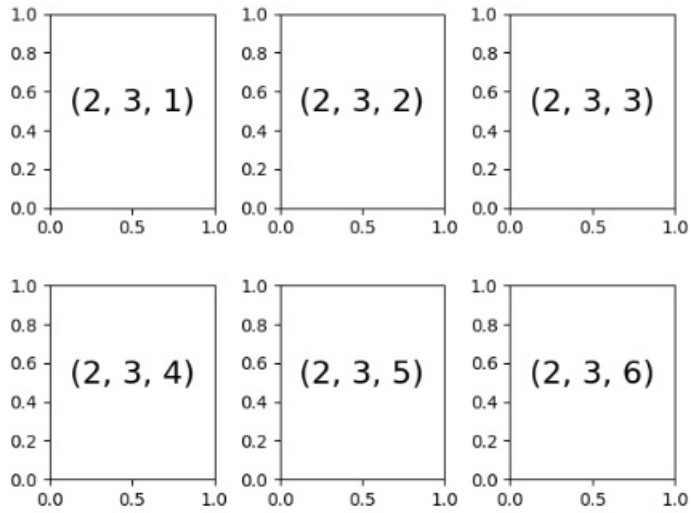
```
for i in range(1, 7):
    plt.subplot(2, 3, i)
    plt.text(0.5, 0.5, str((2, 3, i)),
        fontsize=18, ha='center')
```



调整子图之间的间隔

如上图`y`轴的刻度有的已经和前面的子图重叠，`matplotlib`提供`plt.subplots_adjust`命令调整子图之间的间隔。用面向对象接口的命令`fig.add_subplot()`可以取得同样的效果。

```
fig = plt.figure()
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(1, 7):
    ax = fig.add_subplot(2, 3, i)
    ax.text(0.5, 0.5, str((2, 3, i)),
        fontsize=18, ha='center')
```

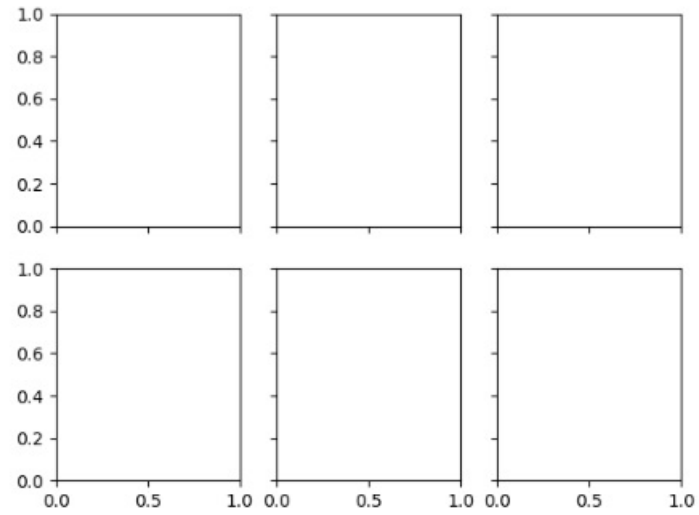


这里我们通过设置 `plt.subplots_adjust` 的 `hspace` 与 `wspace` 参数设置与图形高度与宽度一致的子图间距，数值以子图的尺寸为单位。

plt.subplots创建网格

当我们需要创建一个大型网格子图时，就没办法使用前面那种亦步亦趋的方法了，尤其是当你想隐藏内部子图的 `x` 轴与 `y` 轴标题时。 `matplotlib` 提供了 `plt.subplots()` 来解决这个问题。这个函数不是用来创建单个子图的，而是用一行代码创建多个子图，并放回一个包含子图的 `numpy` 数组。关键参数是行数与列数以及可选参数 `sharex` 与 `sharey` 。 让我们创建一个 `2 × 3` 的网格子图，同行使用相同的 `y` 坐标，同列使用相同的 `y` 轴坐标：

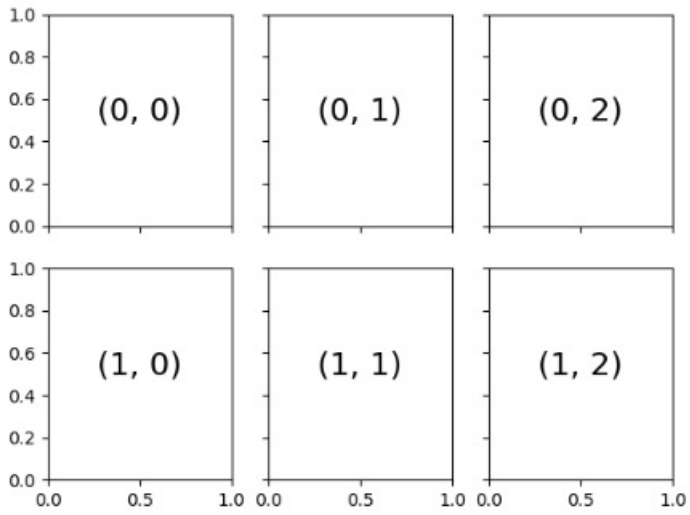
```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
```



设置 `sharex` 与 `sharey` 参数后，我们就可以自动去掉网格内部子图的标签。 坐标轴实例网格的放回结果是一个 `numpy` 数组，这样就可以通过标准的数组取值方式轻松获取想要的坐标轴了:

```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
for i in range(2):
```

```
    for j in range(3):
        ax[i, j].text(0.5, 0.5, str((i, j)),
            fontsize=18, ha='center')
```



4.1.8：更复杂的排列方式

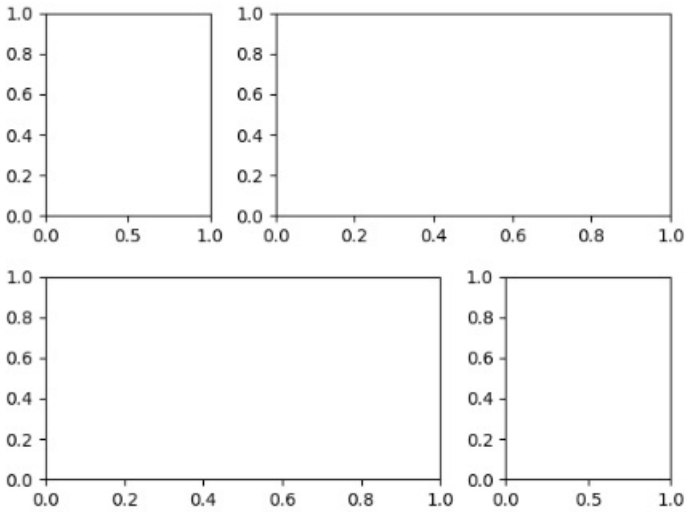
不规则子图

`matplotlib` 还支持不规则的多行多列子图网格。 `plt.GridSpec()` 对象本事不能直接创建一个图形，他只是 `plt.subplot()` 命令可以识别的简易接口。这里创建了一个带行列间距的 `2×3` 网格：

```
grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
```

`plt.GridSpec()` 支持通过类似 `python` 切片的语法设置子图的位置和扩展尺寸：

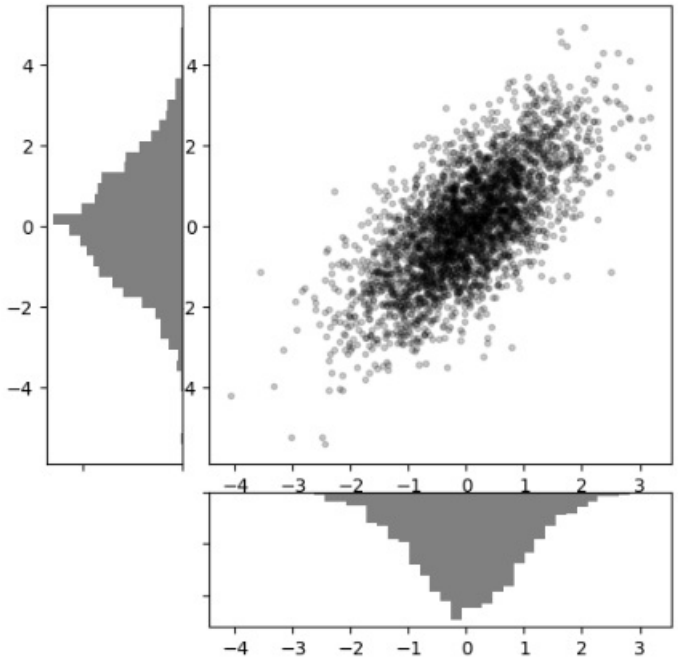
```
plt.subplot(grid[0, 0])
plt.subplot(grid[0, 1:])
plt.subplot(grid[1, :2])
plt.subplot(grid[1, 2])
```



这种灵活的网格排列方式用途十分广泛，接下来我们用它创建多轴频次直方图：

```
# 创建一些正态分布数据
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 3000).T
# 设置坐标轴和网格配置方式
fig = plt.figure(figsize=(6, 6))
grid = plt.GridSpec(4, 4, hspace=0.2, wspace=0.2)
main_ax = fig.add_subplot(grid[:-1, 1:])
y_hist = fig.add_subplot(grid[:-1, 0], xticklabels=[], sharey=main_ax)
x_hist = fig.add_subplot(grid[-1, 1:], yticklabels=[], sharex=main_ax)
# 主坐标轴画散点图
main_ax.plot(x, y, 'ok', markersize=3, alpha=0.2)
# 次坐标轴画频次直方图
x_hist.hist(x, 40, histtype='stepfilled',
orientation='vertical', color='gray')
x_hist.invert_yaxis()
```

```
y_hist.hist(y, 40, histtype='stepfilled',
orientation='horizontal', color='gray')
y_hist.invert_xaxis()
```



4.2.1：配置颜色条

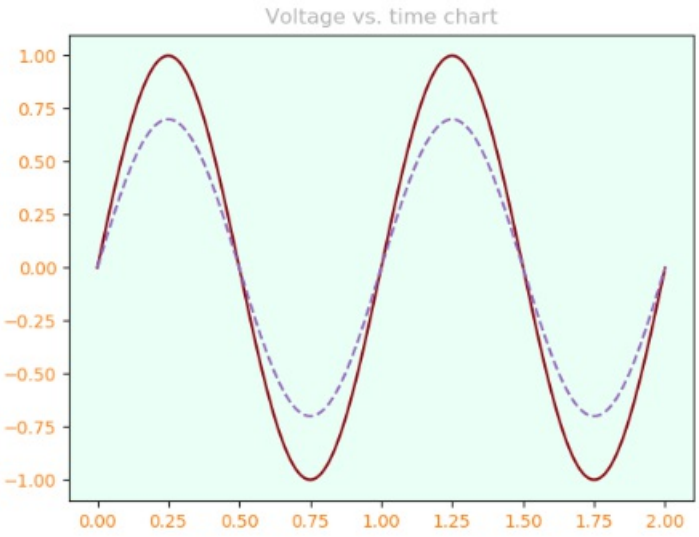
基本颜色演示

`Matplotlib` 提供了 `8` 种指定颜色的方法：

- 在[`0` ， `1`]中的浮点值的 `RGB` 或 `RGBA` 元组（例如 (`0.1` , `0.2` , `0.5`) 或 (`0.1` ， `0.2` ， `0.5` ， `0.3`)）。 `RGBA` 是红色，绿色，蓝色， `Alpha` 的缩写。
- 十六进制 `RGB` 或 `RGBA` 字符串 (例如： `#0F0F0F` 或者 `#0F0F0F0F`)。
- [`0` , `1`]中浮点值的字符串表示，包括灰度级（例如， `0.5` ）。
- 单字母字符串，例如这些其中之一：{ `b` ， `g` ， `r` ， `c` ， `m` ， `y` ， `k` ， `w` }。
- 一个 `X11` / `CSS4` (`html`) 颜色名称，例如： `blue` 。
- 来自 `xkcd` 的颜色调研的名称，前缀为 `xkcd` ：(例如： `xkcd:sky blue`)。
- 一个 `Cn` 颜色规范，即 `C` 后跟一个数字，这是默认属性循环的索引（ `matplotlib.rcParams[axes.prop_cycle]` ）； 索引在艺术家对象创建时发生，如果循环不包括颜色，则默认为黑色。
- 其中一个 { `tab:blue` ， `tab:orange` ， `tab:green` ， `tab:red` ， `tab:purple` ， `tab:brown` ， `tab:pink` ， `tab:gray` ， `tab:olive` ， `tab:cyan` }，它们是 `tab10` 分类调色板中的 `Tableau` 颜色（这是默认的颜色循环）。

示例如下：

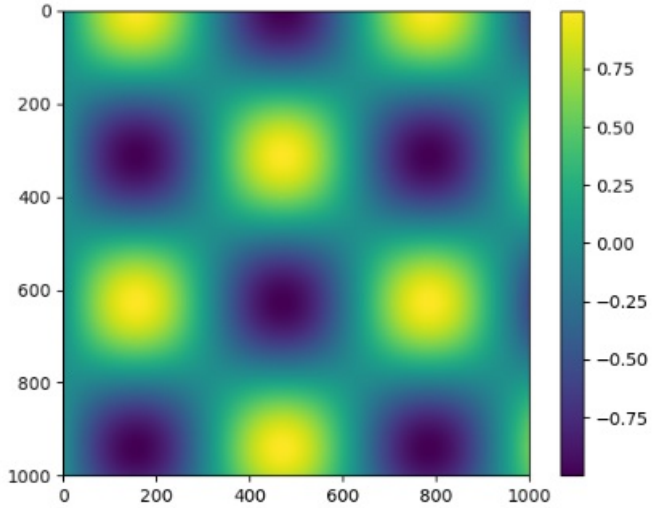
```
t = np.linspace(0.0, 2.0, 201)
s = np.sin(2 * np.pi * t)
fig, ax = plt.subplots(facecolor=(.18, .31, .31))#RGB 元组
ax.set_facecolor('#eafff5')#hex字符串
ax.set_title('Voltage vs. time chart', color='0.7')#灰度字符串
ax.plot(t, s, 'xkcd:crimson')
ax.plot(t, .7*s, color='C4', linestyle='--')#CN颜色选择
ax.tick_params(labelcolor='tab:orange')
```



颜色条

颜色的设置应该是在 `matploblib` 使用最频繁的配置之一了。 `matplotlib` 通过 `cmap` 参数为图形设置颜色条的配色方案：

```
x = np.linspace(0, 10, 1000)
I = np.sin(x) * np.cos(x[:, np.newaxis])
plt.imshow(I, cmap='gray')#采用灰度配色的图形
```



`matplotlib` 所有可用的配色方案都在 `plt.cm` 命名空间中。在 `Ipython` 里通过 `Tab` 键就可以查看所有的配置方案：

```
plt.cm<TAB>
```

选择配色方案

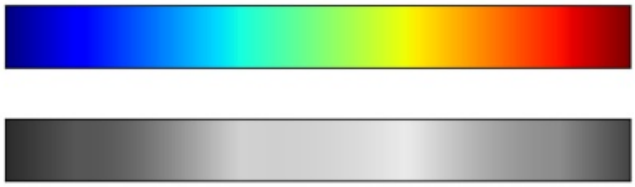
有了这么多能够选择的配色方案只是第一步，重要的是如何确定用那种方案。一般情况下我们只需要关注三种不同的配色方案：

- 顺序配色方案，由一组连续的颜色构成的配色方案（例如 `binary` 或 `viridis` ）。
- 互逆配色方案，通常由两种互补的颜色构成，表示正反两种含义（例如 `RdBu` 或 `PuOr` ）。
- 定性配色方案，随机顺序的一组颜色（例如 `rainbow` 或 `jet` ）。

`jet` 是一种定性配色方案，定性配色方案在对定性数据进行可视化时的选择空间非常有限。随着图形亮度的提高，经常会出现颜色无法区分的问题。接下来将演示几个常用的配色方案。

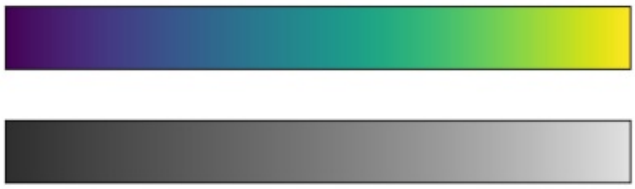
这里通过把 `jet` 转换为黑白的灰度图看看：

```
from matplotlib.colors import LinearSegmentedColormap
import matplotlib.pyplot as plt
import numpy as np
def grayscale_cmap(cmap):

    """为配色方案显示灰度图"""
    cmap = plt.cm.get_cmap(cmap)
    colors = cmap(np.arange(cmap.N))
    # 将RGBA色转换为不同亮度的灰度值
    # 参考链接http://alienryderflex.com/hsp.html
    RGB_weight = [0.299, 0.587, 0.114]
    luminance = np.sqrt(np.dot(colors[:, :3] ** 2, RGB_weight))
    colors[:, :3] = luminance[:, np.newaxis]
    return LinearSegmentedColormap.from_list(cmap.name + "_gray", colors, cmap.N)

def view_colormap(cmap):
    """用等价的灰度图表示配色方案"""
    cmap = plt.cm.get_cmap(cmap)
    colors = cmap(np.arange(cmap.N))
    cmap = grayscale_cmap(cmap)
    grayscale = cmap(np.arange(cmap.N))
    fig, ax = plt.subplots(2, figsize=(6, 2),
    subplot_kw=dict(xticks=[], yticks=[]))
    ax[0].imshow([colors], extent=[0, 10, 0, 1])
    ax[1].imshow([grayscale], extent=[0, 10, 0, 1])


view_colormap('jet')
plt.show()
```



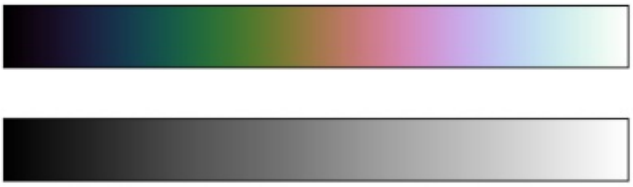
从上图我们观察灰度图里比较亮的那部分条纹。这些亮度变化不均匀的条纹在彩色图中对应某一段彩色区间，由于色彩太接近容易突显出数据集中不重要的部分，导致眼睛无法识别重点。更好的配色方案是 `viridis` ，它采用了精心设计的亮度渐变方式，这样不仅便于视觉观察，而且更清晰：

```
view_colormap('viridis')
```



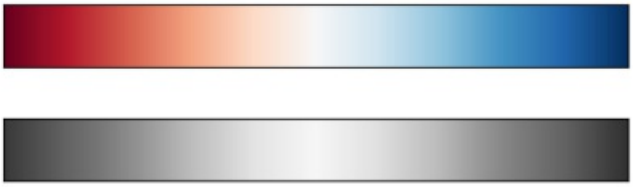
还可以使用 `cubehelix` 实现彩虹效果， `cubehelix` 配色方案可以可视化连续的数值：

```
view_colormap('cubehelix')
```



还有一种可以用两种颜色表示正反两种含义的方案，实现函数为 `RdBu` ：

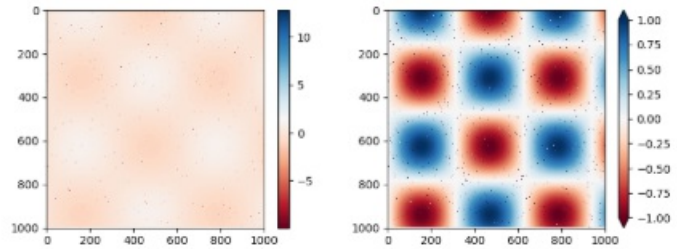
```
view_colormap('RdBu')
```



颜色条刻度的限制与扩展功能的设置

`Matplotlib` 提供了丰富的颜色条配置功能。由于可以将颜色条本身仅看作是一个 `plt.Axes` 实例，因此前面所学的所有关于坐标轴和刻度值的格式配置技巧都可以派上用场。颜色条有一些有趣的特性。例如，我们可以缩短颜色取值的上下限，对于超出上下限的数据，通过 `extend` 参数用三角箭头表示比上限大的数或者比下限小的数。下面展示一张噪点图：

```
x = np.linspace(0, 10, 1000)
I = np.sin(x) * np.cos(x[:, np.newaxis])
speckles = (np.random.random(I.shape) < 0.01)
I[speckles] = np.random.normal(0, 3, np.count_nonzero(speckles))
plt.figure(figsize=(10, 3.5))
plt.subplot(1, 2, 1)
plt.imshow(I, cmap='RdBu')
plt.colorbar()
plt.subplot(1, 2, 2)

plt.imshow(I, cmap='RdBu')
plt.colorbar(extend='both')
plt.clim(-1, 1)
```

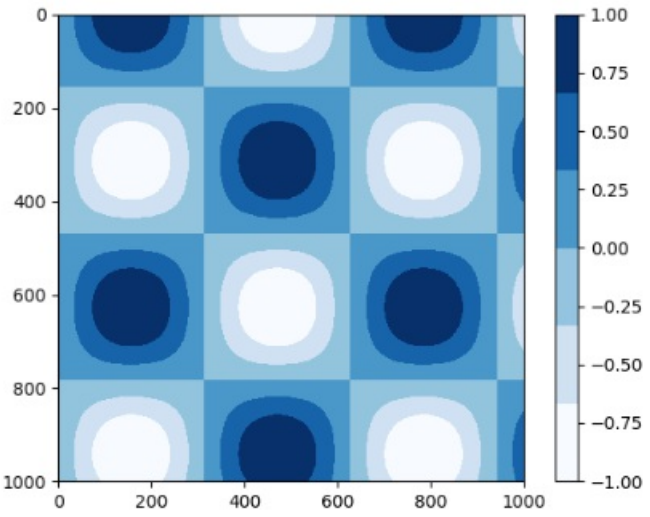


左边的图是用默认的颜色条刻度限制实现的效果，噪点的范围覆盖掉了我们感兴趣的数据。而右边的图形设置了颜色条的刻度上下限，并在上下限之外增加了扩展功能。

离散型颜色条

虽然大多数颜色条默认都是连续的，但有的时候你可能也需要表示离散数据。最简单的做法就是使用 `plt.cm.get_cmap()` 函数，将适当的配色方案的名称以及需要的区间数量传进去即可：

```
x = np.linspace(0, 10, 1000)
I = np.sin(x) * np.cos(x[:, np.newaxis])
plt.imshow(I, cmap=plt.cm.get_cmap('Blues', 6))
plt.colorbar()
plt.clim(-1, 1);
```



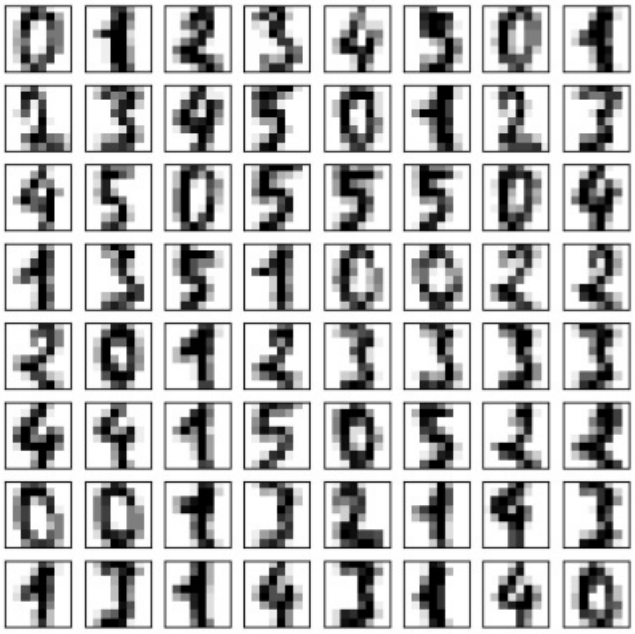
手写数字

接下来我们将学习一个比较实用的案例，手写数字可视化图。数据在 `sklearn` 中，包含近 `2000` 份 `8 x 8` 的手写数字缩略图。

先下载数据，然后使用 `plt.imshow()` 对一些图形进行可视化：

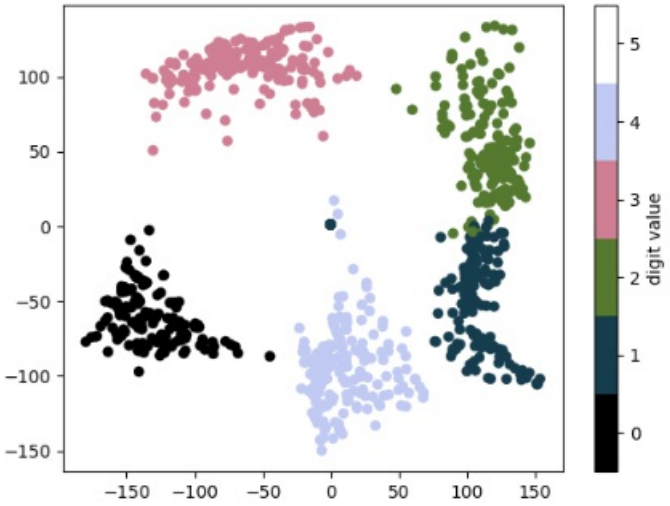
```
from sklearn.datasets import load_digits
digits = load_digits(n_class=6)
fig, ax = plt.subplots(8, 8, figsize=(6, 6))
for i, axi in enumerate(ax.flat):
    axi.imshow(digits.images[i], cmap='binary')
```

```
    axi.set(xticks=[], yticks=[])
```



由于每个数字都由 64 像素的色相构成，因此可以将每个数字看成是一个位于 64 维空间的点，即每个维度表示一个像素的亮度。但是想通过可视化来描述如此高维度的空间是非常困难的。一种解决方案是通过降维技术，在尽量保留数据内部重要关联性的同时降低数据的维度，例如流形学习。下面展示如何用流形学习将这些数据投影到二维空间进行可视化：

```
from sklearn.datasets import load_digits
from sklearn.manifold import Isomap
iso = Isomap(n_components=2)
digits = load_digits(n_class=6)
projection = iso.fit_transform(digits.data)
plt.scatter(projection[:, 0], projection[:, 1], lw=0.1,
c=digits.target, cmap=plt.cm.get_cmap('cubehelix', 6))
plt.colorbar(ticks=range(6), label='digit value')
plt.clim(-0.5, 5.5)
```



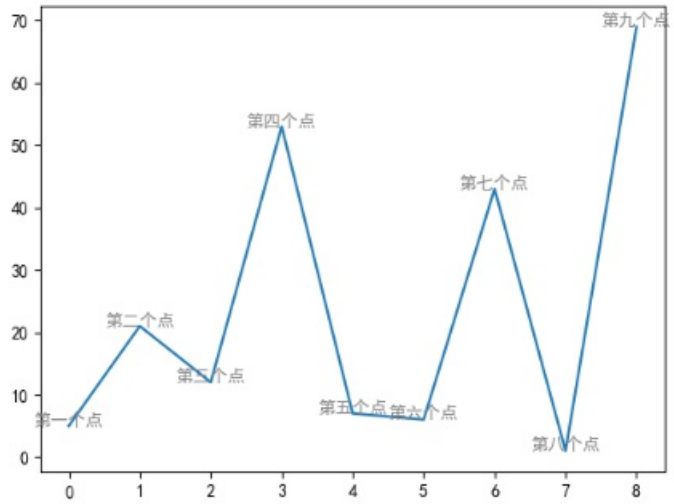
我们使用了离散型颜色条来显示结果，调整 `ticks` 和 `clim` 参数来改善颜色条。这个结果向我们展示了一些数据集的有趣特性。比如数字 `5` 与数字 `3` 在投影中有大面积重叠，说明一些手写的 `5` 与 `3` 难以区分，因此自动分类算法也更容易搞混它们。其它的数字，像数字 `0` 与数字 `1` ，隔得特别远，说明两者不太可能出现混淆。

4.2.2：设置注释

添加注释

为了使我们的可视化图形让人更加容易理解，在一些情况下辅之以少量的文字提示和标签可以更恰当地表达信息。 `matplotlib` 可以通过 `plt.txt` / `ax.txt` 命令手动添加注释，它们可以在具体的 `x/y` 坐标上放文字：

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import matplotlib as mpl
plt.rcParams['font.sans-serif']=['simhei']
plt.rcParams['font.family']='sans-serif'
plt.rcParams['axes.unicode_minus']=False#画图可中文
style = dict(size=10, color='gray')
#绘制图形
plt.plot([5,21,12,53,7,6,43,1,69])
#设置注释
plt.text(0, 5, "第一个点",ha='center', **style)
plt.text(1, 21, "第二个点",ha='center', **style)
plt.text(2, 12, "第三个点",ha='center', **style)
plt.text(3, 53, "第四个点",ha='center', **style)
plt.text(4, 7, "第五个点",ha='center', **style)
plt.text(5, 6, "第六个点",ha='center', **style)
plt.text(6, 43, "第七个点",ha='center', **style)
plt.text(7, 1, "第八个点",ha='center', **style)
plt.text(8, 69, "第九个点",ha='center', **style)
plt.show()
```



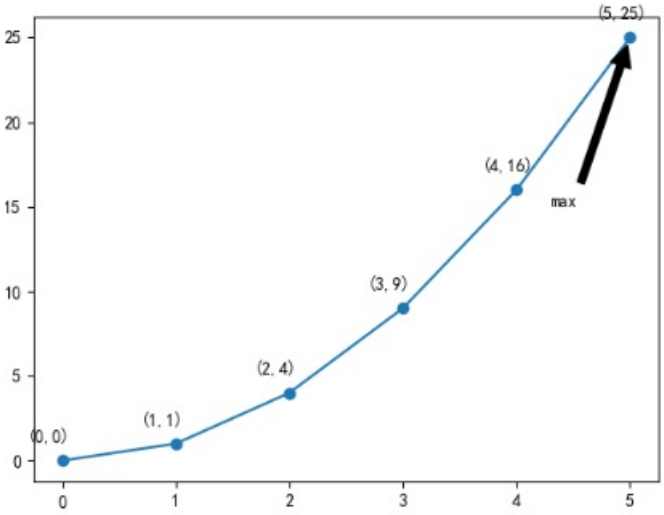
`plt.text` 方法需要一个 `x` 轴坐标、一个 `y` 轴坐标、一个字符串和一些可选参数，比如文字的颜色、字号、风格、对齐方式等其它文字属性。这里用了 `ha="center"` ， `ha` 是设置水平对齐方式。其他常用参数如下：

1. `fontsize` 设置字体大小，默认 `12` ，可选参数 `xx-small，x-small，small，medium，large，x-large，xx-large`
2. `fontweight` 设置字体粗细，可选参数 `light，normal，medium，semibold，bold，heavy，black` 。

3. `fontstyle` 设置字体类型，可选参数 `normal，italic，oblique]` 。
4. `verticalalignment` 设置垂直对齐方式，可选参数：`center，top，bottom，baseline` 。
5. `horizontalalignment` 设置水平对齐方式，可选参数： `left，right，center` 。
6. `rotation` (旋转角度)可选参数为: `vertical，horizontal` 也可以为数字,数字代表旋转的角度。
7. `alpha` 透明度，参数值0至1之间。
8. `backgroundcolor` 标题背景颜色。

还可以通过 `plt.annotate()` 函数来设置注释，该函数既可以创建文字，也可以创建箭头，而且它创建的箭头能够进行非常灵活的配置。

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 6)
y = x * x
plt.plot(x, y, marker='o')
for xy in zip(x, y):
    plt.annotate("(%s,%s)" % xy, xy=xy, xytext=(-20, 10), textcoords='offset points')
plt.annotate('max', xy=(5, 25), xytext=(4.3, 15),arrowprops=dict(facecolor='black', shrink=0.05))
plt.show()
```



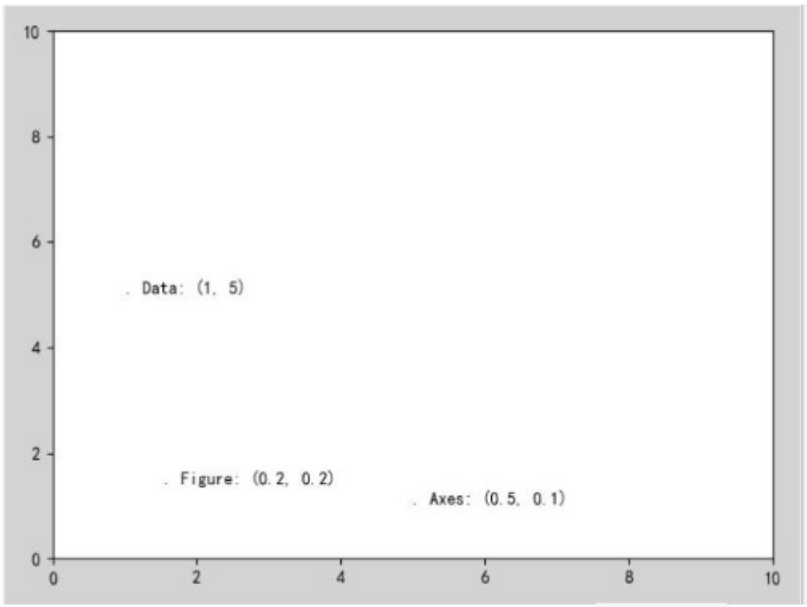
这里设置了两个注释，一个显示了点的坐标，另一个显示了最高点描述 `max` 。 `annotate` 的第一个参数为注释文本字符串，第二个为被注释的坐标点，第三个为注释文字的坐标位置。

文字、坐标变换

上面的例子将文字注释放在目标数据的位置上，有的时候可能需要将文字放在与数据无关的位置上，比如坐标轴或图形， `matplotlib` 中通过调整坐标变换 `transorm` 参数来实现：

```
fig, ax = plt.subplots()
ax.axis([0, 10, 0, 10])
ax.text(1, 5, ". Data: (1, 5)", transform=ax.transData)
ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)", transform=ax.transAxes)
```

```
ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure)

plt.show()
```



默认情况下，图中的字符是在其对应的坐标位置。通过设置 `transform` 参数修改位置， `transData` 坐标用 `x,y` 的标签作为数据坐标； `transAxes` 坐标以白色矩阵左下角为原点，按坐标轴尺寸的比例呈现坐标； `transFigure` 坐标类似以灰色矩阵左下角为原点，按坐标轴尺寸的比例呈现坐标。

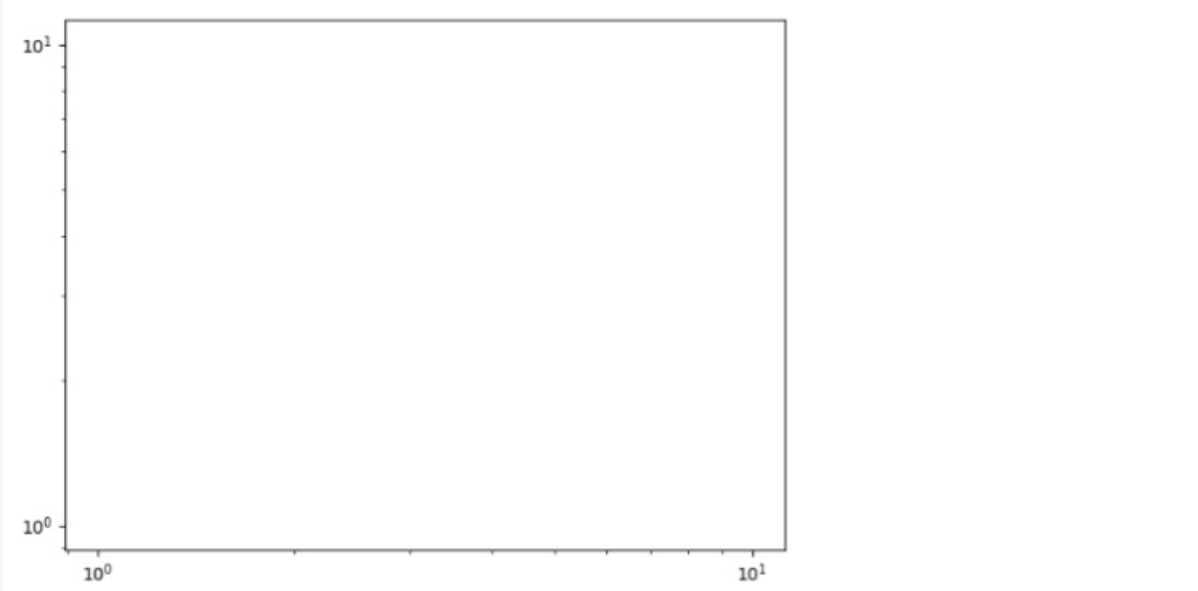
4.2.3：自定义坐标刻度

主次要刻度

学习前最好先对 `matplotlib` 图形的对象层级有深入了解。 `matplotlib` 的 `figure` 对象是一个盛放图形元素的包围盒。可以将每个 `matplotlib` 对象都看成是子对象的容器，每个 `figure` 都包含 `axes` 对象，每个 `axes` 对象又包含其他表示图形内容的对象，比如 `xaxis/yaxis` ，每个属性包含构成坐标轴的线条、刻度和标签的全部属性。

每一个坐标轴都有主次要刻度，主要刻度要比次要刻度更大更显著，而次要刻度往往更小。

```
import matplotlib.pyplot as plt
import numpy as np

ax = plt.axes(xscale='log', yscale='log')
plt.show()
```



可以看到主要刻度都显示为一个较大的刻度线和标签，而次要刻度都显示为一个较小的可读性，不显示标签。

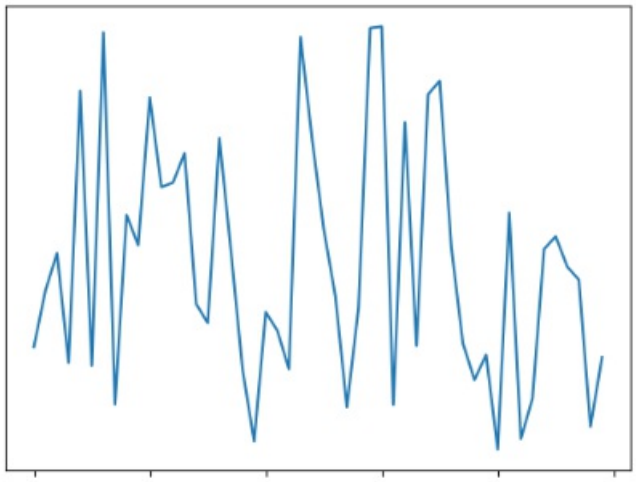
隐藏刻度与标签

最常用的刻度/标签格式化操作可能就是隐藏刻度与标签了，可以通过 `plt.NullLocator()` 和 `plt.NullFormatter()` 实现。

示例如下：

```
ax = plt.axes()
ax.plot(np.random.rand(50))
ax.yaxis.set_major_locator(plt.NullLocator())
```

```
ax.xaxis.set_major_formatter(plt.NullFormatter())
plt.show()
```



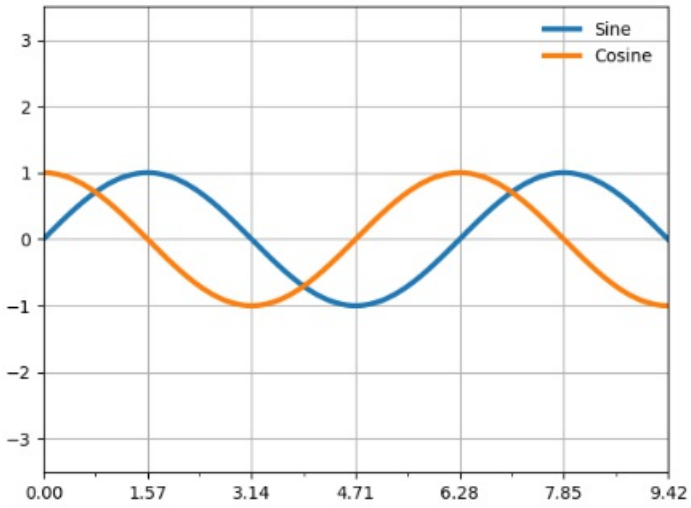
这里 `x` 轴的标签隐藏了但是保留了刻度线， `y` 轴的刻度和标签都隐藏了。有的图片中都不需要刻度线，比如下面这张包含人脸的图形：

```
fig, ax = plt.subplots(5, 5, figsize=(5, 5))
fig.subplots_adjust(hspace=0, wspace=0)
# 从scikit-learn获取一些人脸照片数据
from sklearn.datasets import fetch_olivetti_faces
faces = fetch_olivetti_faces().images
for i in range(5):
    for j in range(5):
        ax[i, j].xaxis.set_major_locator(plt.NullLocator())
        ax[i, j].yaxis.set_major_locator(plt.NullLocator())
        ax[i, j].imshow(faces[10 * i + j], cmap="bone")
plt.show()
```

花哨的刻度格式

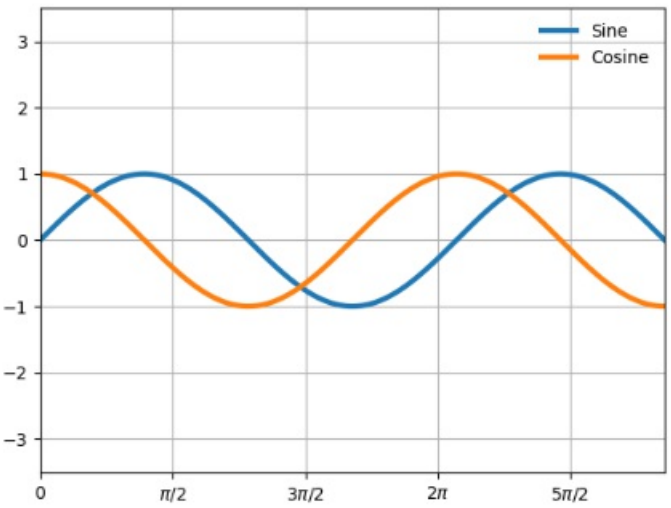
`matplotlib` 默认的刻度格式可以满足大部分的需求。虽然默认配置已经很不错了，但是有时候可能需要更多的功能，比如正弦曲线和余弦曲线,默认情况下刻度为整数，如果将刻度与网格线画在 `π` 的倍数上图形会更加自然，可以通过设置一个 `multipleLocator` 来实现将刻度放在你提供的数值倍数上：

```
fig, ax = plt.subplots()
x = np.linspace(0, 3 * np.pi, 1000)
ax.plot(x, np.sin(x), lw=3, label='Sine')
ax.plot(x, np.cos(x), lw=3, label='Cosine')
# 设置网格、图例和坐标轴上下限
ax.grid(True)
ax.legend(frameon=False)
ax.axis('equal')
ax.set_xlim(0, 3 * np.pi)

ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
ax.xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 4))

plt.show()
```



`matplotlib` 还支持用数学符号来做刻度，在数学表达式两侧加上美元符号 `$`，这样就可以方便地显示数学符号和数学公式。可以用 `plt.FuncFormatter` 来实现，用一个自定义函数设置不同刻度标签的显示：

```
def format_func(value, tick_number):
# 找到π/2的倍数刻度
    N = int(np.round(2 * value / np.pi))
    if N == 0:
        return "0"
    elif N == 1:
        return r"$\pi/2$"
    elif N == 2:
        return r"$\pi$"
    elif N % 2 > 0:
        return r"${0}\pi/2$".format(N)
    else:
        return r"${0}\pi$".format(N // 2)
ax.xaxis.set_major_formatter(plt.FuncFormatter(format_func))
```



格式生成器与定位器

前面已经介绍了一些格式生成器和定位器，这里再用表格简单总结一些内置的格式生成器和定位器：

定位器	描述
NullLocator	无刻度
FixedLocator	刻度位置固定
IndexLocator	用索引作为定位器
LinearLocator	从min 到max 均匀分布刻度
LogLocator	从min 到max 按对数分布刻度
MultipleLocator	刻度和范围都是基数的倍数
MaxNLocator	为最大刻度找到最优位置
AutoMinorLocator	次要刻度的定位器

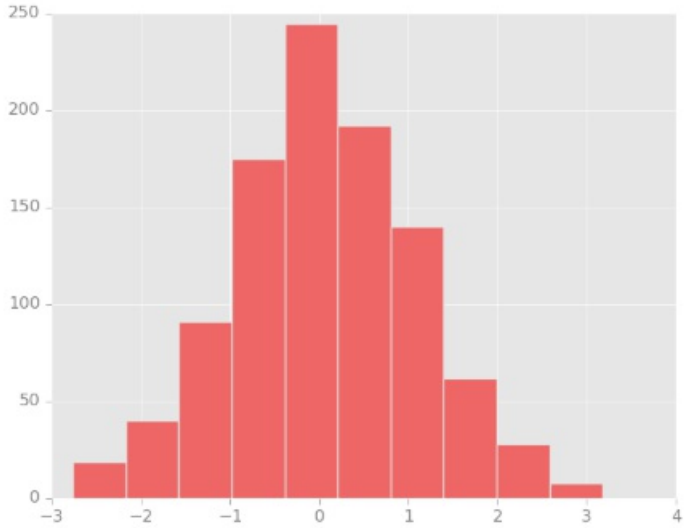
格式生成器	描述
NullFormatter	刻度上无标签
IndexFormatter	将一组标签设置为字符串
FixedFormatter	手动为刻度设置标签
FuncFormatter	用自定义函数设置标签
FormatStrFormatter	为每个刻度值设置字符串格式
ScalarFormatter	为标量值设置标签
LogFormatter	对数坐标轴的默认格式生成器

4.2.4：配置文件与样式表

配置图形

我们可以通过修个单个图形配置，使得最终图形比原来的图形更好看。可以为每个单独的图形进行个性化设置。这里我们通过手动调整，将 `matplotlib` 土到掉渣的默认直方图修改成美图:

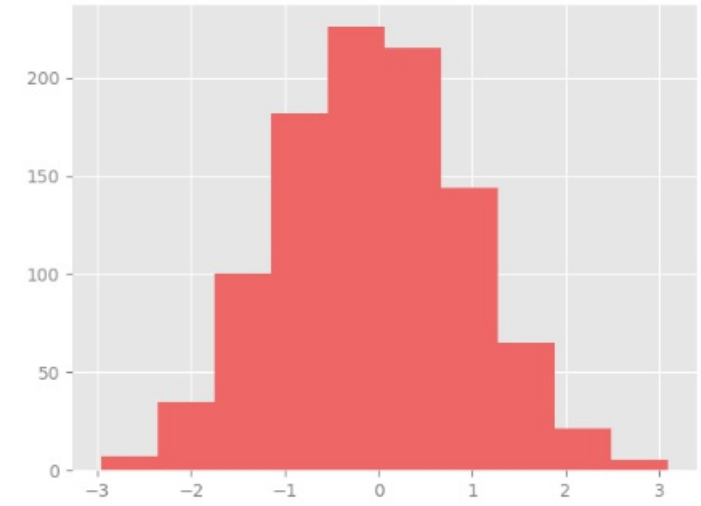
```
import matplotlib.pyplot as plt
plt.style.use('classic')
import numpy as np

x = np.random.randn(1000)
plt.hist(x);

# 用灰色背景
ax = plt.axes(facecolor='#E6E6E6')
ax.set_axisbelow(True)

# 画上白色的网格线
plt.grid(color='w', linestyle='solid')

# 隐藏坐标轴的线条
for spine in ax.spines.values():
    spine.set_visible(False)
ax.xaxis.tick_bottom()
ax.yaxis.tick_left()

# 弱化刻度与标签
ax.tick_params(colors='gray', direction='out')
for tick in ax.get_xticklabels():
    tick.set_color('gray')
for tick in ax.get_yticklabels():
    tick.set_color('gray')

# 设置频次直方图轮廓色与填充色
ax.hist(x, edgecolor='#E6E6E6', color='#EE6666')
plt.show()
```

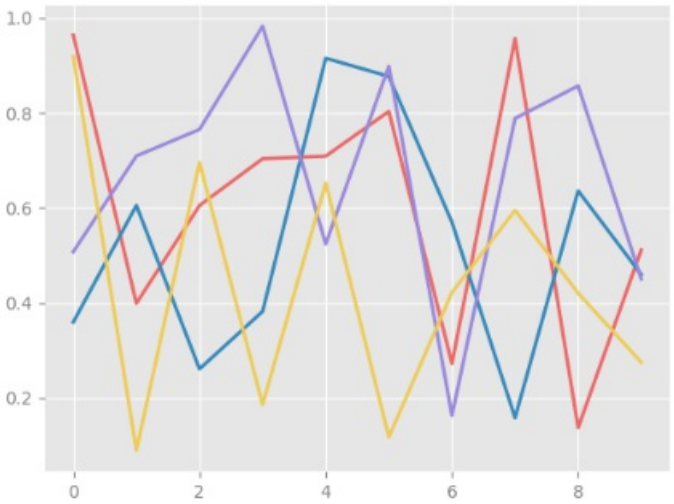


修改默认配置rcParams

通过手动配置确实能达到我们想要的效果，但是如有很多个图形，我们肯定不希望对每一个图都这样手动配置一番。 `matplotlib` 作为一个强大的工具当然有方法可以让我们只配置一次默认图形，就可以应用到所有图形上。这个方法就是通过修改默认配置 `rcParams` 。 `matplotlib` 在每次加载的时候，都会定义一个运行时配置 `rc` ，其中包含了我们创建的图形元素的默认风格。

```
from matplotlib import cycler
import matplotlib.pyplot as plt
import numpy as np

colors = cycler('color',['#EE6666', '#3388BB', '#9988DD','#EECC55', '#88BB44', '#FFBBBB'])

plt.rc('axes', facecolor='#E6E6E6', edgecolor='none',axisbelow=True, grid=True, prop_cycle=colors)

plt.rc('grid', color='w', linestyle='solid')

plt.rc('xtick', direction='out', color='gray')

plt.rc('ytick', direction='out', color='gray')

plt.rc('patch', edgecolor='#E6E6E6')

plt.rc('lines', linewidth=2)

x = np.random.randn(1000)

plt.hist(x)#画直方图
plt.show()

for i in range(4):
    plt.plot(np.random.rand(10))#折线图
plt.show()
```





所有 `rc` 设置都存储在一个名为 `matplotlib.rcParams` 的类字典变量中，可以通过这个变量来查看我们的配置。 `rc` 的第一个参数是希望自定义的对象，如 `figure` 、 `axes` 、 `grid` 等。其后可以跟上一系列的关键字参数。

样式表

`matplotlib` 从 `1.4` 版本中增加了一个非常好用的 `style` 模块，里面包含了大量的新式默认样式表，还支持创建和打包自己的风格。通过 `plt.style.available` 命令可以看到所有可用的风格。

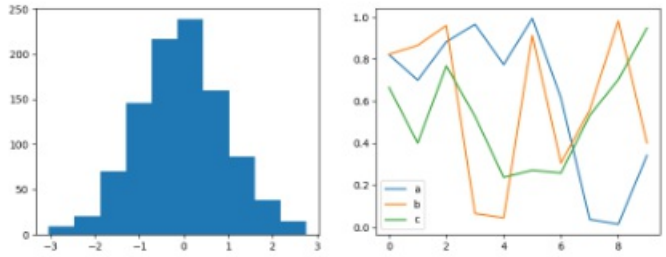
```
 plt.style.available[:5]#查看前5个风格样式
'''
输出:['bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight']
'''
```

使用某种样式表的基本方法为 `plt.style.use('stylename')` ，这样就改变后面代码的所有风格。支持组合样式，通过传递样式列表可以轻松组合这些样式。如果需要，也可以使用风格上下文管理器临时更换风格：

```
with plt.style.context('stylename'):
    make_a_plot()
```

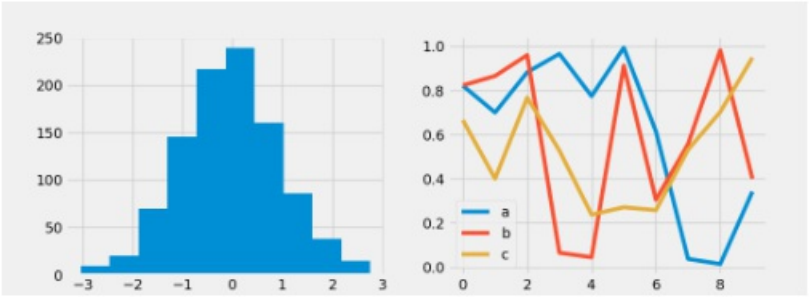
首先创建一个画两种基本图形的函数:

```
def hist_and_lines():
    np.random.seed(0)
    fig, ax = plt.subplots(1, 2, figsize=(11, 4))
    ax[0].hist(np.random.randn(1000))
    for i in range(3):
        ax[1].plot(np.random.rand(10))
        ax[1].legend(['a', 'b', 'c'], loc='lower left')
    plt.show()
```



再通过修改风格绘制图形：

```
with plt.style.context('fivethirtyeight'):
    hist_and_lines()
```

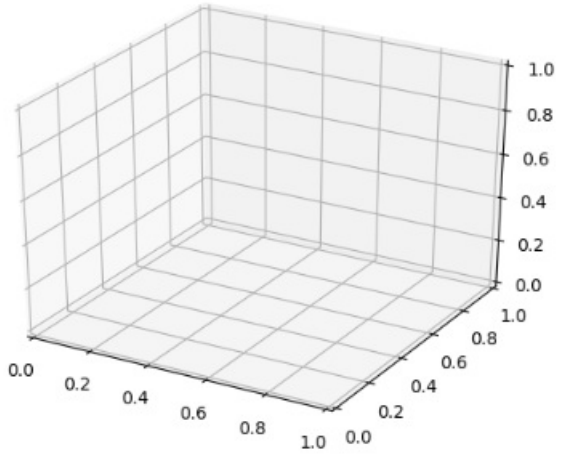


4.2.5：绘制三维图

matplotlib画三维图

要画三维图需要先导入 `from mpl_toolkits import mplot3d` 。导入这个子模块后，就可以在创建任意一个普通坐标轴的过程中添加 `projection='3d'` 参数，从而创建一个三维坐标轴。三维图的优点是在 `notebook` 中可以交互浏览。

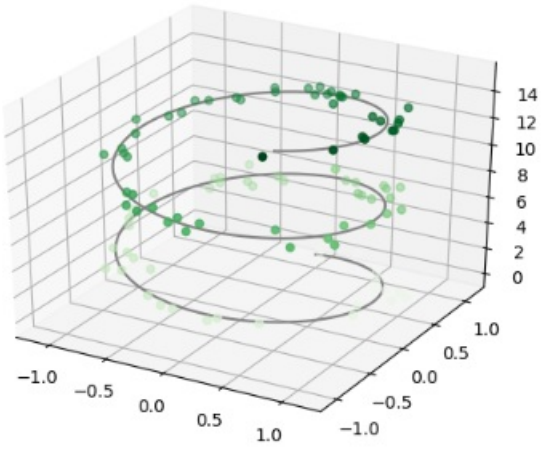
```
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt

fig = plt.figure()
ax = plt.axes(projection='3d')
plt.show()
```



最基本的三维图是由 `(x,y,z)` 三维坐标点构成的线图与散点图。与之前普通二维图类似，可以用 `ax.plot3D` 与 `ax.scatter3D` 函数来创建。不仅创建方式类似，三维图函数的参数也和二维图函数的参数基本相同。 下面来画一个三角螺旋线并在线上随机分布一些散点：

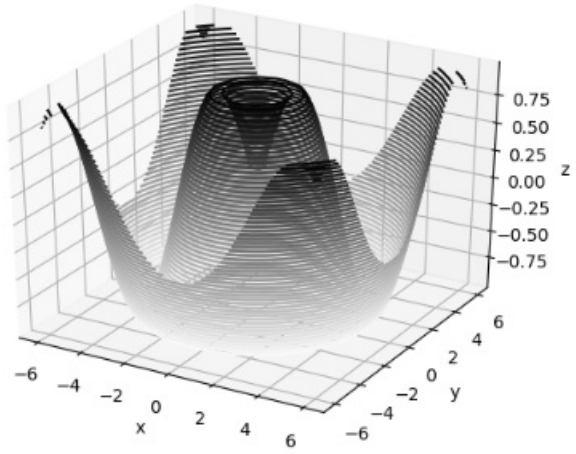
```
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
import numpy as np

ax = plt.axes(projection='3d')

# 三维线的数据
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')

# 三维散点的数据
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
```

```
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');

plt.show()
```



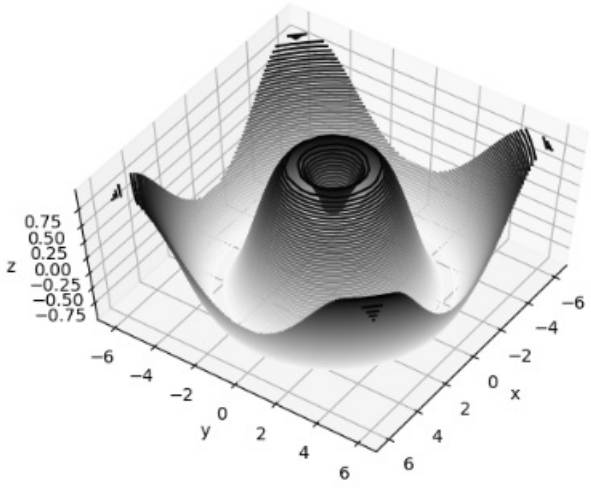
与二维 `ax.contour` 图形一样， `ax.contour3D` 要求所有数据都是二维网格数据的形式，并且由函数计算 `z` 轴数值。下面用三维正弦函数画三维等高线图：

```
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))

x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');

plt.show()
```



默认的初始显示角度有时不是最优的， `matplotlib` 提供了 `view_init` 可以调整观察角度与方位角。下面我们把俯仰角调整为 `60` 度(`x-y` 平面的旋转角度)，方位角调整为 `35` 度(绕 `z` 轴顺时针旋转 `35` 度)。

```
ax.view_init(60,35)
plt.show()
```

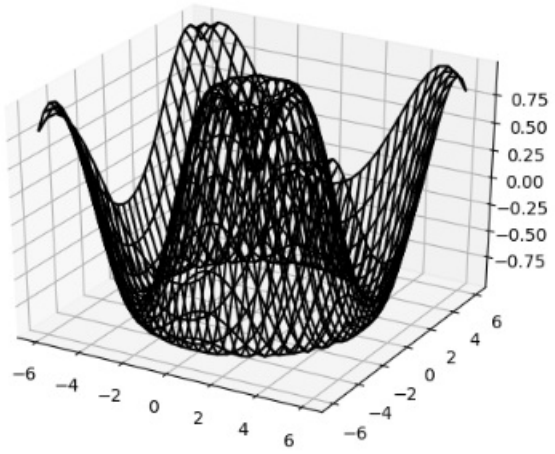


其实，也可以在 `matplotlib` 的交互式后端界面直接通过点击、拖拽图形，实现同样的交互旋转效果。

线框图和曲面图

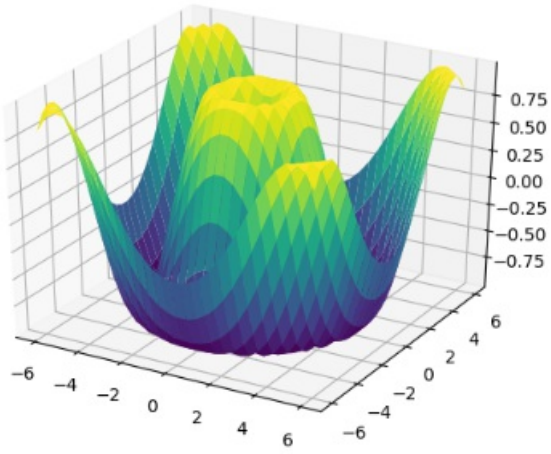
接下来我们将学习线框图和曲面图。它们都是将网格数据映射成三维曲面，得到的三维形状非常容易可视化：

```
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
fig = plt.figure()

#绘制线框图
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
plt.show()
```



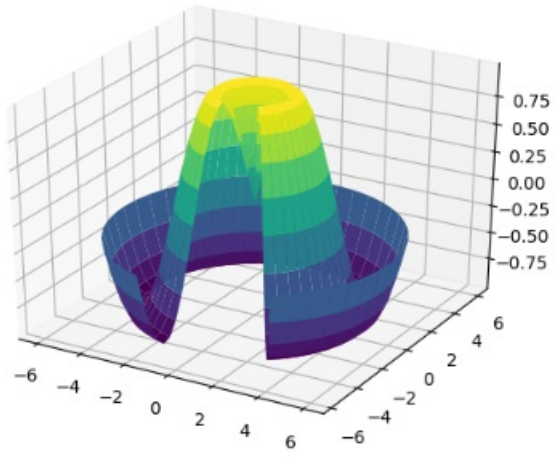
曲面图和线框图类似，只不过线框图的每个面都是由多边形构成的。需要注意的是，画曲面图需要二维数据，但可以不是直角坐标系。

```
#绘制曲面图
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
cmap='viridis', edgecolor='none')
plt.show()
```



下面创建一个局部的极坐标网络，当我们把它画成 `surface3D` 图形时，可以获得一种使用了切片的可视化效果：

```
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))

r = np.linspace(0, 6, 20)
theta = np.linspace(-0.9 * np.pi, 0.8 * np.pi, 40)
r, theta = np.meshgrid(r, theta)

X = r * np.sin(theta)
Y = r * np.cos(theta)
Z = f(X, Y)
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
cmap='viridis', edgecolor='none')
plt.show()
```

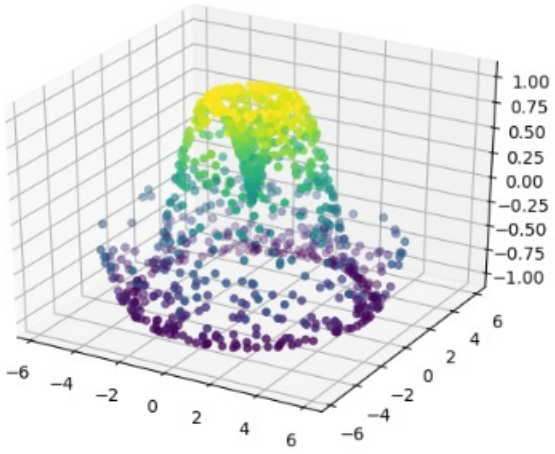

0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−6
−4
−2
0
2
4
6
−6
−4
−2
0
2
4
6

4.2.6：曲面三角剖分

曲面三角剖分

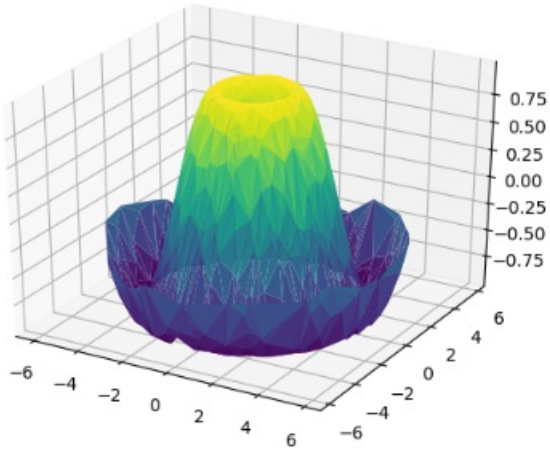
在某些应用的场景中，之前那些些要求均匀采样的网格数据显得太过严格且不太容易实现。这时就可以使用三角剖分图形了。

```
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))
theta = 2 * np.pi * np.random.random(1000)
r = 6 * np.random.random(1000)
x = np.ravel(r * np.sin(theta))
y = np.ravel(r * np.cos(theta))
z = f(x, y)
ax = plt.axes(projection='3d')

ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5)
plt.show()
```



可以看到图形中还有许多地方需要修补，这些工作可以由 `ax.plot_trisurf` 函数完成。它首先找到一组所有点都连接起来的三角形，然后用这些三角形创建曲面。

```
ax = plt.axes(projection='3d')
ax.plot_trisurf(x, y, z,cmap='viridis', edgecolor='none');
```



虽然结果没有之前用均匀网格画的图完美，但是这种三角剖分方法很灵活，可以创建各种有趣的三维图。

莫比乌斯带

莫比乌斯带是把一根纸条扭转 `180` 度后，再把两头粘起来做成的纸带圈。从拓扑学的角度看，莫比乌斯带非常神奇，因为它总共只有一个面！接下来让我们用 `matplotlib` 的三维功能来画一条莫比乌斯带。绘制的关键是想出它的绘图参数：由于它是一条二维带，因此需要两个内在维度。让我们把一维度定义为 `θ`，取值范围为 `0~2 π`；另一个维度是 `w`，取值范围是 `-1~1`，表示莫比乌斯带的宽度：

```
theta = np.linspace(0, 2 * np.pi, 30)
w = np.linspace(-0.25, 0.25, 8)
w, theta = np.meshgrid(w, theta)
```

有了参数之后，我们必须确定带上每个点的直角坐标（ `x`， `y`， `z` ）。仔细思考一下，我们可能会找到两种旋转关系：一种是圆圈绕着圆心旋转（角度用 `θ` 定义），另一种是莫比乌斯带在自己的坐标轴上旋转（角度用 `Φ` 定义）。因此，对于一条莫比乌斯带，我们必然会有环的一半扭转 `180` 度，即 `Δ Φ = Δ θ / 2`。

```
phi = 0.5 * theta
```

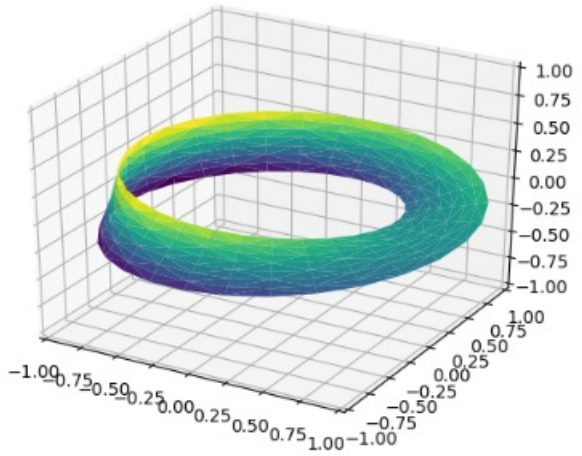
现在用我们的三角学知识将极坐标转换成三维直角坐标。定义每个点到中心的距离（半径） `r`，那么直角坐标（ `z`， `y`， `z` ）就是：

```
r = 1 + w * np.cos(phi)
x = np.ravel(r * np.cos(theta))
y = np.ravel(r * np.sin(theta))
z = np.ravel(w * np.sin(phi))
```

最后，要画出莫比乌斯带，还必须确保三角剖分是正确的。最好的实现方法就是首先用基本参数化方法定义三角剖分，然后用 `Matplotlib` 将这个三角剖分映射到莫比乌斯带的三维空间里，这样就可以画出图形：

```
from matplotlib.tri import Triangulation
tri = Triangulation(np.ravel(w), np.ravel(theta))
ax = plt.axes(projection='3d')
ax.plot_trisurf(x, y, z, triangles=tri.triangles,
```

```
cmap='viridis', linewidths=0.2);
ax.set_xlim(-1, 1); ax.set_ylim(-1, 1); ax.set_zlim(-1, 1);
```



第二部分：机器学习算法与综合实战

作为人工智能的重要组成部分，机器学习也成了炙手可热的概念。在这一部分中，将会从机器学习的一些基础概念到算法再到综合实战演练，一步一步地引导读者学会机器学习的相关知识，以及在怎样使用机器学习技术解决实际问题。准备好了吗？Let’s Go！

第五章：机器学习

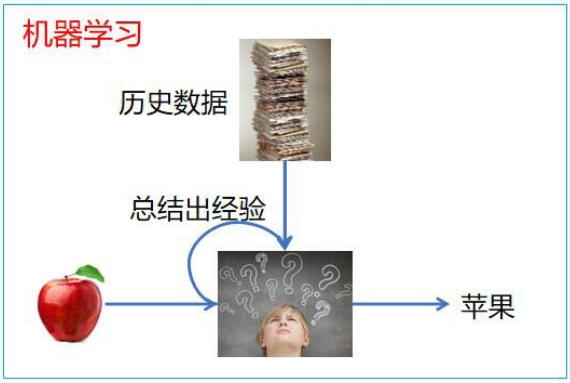
近年来，全球新一代信息技术创新浪潮迭起。作为全球信息领域产业竞争的新一轮焦点，人工智能的发展也迎来了第三次浪潮，它正在推动工业发展进入新的阶段，掀起第四次工业革命的序幕。而作为人工智能的重要组成部分，机器学习也成了炙手可热的概念。本章将向您介绍机器学习的基础知识，为后面的学习打好基础。

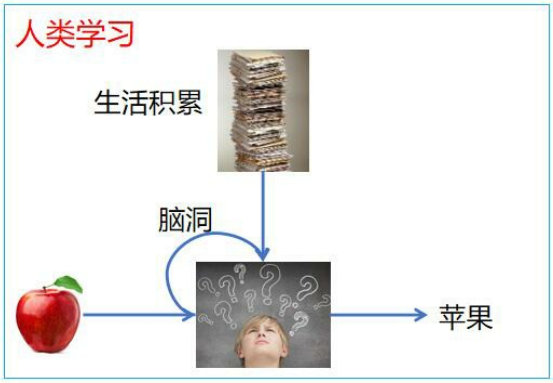
5.1.1 什么是机器学习

机器学习的定义有很多种，但是最准确的定义是："A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." 这个定义除了非常押韵之外，还体现了机器学习的几个关键点，即：task , experience 和 performance 。

task 即机器学习需要解决的问题，experience 即机器学习程序为了解决问题，从历史数据中总结出来的规律， performance 即机器学习程序总结出来的规律是好还是坏。

其实机器学习与人类学习是非常类似的。机器学习会从历史数据中不断地总结，归纳出规律得出数学模型，然后当有问题输入到机器学习程序时，机器学习程序能根据学到地数学模型进行预测，计算出预测结果。人类学习则是根据从小到大的耳濡目染和生活经验的积累，在大脑中形成了潜意识，这种潜意识其实就是我们人类的生活经验和规律。当我们碰到一些没见过的事物时，我们能够根据潜意识来对新事物有一个认知。



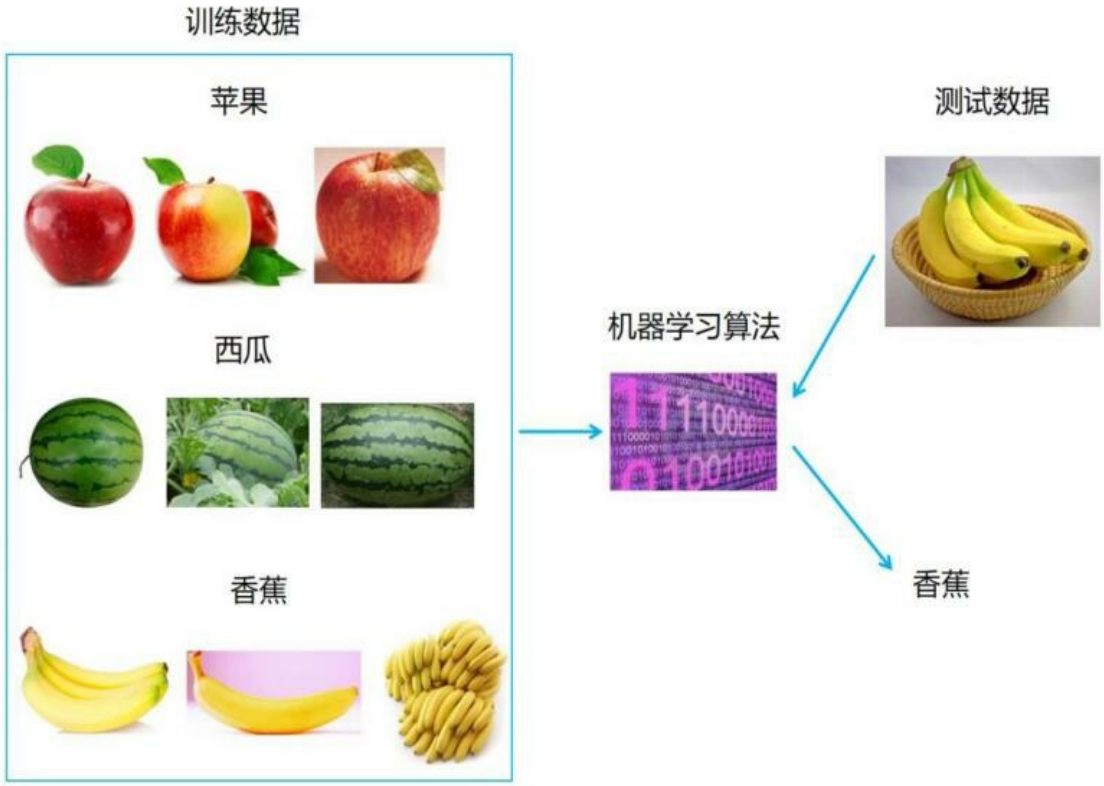


5.1.2 机器学习的主要任务

机器学习能够完成的任务主要有：分类、回归、聚类。

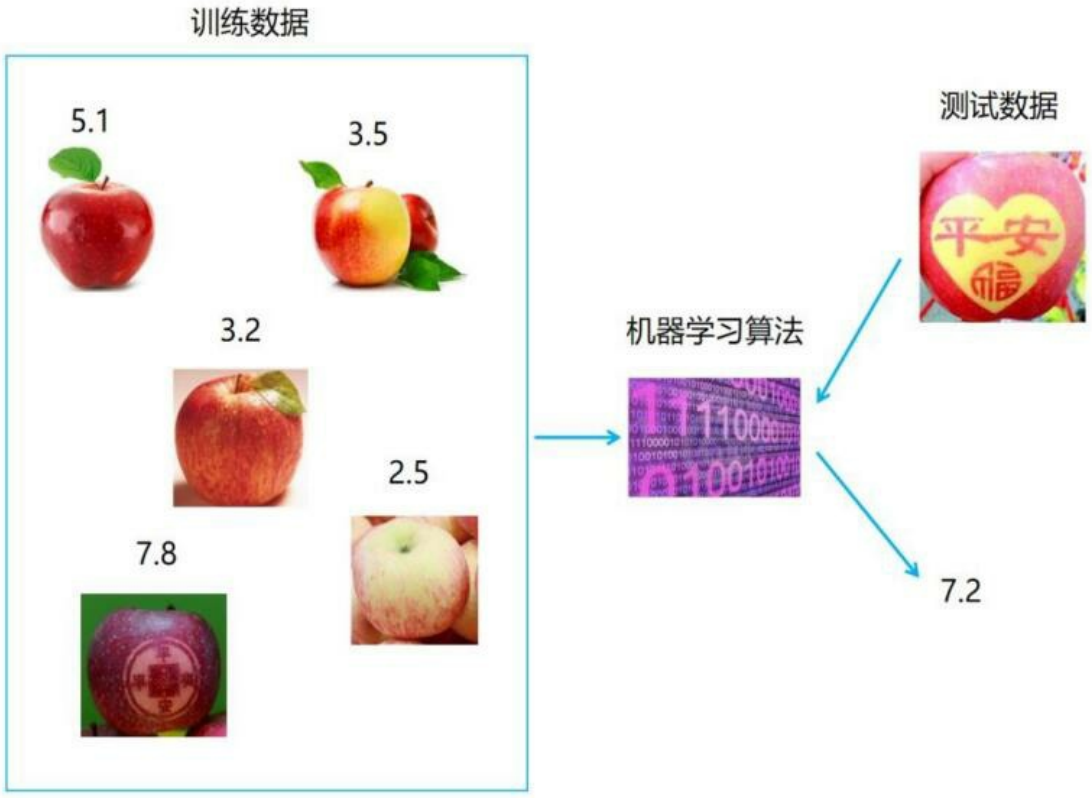
分类

假如现在有一些苹果、西瓜和香蕉的图片作为训练集(有标签)，现在想要机器学片来分辨出该图片中的是苹果、西瓜还是香蕉。像这样的任务称为分类任务。



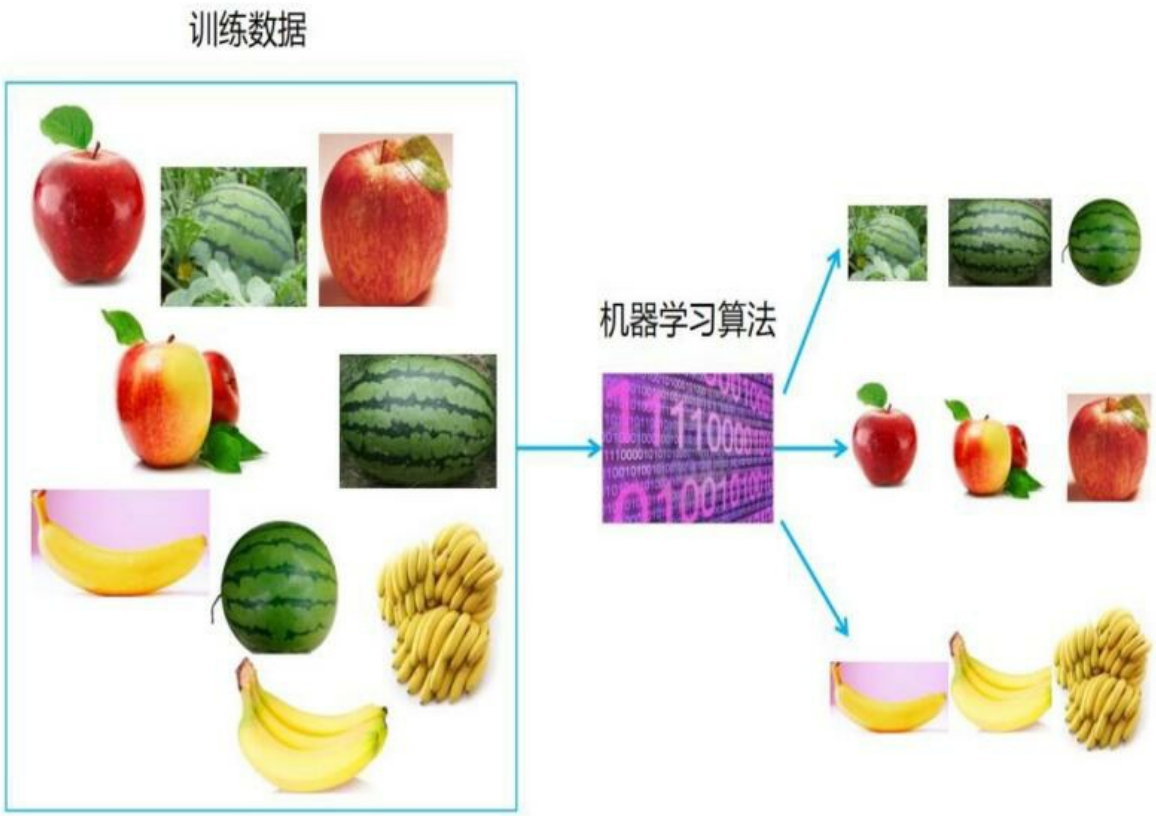
回归

假如现在有一些苹果的售价数据作为训练集(有标签)，现在想要机器学习算法能够根据新的测试图片来分辨出该图片中的苹果能卖多少钱。像这样的任务称为回归任务。



聚类

假如现在有一些水果的图片作为训练集(无标签)，现在想要机器学习算法能够根据训练集中的图片将这些图片进行归类，但是并不知道这些类别是什么。像这样的任务称为聚类任务。

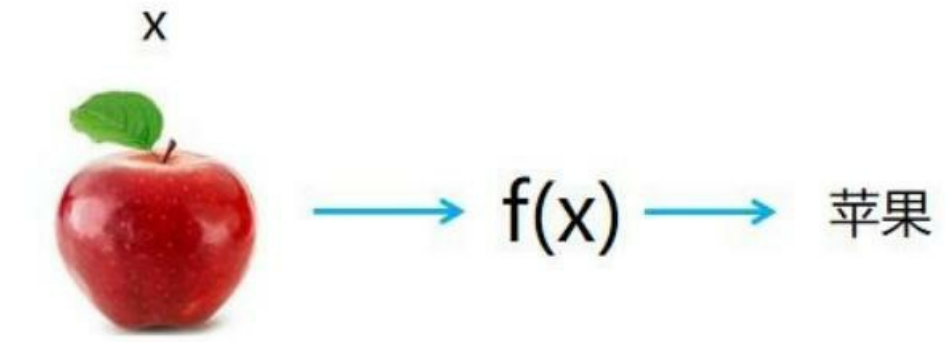


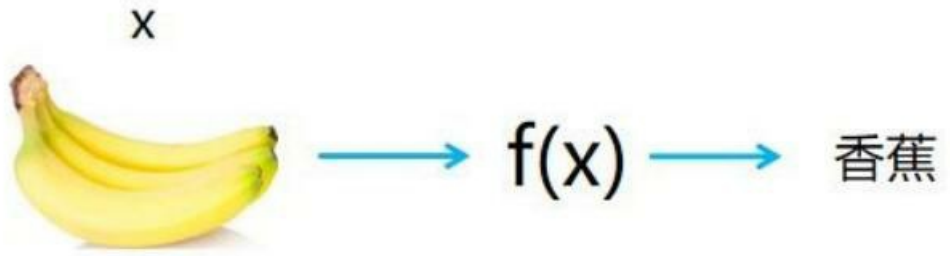
细心的你可能注意到了，分类和回归问题的训练集中都是带有标签的。也就是说数据已经告诉了机器学习算法我这条数据的答案是这个，那条数据的答案是那个，就像有老师在监督学生做题目一样，一看到学生做错了就告诉他题目做错了，看到学生做对了就鼓励他。所以用来解决分类和回归问题的机器学习算法又称为监督学习。而像用来解决聚类问题的机器学习算法又称为无监督学习。

5.1.3 机器学习常见术语

模型

根据历史数据总结归纳出规律的过程，即学习过程，或模型的训练过程。模型这个词看上去很高大上，其实可以把他看成是一个函数。例如：现在想用机器学习来识别图片里的是香蕉还是苹果，那么机器学习所的事情就是得到一个比较好的函数，当输入一张香蕉图片时，能得到识别结果为香蕉的输 出，当输入一张苹果图片时，能得到识别结果为苹果的输出。





训练集，测试集，样本，特征

假设收集了一份西瓜数据：

色泽	纹理	声音	甜不甜
青绿	清晰	清脆	不甜
青绿	模糊	浑浊	甜
乌黑	清晰	清脆	不甜
乌黑	模糊	浑浊	甜

并假设现在已经使用机器学习算法根据这份数据的特点训练出了一个很厉害的模型，成为了一个挑瓜好手，只需告诉它这个西瓜的色泽，纹理和声音就能告诉你这个西瓜甜不甜。

通常将这种喂给机器学习算法来训练模型的数据称为训练集，用来让机器学习算法预测的数据称为测试集。

训练集中的所有行称为样本。由于我们的挑瓜好手需要的西瓜信息是色泽、纹理和声音，所以此训练集中每个样本的前 `3` 列称为特征。挑瓜好手给出的结果是甜或不甜，所以最后 `1` 列称为标签。

因此，这份数据是一个有 `4` 个样本， `3` 个特征的训练集，训练集的标签是“甜不甜”。

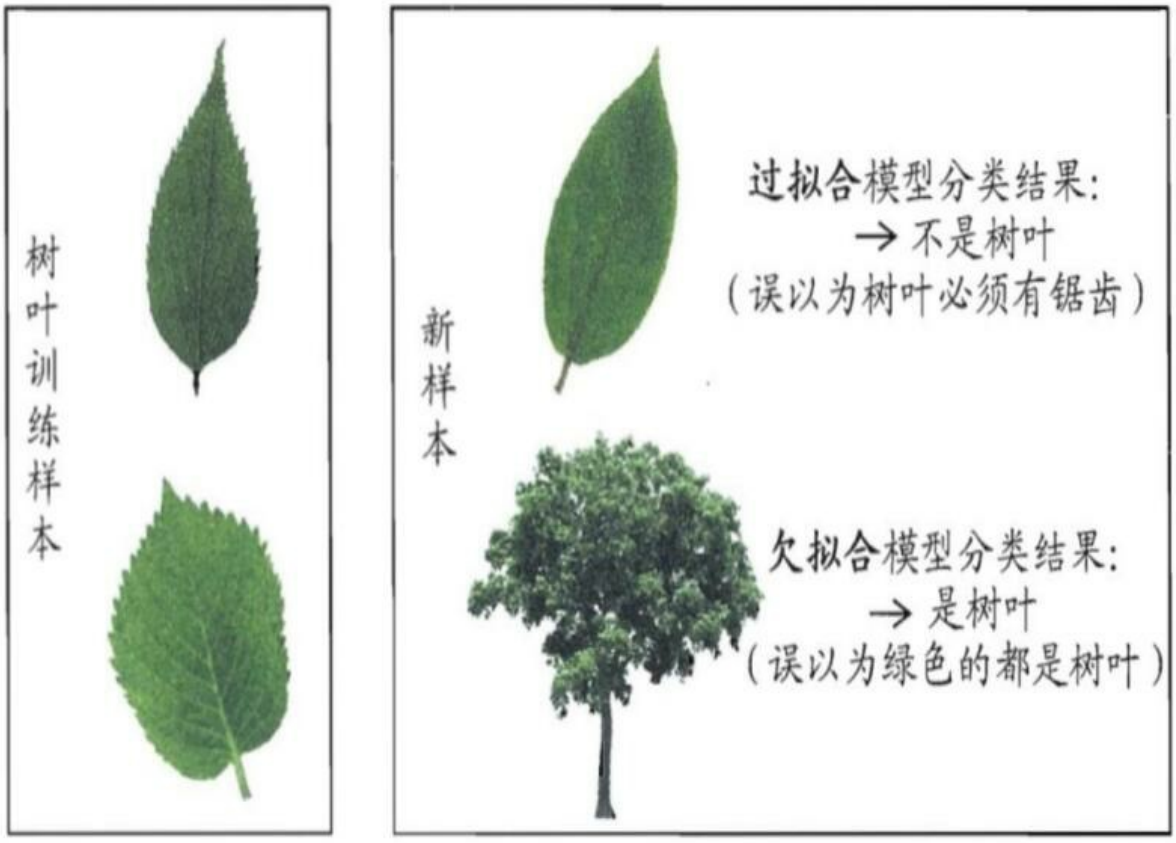
欠拟合与过拟合

最好的情况下，模型应该不管在训练集上还是测试集上，它的性能都不错。但是有的时候，模型在训练集上的性能比较差，那么这种情况我们称为欠拟合。那如果模型在训练集上的性能好到爆炸，但在测试集上的性能却不尽人意，那么这种情况我们称为过拟合。

其实欠拟合与过拟合的区别和生活中学生考试的例子很像。如果一个学生在平时的练习中题目的正确率都不高，那么说明这个学生可能基础不牢或者心思没花在学习上，所以这位学生可能欠缺基础知识或者智商可能不太高或者其他种种原因，像这种情况可以看成是欠拟合。那如果这位学生平时练习的正确率非常高，但是他不怎么灵光，喜欢死记硬背，只会做已经做过的题，一碰到没见过的新题就不知所措了。像这种情况可以看成是过拟合。

那么是什么原因导致了欠拟合和过拟合呢？

当模型过于简单，很可能会导致欠拟合。如果模型过于复杂，就很可能会导致过拟合。



验证集与交叉验证

在真实业务中，我们可能没有真正意义上的测试集，或者说不知道测试集中的数据长什么样子。那么怎样在没有测试集的情况下来验证模型好还是不好呢？这个时候就需要验证集了。

那么验证集从何而来，很明显，可以从训练集中抽取一小部分的数据作为验证集，用来验证模型的性能。

但如果仅仅是从训练集中抽取一小部分作为验证集的话，有可能会让对模型的性能有一种偏见或者误解。

比如现在要对手写数字进行识别，那么我就可能会训练一个分类模型。但可能模型对于数字 `1` 的识别准确率比较低，而验证集中没多少个数字为 `1` 的样本，然后用验证集测试完后得到的准确率为 `0.96`。然后您可能觉得哎呀，我的模型很厉害了，但其实并不然，因为这样的验证集让您的模型的性能有了误解。那有没有更加公正的验证算法性能的方法呢？有，那就是**k-**折交叉验证！

在**K-**折交叉验证中，把原始训练数据集分割成K个不重合的■数据集，然后做K次模型训练和验证。每■次，使■ 个■数据集验证模型，并使■其它K−1个■数据集来训练模型。在这K次训练和验证中，每次■来验证模型的■数据集都不同。最后，对这K次在验证集上的性能求平均。

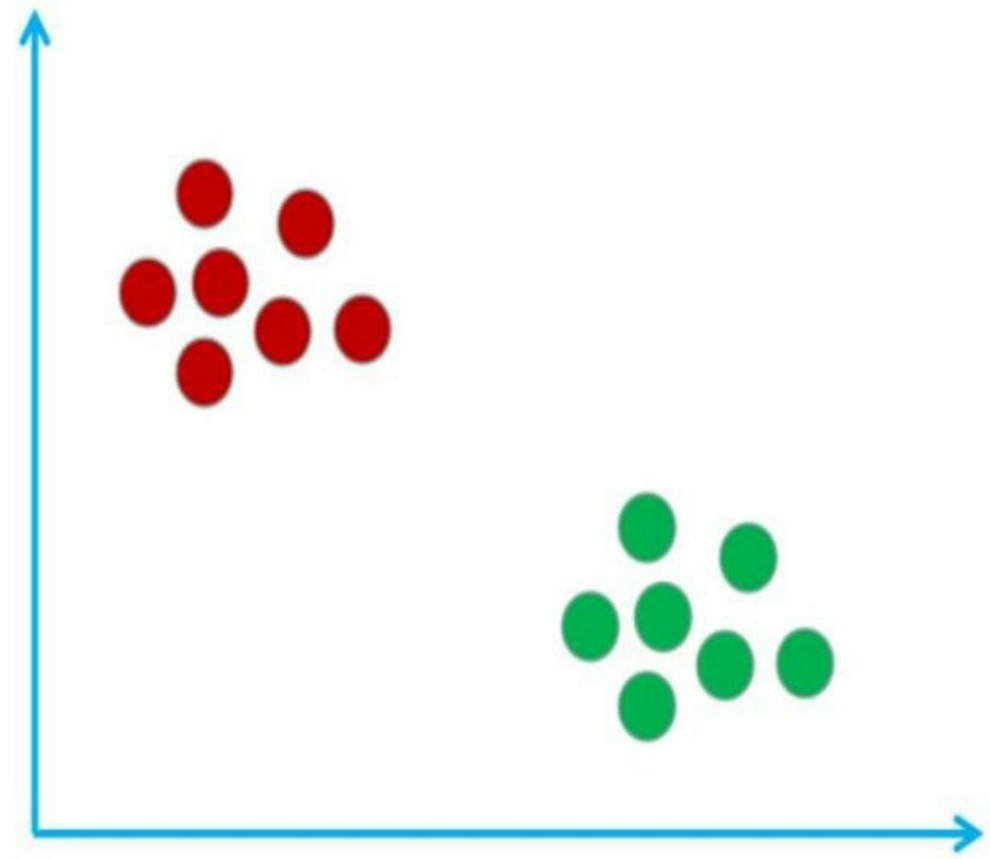
`K` 的值由我们自己来指定，如以下为 `5` 折交叉验证。

5.2.1 近朱者赤近墨者黑---kNN

kNN 算法其实是众多机器学习算法中最简单的一种，因为该算法的思想完全可以用 8 个字来概括：“近朱 者赤，近墨者黑”。

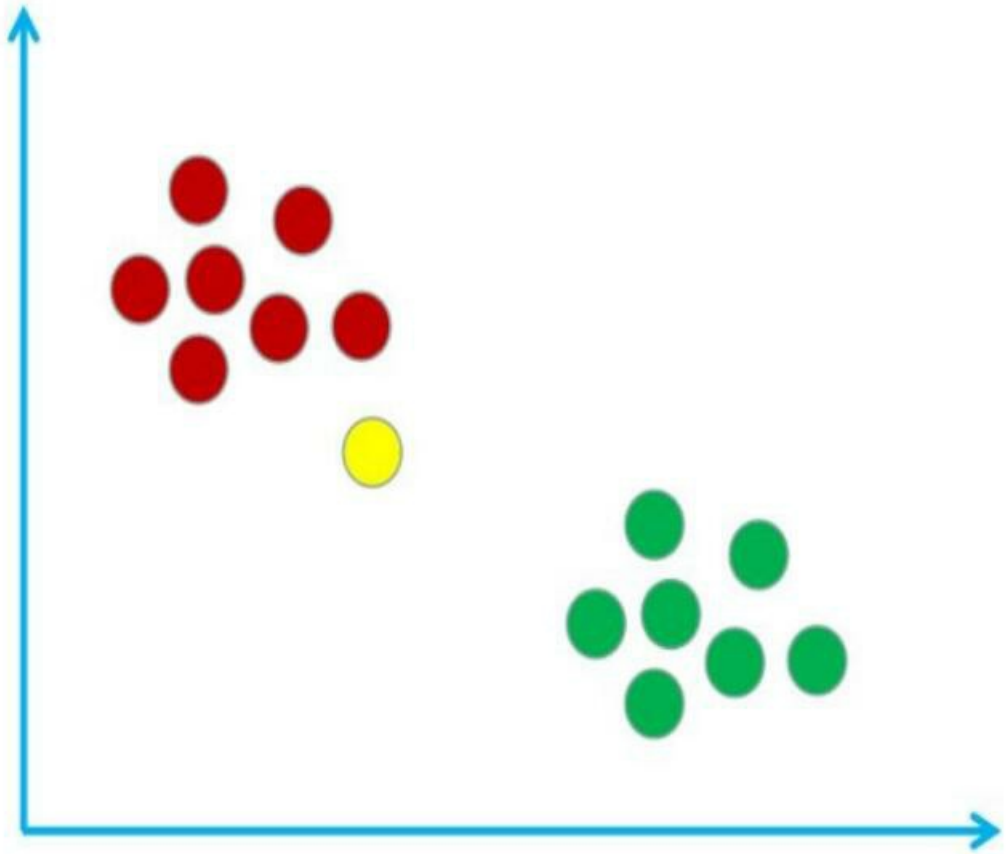
kNN算法解决分类问题

假设现在有这样的一个样本空间(由样本组成的一个空间)，该样本空间里有宅男和文艺青年这两个类别，其中红圈表示宅男，绿圈表示文艺青年。如下图所示：



其实构建出这样的样本空间的过程就是 kNN 算法的训练过程。可想而知 kNN 算法是没有训练过程的，所以 kNN 算法属于懒惰学习算法。

假设我在这个样本空间中用黄圈表示，如下图所示：



现在使用 kNN 算法来鉴别一下我是宅男还是文艺青年。首先需要计算我与样本空间中所有样本的距离。假设计算得到的距离表格如下：

样本编号	**1**	**2**	**...**	**13**	**14**
标签	宅男	宅男	...	文艺青年	文艺青年
距离	11.2	9.5	...	23.3	37.6

然后找出与我距离最小的 `k` 个样本(`k` 是一个超参数，需要自己设置，一般默认为 `5`)，假设与我离得最近的 `5` 个样本的标签和距离如下：

样本编号	**4**	**5**	**6**	**7**	**8**
标签	宅男	宅男	宅男	宅男	文艺青年
距离	11.2	9.5	7.7	5.8	15.2

最后只需要对这 `5` 个样本的标签进行统计，并将票数最多的标签作为预测结果即可。如上表中，宅男是 `4` 票，文艺青年是 `1` 票，所以我是宅男。

注意：有的时候可能会有票数一致的情况，比如 `k = 4` 时与我离得最近的样本如下：

样本编号	**4**	**9**	**11**	**13**
标签	宅男	宅男	文艺青年	文艺青年
距离	4.2	9.5	7.7	5.8

可以看出宅男和文艺青年的比分是 `2 : 2`，那么可以尝试将属于宅男的 `2` 个样本与我的总距离和属于文艺青年的 `2` 个样本与我的总距离进行比较。然后选择总距离最小的标签作为预测结果。在这个例子中预测结果为文艺青年(宅男的总距离为 `4.2 + 9.5`，文艺青年的总距离为 `7.7 + 5.8`)。

kNN算法解决回归问题

很明显，刚刚使用**kNN**算法解决了一个分类问题，那**kNN**算法能解决回归问题吗？当然可以！

在使用 `kNN` 算法解决回归问题时的思路和解决分类问题的思路基本一致，只不过预测标签值是多少的的时候是将距离最近的 `k` 个样本的标签值加起来再算个平均，而不是投票。例如离待预测样本最近的 `5` 个样本的标签如下：

样本编号	**4**	**9**	**11**	**13**	**15**
标签	1.2	1.5	0.8	1.33	1.19

所以待预测样本的标签为： `(1.2+1.5+0.8+1.33+1.19)/5=1.204`

sklearn中的kNN算法

想要使用 `sklearn` 中使用 `kNN` 算法进行分类，代码如下(其中 `train_feature` 、 `train_label` 和 `test_feature` 分别表示训练集数据、训练集标签和测试集数据)：

```
from sklearn.neighbors import KNeighborsClassifier

#生成K近邻分类器
clf=KNeighborsClassifier()
#训练分类器
clf.fit(train_feature, train_label)
#进行预测
predict_result=clf.predict(test_feature)
```

当我们的 `kNN` 算法需要不同的参数时，上面的代码就不能满足我的需要了。所需要做的改变是在 `clf=KNeighborsClassifier()` 这一行中。 `KNeighborsClassifier()` 的构造函数其实还是有其他参数的。

比较常用的参数有以下几个:

- `n_neighbors` ： 即 `kNN` 算法中的 `K` 值，为一整数，默认为 `5` 。
- `metric` ： 距离函数。参数可以为字符串（预设好的距离函数）或者是callable对象。默认值为闵可夫斯基距离。
- `p` ： 当 `metric` 为闵可夫斯基距离公式时可用，为一整数，默认值为 `2` ，也就是欧式距离。

如果想动手实现 kNN 算法，并掌握如何使用 sklearn 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/aw9bxy75/challenges

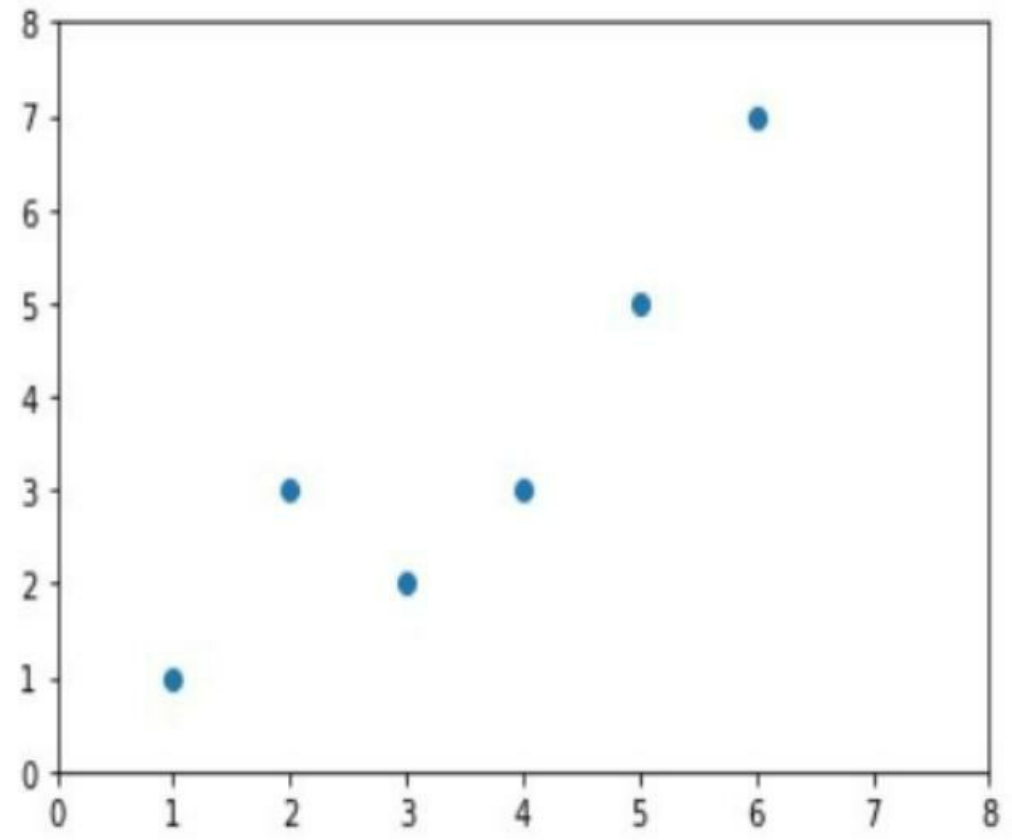
5.2.2 最简单的回归算法---线性回归

什么是线性回归

线性回归是什么意思？可以拆字释义。回归肯定不用我多说了，那什么是线性呢？我们可以回忆一下初中时学过的直线方程：`y=k*x+b`

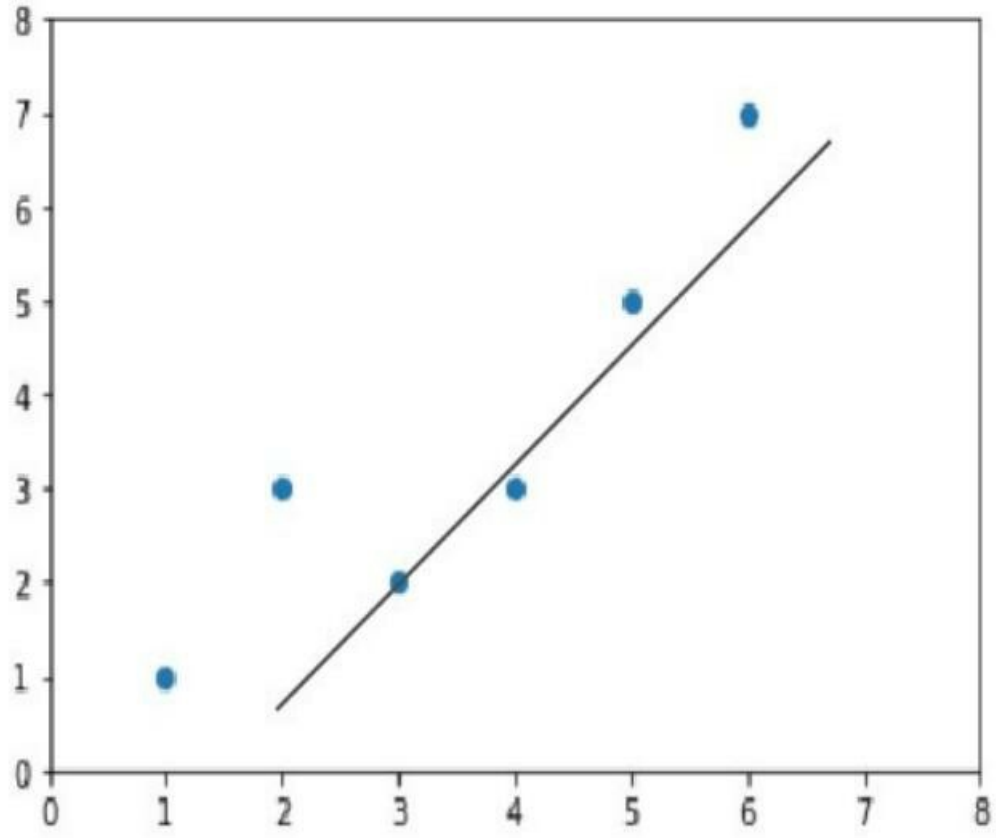
这个式子表达的是，当我知道 k （参数）和 b （参数）的情况下，我随便给一个 x 我都能通过这个方程 算出 y 来。而且呢，这个式子是线性的，为什么呢？因为从直觉上来说，你都知道，这个式子的函数图像 是条直线。

从理论上来说，这式子满足线性系统的性质(至于线性系统是什么，可以查阅相关资料，这里就不多做赘述了，不然没完没了)。你可能会觉得疑惑，这一节要说的是线性回归，我说个这么 low 直线方程干啥？其实，说白了，线性回归就是在 N 维空间中找一个形式像直线方程一样的函数来拟合数据而已。比如说，我现在有这么一张图，横坐标代表房子的面积，纵坐标代表房价。

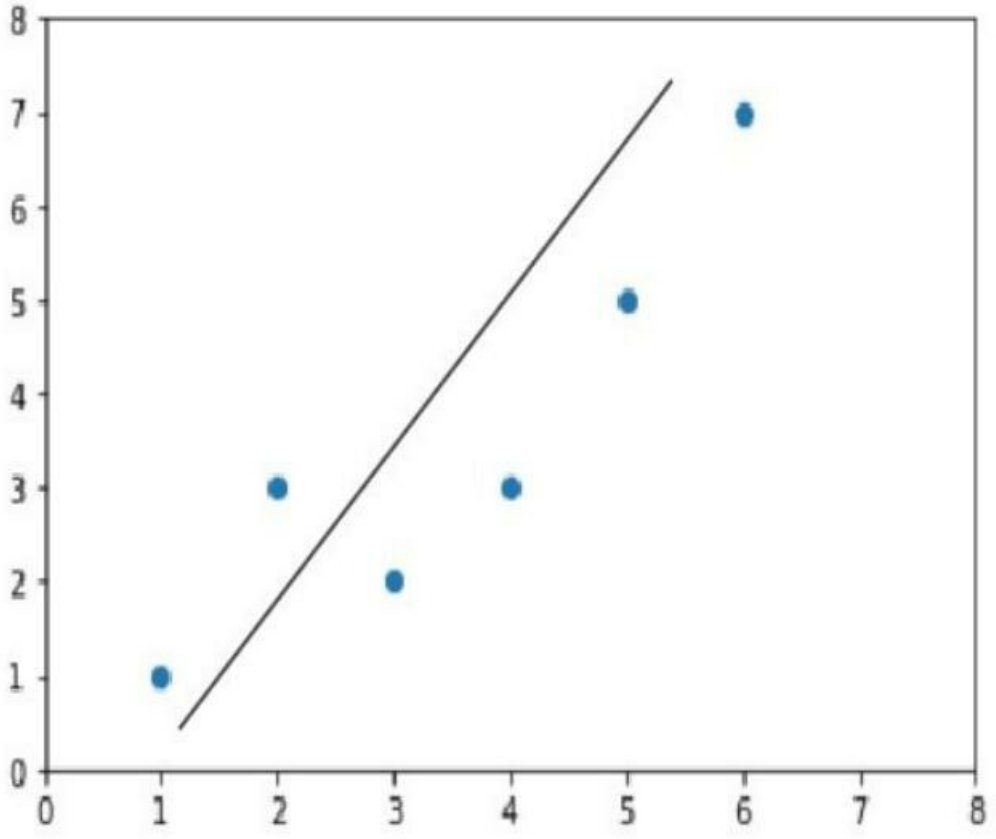


然后呢，线性回归就是要找一条直线，并且让这条直线尽可能地拟合图中的数据点。

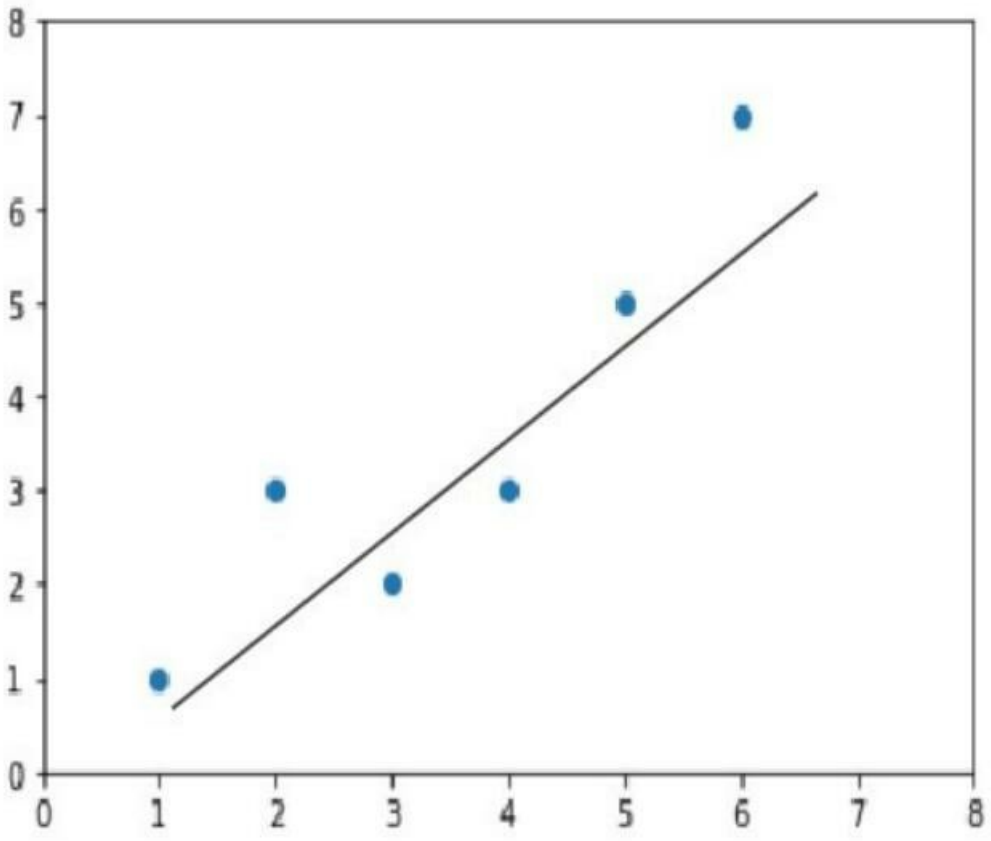
那如果让 1000 位朋友来找这条直线就可能找出 1000 种直线来，比如这样



这样



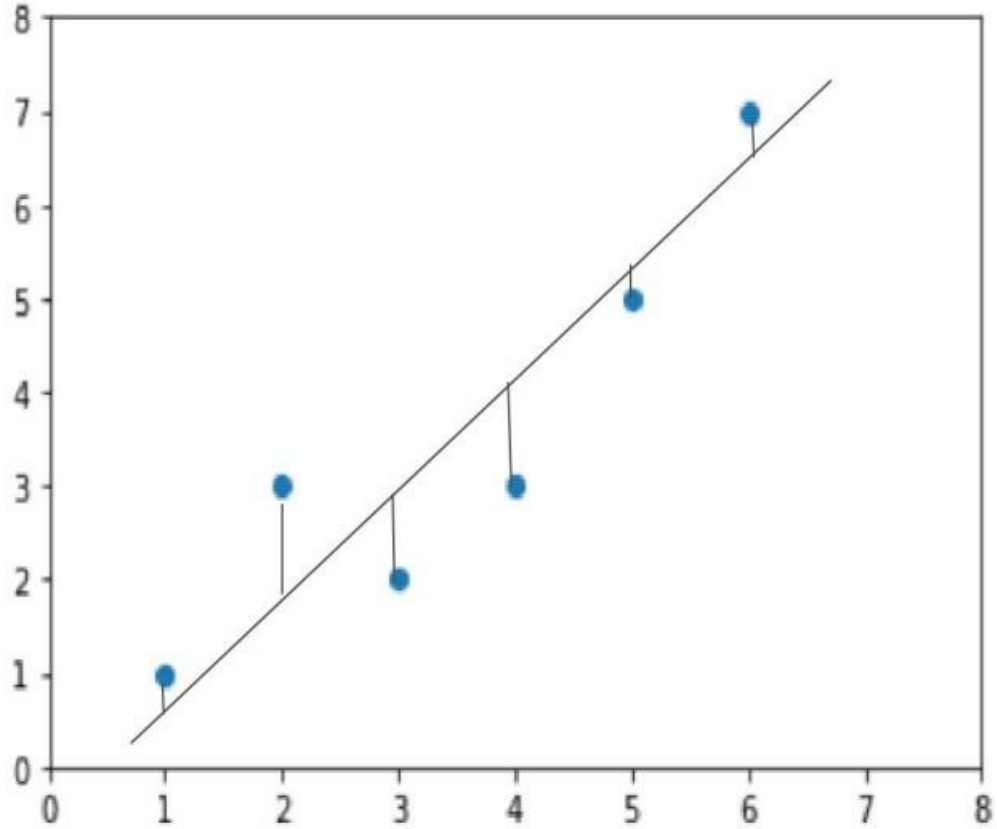
或者这样



喏，其实找直线的过程就是在做线性回归，只不过这个叫法更有高大上而已。

损失函数

那既然是找直线，那肯定是要有一个评判的标准，来评判哪条直线才是最好的。道理我们都懂，那咋评判呢？其实只要算一下实际房价和我找出的直线根据房子大小预测出来的房价之间的差距就行了。说白了就是算两点的距离。当把所有实际房价和预测出来的房价的差距（距离）算出来然后做个加和，就能量化出现在预测的房价和实际房价之间的误差。例如下图中我画了很多条小数线，每一条小数线就是实际房价和预测房价的差距（距离）。



然后把每条小竖线的长度加起来就等于现在通过这条直线预测出的房价与实际房价之间的差距。那每条小竖线的长度的加和怎么算？其实就是欧式距离加和，公式为：$\sum_{i=1}^{m}(y^{(i)}-\hat{y}^{(i)})^2$(其中$y(i)$表示的是实际房价，$\hat{y}(i)$表示的是预测房价)。

这个欧氏距离加和其实就是用来量化预测结果和真实结果的误差的一个函数。在机器学习中称它为损失函数（说白了就是计算误差的函数）。那有了这个函数，就相当于有了一个评判标准，当这个函数的值越小，就越说明找到的这条直线越能拟合房价数据。所以说啊，线性回归就是通过这个损失函数做为评判标准来找出一条直线。

如果假设$h_{(\theta)}(x)$表示当权重为θ，输入为x时计算出来的$\hat{y}(i)$，那么线性回归的损失函数$J(\theta)$就是：

$$J(\theta)=\frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^i)-y^i)^2$$

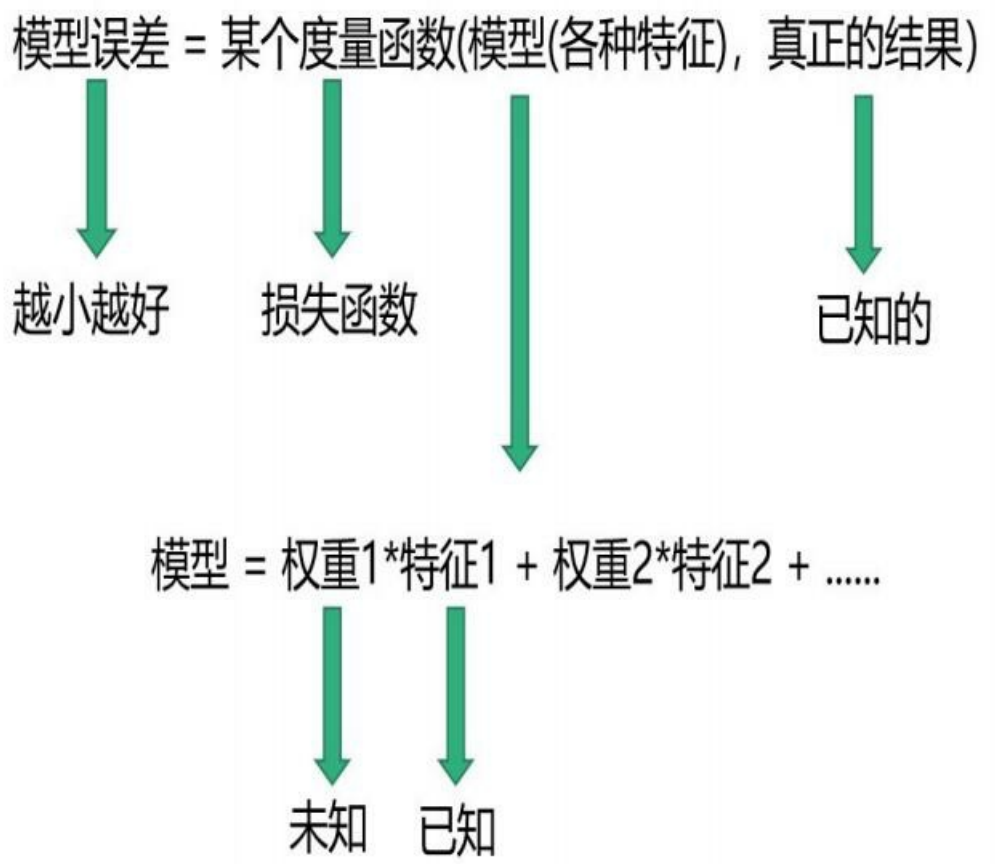
怎样计算出线性回归的解？

现在你应该已经弄明白了一个事实，那就是我只要找到一组参数（也就是线性方程每一项上的系数）能让我的损失函数的值最小，那我这一组参数就能最好的拟合我现在的训练数据。

那怎么来找到这一组参数呢？其实有两种套路，一种就是用大名鼎鼎的梯度下降，其大概思想就是根据每个参数对损失函数的偏导来更新参数。另一种是线性回归的正规方程解，这名字听起来高大上，其实本质就是根据一个固定的式子计算出参数。由于正规方程解在数据量比较大的时候时间复杂度比较高，所以在这一部分中，主要聊聊怎样使用梯度下降的方法来更新参数。

什么是梯度下降

其实梯度下降不是一个机器学习算法，而是一种基于搜索的最优化方法。因为很多算法都没有正规解的，所以需要通过一次一次的迭代来找到找到一组参数能让损失函数最小。损失函数的大概套路可以参看这个图：



所以说，梯度下降的作用是不断的寻找靠谱的权重是多少。

现在已经知道了梯度下降就是用来找权重的，那怎么找权重呢？瞎猜？不可能的....这辈子都不可能猜的。想想都知道，权重的取值范围可以看成是个实数空间，那 100 个特征就对应着 100 个权重， 10000 个特征就对应着 10000 个权重。如果靠瞎猜权重的话，应该这辈子都猜不中了。所以找权重的找个套路来找，这个套路就是梯度。梯度其实就是让函数值为 0 时其中各个变量的偏导所组成的向量，而且梯度方向是使得函数值增长最快的方向。

这个性质怎么理解呢？举个例子。假如我是个想要成为英雄联盟郊区王者的死肥宅，然后要成为郊区王者可能有这么几个因素，一个是英雄池的深浅，一个是大局观，还有一个是骚操作。他们对我成为王者来说都有一定的权重。如图所示，每一个因素的箭头都有方向（也就是因素对于我成为王者的偏导的方向）和长度（偏导的值的大小）。然后在这些因素的共同作用下，我最终会朝着一个方向来训练（好比物理中分力和合力的关系），这个时候我就能以最快的速度向郊区王者更进一步。

也就是说我如果一直朝着最终的那个方向努力的话，理论上来说我就能以最快的速度成为郊区王者。

现在你知道了梯度的方向是函数增长最快的方向，那我在梯度前面取个负号（反方向），那不就是函数下降最快的方向了么。所以，梯度下降它的本质就是更新权重的时候是沿着梯度的反方向更新。好比下面这个图，假如我是个瞎子，然后莫名其妙的来到了一个山谷里。现在我要做的事情就是走到山谷的谷底。因为我是瞎子，所以我只能一点一点的挪。要挪的话，那我肯定是那我的脚在我四周扫一遍，觉得哪里感觉起来更像是在下山那我就往哪里走。然后这样循环反复一发我最终就能走到山谷的谷底。

所以，梯度下降的伪代码如下：

$$repeat \quad unitl \quad convergenc \ e$$

$$for \quad every \quad \theta$$

$$\theta_j = \theta_j - \alpha \frac{\partial J(\theta_j)}{\partial \theta_j}$$

循环干的事情就相当于我下山的时候在迈步子，代码里的 α 高端点叫学习率，实际上就是代表我下山的时候步子迈多大。值越小就代表我步子迈得小，害怕一脚下去掉坑里。值越大就代表我胆子越大，步子迈得越大，但是有可能会越过山谷的谷底。

使用梯度下降求解线性回归的解

线性回归的损失函数 J 为：$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^i) - y^i)^2$，其中$\theta$为线性回归的解。使用梯度下降来求解，最关键的一步是算梯度(也就是算偏导)，通过计算可知第j个权重的偏导为：

$$\frac{\partial J(\theta_j)}{\theta_j} = (h_\theta(x) - y)x_j$$

所以很自然的可以想到，使用梯度下降求解线性回归的解的流程如下：

```
循环若干次
    计算当前参数theta对损失函数的梯度 gradient
    theta = theta - alpha * gradient
```

当θ更新好了之后，就相当于得到了一个线性回归模型。也就是说只要将数据放到模型中进行计算就能得到预测输出了。

sklearn中的线性回归

`LinearRegression` 的构造函数中有两个常用的参数可以设置：

- `fit_intercept` ：是否有截据，如果没有则直线过原点，默认为 `Ture` 。
- `normalize` ：是否将数据归一化,默认为 `False` 。

`LinearRegression` 类中的 `fit` 函数用于训练模型， `fit` 函数有两个向量输入：

- `X` ：大小为**[**样本数量,特征数量**]**的 `ndarray` ，存放训练样本
- `Y` ：值为整型，大小为**[**样本数量**]**的 `ndarray` ，存放训练样本的标签值

`LinearRegression` 类中的 `predict` 函数用于预测，返回预测值， `predict` 函数有一个向量输入：

- `X` ：大小为**[**样本数量,特征数量**]**的 `ndarray` ，存放预测样本

`LinearRegression` 的使用代码如下：

```
lr = LinearRegression()
lr.fit(X_train, Y_train)
predict = lr.predict(X_test)
```

如果想动手实现线性回归算法，并掌握如何使用 sklearn 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/4awq25iv/challenges

5.2.3 别被我的名字蒙蔽了---逻辑回归

逻辑回归是属于机器学习里面的监督学习，它是以回归的思想来解决分类问题的一种非常经典的二分类分类器。由于其训练后的参数有较强的可解释性，在诸多领域中，逻辑回归通常用作 `baseline` 模型，以方便后期更好的挖掘业务相关信息或提升模型性能。

逻辑回归大体思想

当一看到“回归”这两个字，可能会认为逻辑回归是一种解决回归问题的算法，然而逻辑回归是通过回归的思想来解决二分类问题的算法。

那么问题来了，回归的算法怎样解决分类问题呢？其实很简单，逻辑回归是将样本特征和样本所属类别的概率联系在一起，假设现在已经训练好了一个逻辑回归的模型为$f(x)$，模型的输出是样本x的标签是1的概率，则该模型可以表示成$\hat{p} = f(x)$。若得到了样本x属于标签1的概率后，很自然的就能想到当$\hat{p} > 0.5$时x属于标签1，否则属于标签 0 。所以就有：

$$\hat{y} = \begin{cases} 0 & \hat{p} < 0.5 \\ 1 & \hat{p} > 0.5 \end{cases}$$

(其中$\hat{y}$为样本 x 根据模型预测出的标签结果，标签 0 和标签 1 所代表的含义是根据业务决定的，比如在癌细胞识别中可以使 0 代表良性肿瘤，1 代表恶性肿瘤)。

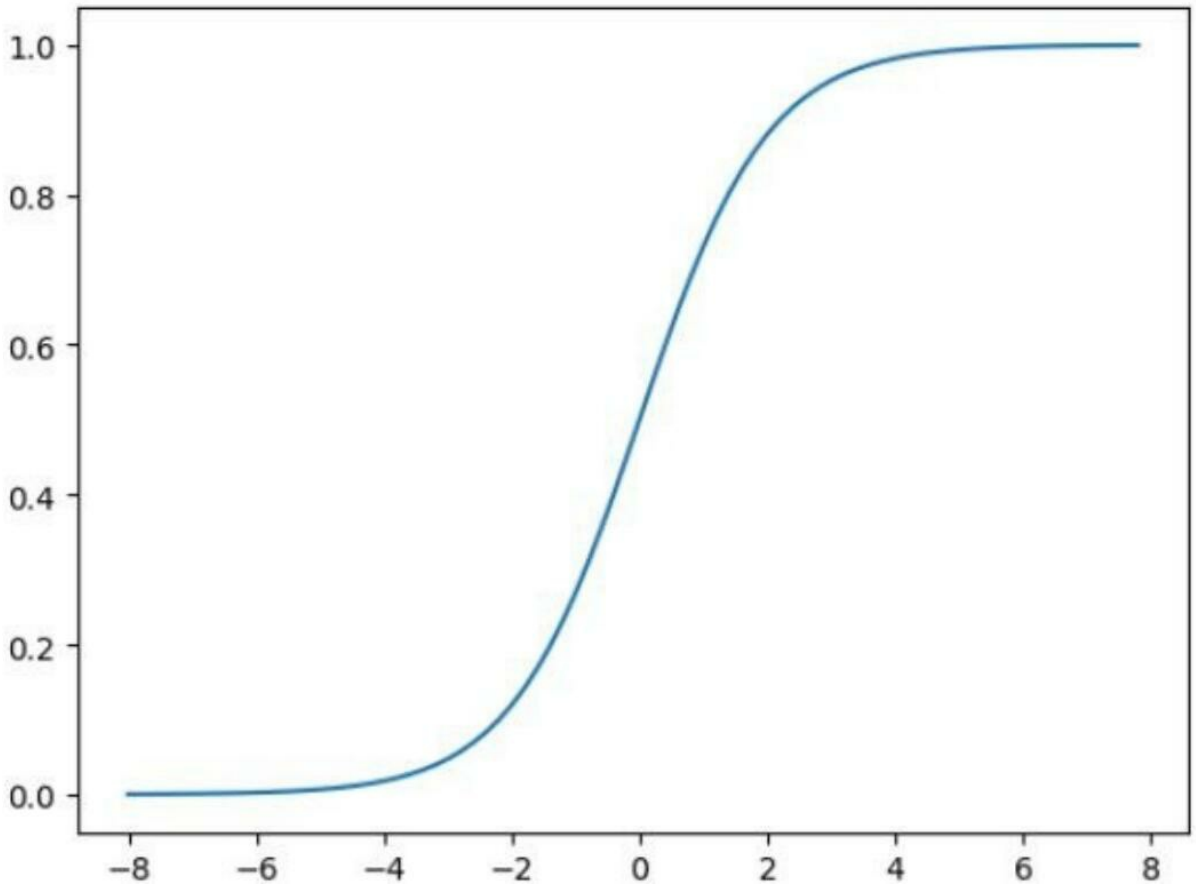
由于概率是 0 到 1 的实数，所以逻辑回归若只需要计算出样本所属标签的概率就是一种回归算法，若需要计算出样本所属标签，则就是一种二分类算法。

那么逻辑回归中样本所属标签的概率怎样计算呢？其实和线性回归有关系，学习了线性回归的话肯定知道线性回归就是训练出一组参数 W 和 b 来拟合样本数据，线性回归的输出为$\hat{y} = Wx + b$ 。不过$\hat{y}$的值域是$(-\infty, +\infty)$，如果能够将值域为$(-\infty, +\infty)$ 的实数转换成 $(0, 1)$ 的概率值的话问题就解决了。要解决这个问题很自然地就能想到将线性回归的输出作为输入，输入到另一个函数中，这个函数能够进行转换工作，假设函数为 σ ，转换后的概率为 $\hat{p}$ ，则逻辑回归在预测时可以看成 $\hat{p} = \sigma(Wx + b)$。 σ 其实就是接下来要介绍的$sigmoid$函数。

sigmoid函数

$sigmoid$函数的公式为：$\sigma(t) = 1/1 + e^{-t}$

函数图像如下图所示：



从$sigmoid$函数的图像可以看出当t趋近于$-\infty$时函数值趋近于0，当t趋近于$+\infty$时函数值趋近于1。可见$sigmoid$函数的值域是$(0,1)$，满足我们要将$(-\infty,+\infty)$的实数转换成$(0,1)$的概率值的需求。因此逻辑回归在预测时可以看成$\hat{p}=1/(1+e^{-Wx+b})$，如果$\hat{p}>0.5$时预测为一种类别，否则预测为另一种类别。

逻辑回归的损失函数

在预测样本属于哪个类别时取决于算出来的$\hat{p}$。从另外一个角度来说，假设现在有一个样本的真实类别为 1 ，模型预测样本为类别 1 的概率为 0.9 的话，就意味着这个模型认为当前样本的类别有 90% 的可能性为 1 ，有 10% 的可能性为 0 。所以从这个角度来看，逻辑回归的损失函数与 $\hat{p}$ 有关。

当然逻辑回归的损失函数不仅仅与 $\hat{p}$ 有关，它还与真实类别有关。假设现在有两种情况，情况 **A** ：现在有个样本的真实类别是 0 ，但是模型预测出来该样本是类别 1 的概率是 0.7（也就是说类别 0 的概率为 0.3 ）；情况 **B** ：现在有个样本的真实类别是 0 ，但是模型预测出来该样本是类别 1 的概率是 0.6（也就是说类别 0 的概率为 0.4 ）；请你思考 2 秒钟，**AB** 两种情况哪种情况的误差更大？很显然，情况 **A** 的误差更大！因为情况 **A** 中模型认为样本是类别 0 的可能性只有 30%，而 情况 **B** 有 40%。

假设现在又有两种情况，情况**A**： 现在有个样本的真实类别是 0 ，但是模型预测出来该样本是类别 1 的概率是 0.7（也就是说类别 0 的概率为 0.3）；情况**B**：现在有个样本的真实类别是 1，但是模型预测出来该样本是类别 1 的概率是 0.3（也就是说类别 0 的概率为 0.7 ）；请你再思考 2 秒钟，**AB**两种情况哪种情况的误差更大？很显然，一样大！

所以逻辑回归的损失函数如下，其中 $cost$ 表示损失函数的值，y 表示样本的真实类别：

$cost = -ylog(\hat{p}) - (1-y)log(1-\hat{p})$

知道了逻辑回归的损失函数之后，逻辑回归的训练流程就很明显了，就是寻找一组合适的 W 和 b，使得损失值最小。找到这组参数后模型就确定下来了。怎么找？很明显，用梯度下降，而且不难算出梯度为：

$(\hat{y} - y)x$。

所以逻辑回归梯度下降的代码如下：

```
#loss
def J(theta, X_b, y):
    y_hat = self._sigmoid(X_b.dot(theta))
    try:
        return -np.sum(y*np.log(y_hat)+(1-y)*np.log(1-y_hat)) / len(y)
    except:
        return float('inf')

# 算theta对loss的偏导
def dJ(theta, X_b, y):
    return X_b.T.dot(self._sigmoid(X_b.dot(theta)) - y) / len(y)

# 批量梯度下降
def gradient_descent(X_b, y, initial_theta, leraning_rate, n_iters=1e4, epsilon=1e-8):
    theta = initial_theta
    cur_iter = 0
    while cur_iter < n_iters:
        gradient = dJ(theta, X_b, y)
        last_theta = theta
        theta = theta - leraning_rate * gradient
        if (abs(J(theta, X_b, y) - J(last_theta, X_b, y)) < epsilon):
            break
        cur_iter += 1
    return theta
```

sklearn中的逻辑回归

`sklearn` 中的 `LogisticRegression` 默认实现了 `OVR` ,因此 `LogisticRegression` 可以实现多分类。 `LogisticRegression` 的构造函数中有三个常用的参数可以设置：

- `solver` ： `{'newton-cg' , 'lbfgs', 'liblinear', 'sag', 'saga'}` ， 分别为几种优化算法。默认为 `liblinear` 。
- `C` ： 正则化系数的倒数，默认为 `1.0` ，越小代表正则化越强。
- `max_iter` ： 最大训练轮数，默认为 `100` 。

和 `sklearn` 中其他分类器一样， `LogisticRegression` 类中的 `fit` 函数用于训练模型， `fit` 函数有两个向量输入：

- `X` ： 大小为 [样本数量,特征数量] 的 `ndarray` ，存放训练样本
- `Y` ： 值为整型，大小为 [样本数量] 的 `ndarray` ，存放训练样本的分类标签

`LogisticRegression` 类中的 `predict` 函数用于预测，返回预测标签， `predict` 函数有一个向量输入：

- `X` ： 大小为[样本数量,特征数量]的 `ndarray` ，存放预测样本

`LogisticRegression` 的使用代码如下：

```
logreg = LogisticRegression(solver='lbfgs',max_iter =10,C=10)
logreg.fit(X_train, Y_train)
result = logreg.predict(X_test)
```

如果想动手实现逻辑回归算法，并掌握如何使用 sklearn 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/tw9up75v/challenges

5.2.4 用概率说话---朴素贝叶斯分类器

朴素贝叶斯分类算法是基于贝叶斯理论和特征条件独立假设的分类算法。对于给定的训练集，首先基于特征条件独立假设学习数据的概率分布。然后基于此模型，对于给定的特征数据 `x`，利用贝叶斯定理计算出标签 `y`。朴素贝叶斯分类算法实现简单，预测的效率很高，是一种常用的分类算法。

条件概率

朴素贝叶斯分类算法是基于贝叶斯定理与特征条件独立假设的分类方法，因此想要了解朴素贝叶斯分类算法背后的算法原理，就不得不用到概率论的一些知识，首当其冲就是条件概率。

什么是条件概率

概率指的是某一事件 `A` 发生的可能性，表示为 `P(A)`。而条件概率指的是某一事件 `A` 已经发生了条件下，另一事件 `B` 发生的可能性，表示为 `P(B|A)`，举个例子：

今天有 `25%` 的可能性下雨，即 `P(下雨)=0.25`；今天 `75%` 的可能性是晴天，即 `P(晴天)=0.75`；如果下雨，我有 `75%` 的可能性穿外套，即 `P(穿外套|下雨)=0.75`；如果下雨，我有 `25%` 的可能性穿T恤，即 `P(穿T恤|下雨)=0.25`；

从上述例子可以看出，条件概率描述的是 `|` 右边的事件已经发生之后，左边的事件发生的可能性，而不是两个事件同时发生的可能性！

怎样计算条件概率

设 `A，B` 是两个事件，且 `P(A)>0`，称 `P(B|A)=P(AB)/P(A)` 为在事件 `A` 发生的条件下，事件 `B` 发生的条件概率。(其中 `P(AB)` 表示事件**A**和事件**B**同时发生的概率)

举个例子，现在有一个表格，表格中统计了甲乙两个厂生产的产品中合格品数量、次品数量的数据。数据如下：

	甲厂	乙厂	合计
合格品	475	644	1119
次品	25	56	81
合计	500	700	1200

现在想要算一下已知产品是甲厂生产的，那么产品是次品的概率是多少。这个时候其实就是在算条件概率，计算非常简单。

假设事件 `A` 为产品是甲厂生产的，事件 `B` 为产品是次品。则根据表中数据可知 `P(AB)=25/1200`， `P(A)=500/1200`。则 `P(B|A)=P(AB)/P(A)=25/500`。

乘法定理

将条件概率的公式两边同时乘以 `P(A)`，就变成了乘法定理，即 `P(AB)=P(B|A)*P(A)`。那么乘法定理怎么用呢？举个例子：

现在有一批产品共 `100` 件，次品有 `10` 件，从中不放回地抽取 `2` 次，每次取 `1` 件。现在想要算一下第一次为次品，第二次为正品的概率。

从问题来看，这个问题问的是第一次为次品，第二次为正品这两个事件同时发生的概率。所以可以用乘法定理来解决这个问题。

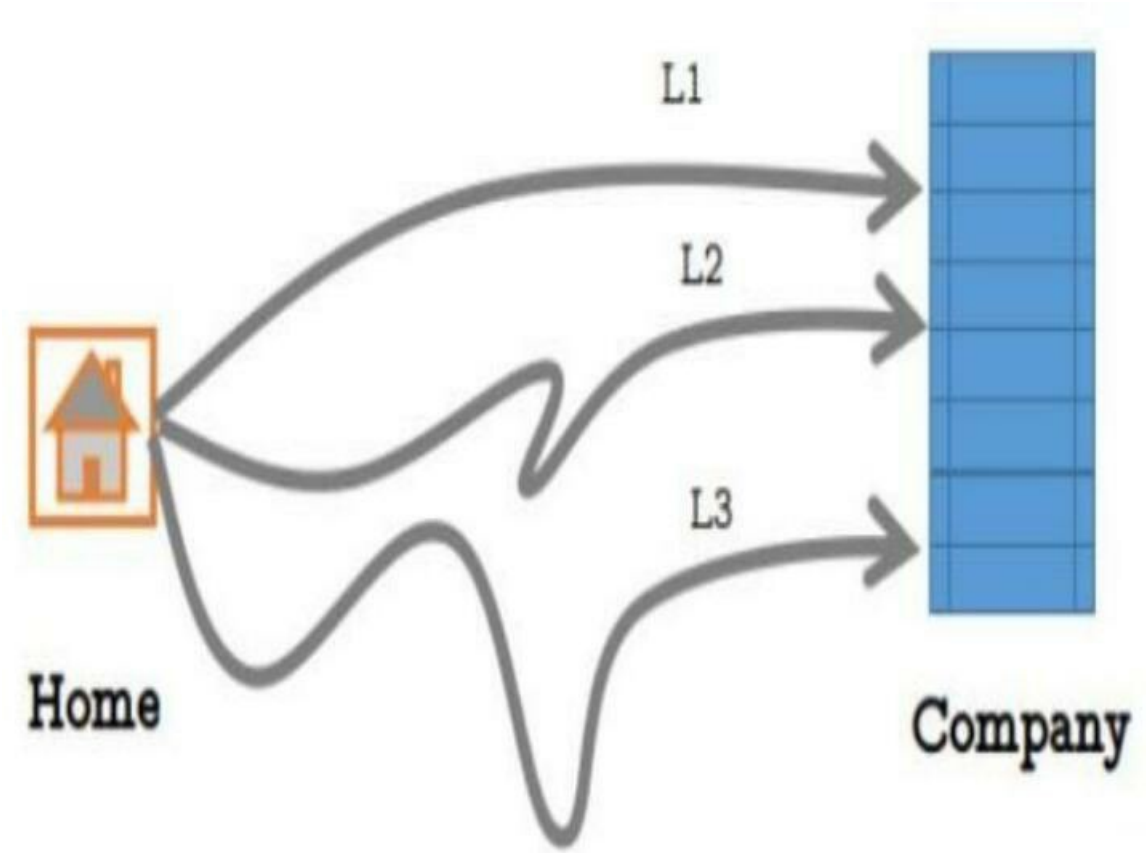
假设事件 `A` 为第一次为次品，事件 `B` 为第二次为正品。则 `P(AB)=P(A)*P(B|A)=(10/100)*(90/99)=0.091` 。

全概率公式

贝叶斯公式是朴素贝叶斯分类算法的核心数学理论，在了解贝叶斯公式之前，我们需要先了解全概率公式的相关知识。

引例

小明从家到公司上班总共有三条路可以直达，如下图：



但是每条路每天拥堵的可能性不太一样，由于路的远近不同，选择每条路的概率如下表所示：

L1	**L2**	**L3**
0.5	0.3	0.2

每天从上述三条路去公司时不堵车的概率如下表所示：

L1不堵车	**L2**不堵车	**L3**不堵车
0.2	0.4	0.7

如果不堵车就不会迟到，现在小明想要算一算去公司上班不会迟到的概率是多少，应该怎么办呢？

其实很简单，假设事件 `C` 为小明不迟到，事件 `A1` 为小明选 `L1` 这条路并且不堵车，事件 `A2` 为小明选 `L2` 这条路并且不堵车，事件 `A3` 为小明选 `L3` 这条路并且不堵车。那么很显然 `P(C)=P(A1)+P(A2)+P(A3)` 。

那么问题来了， `P(A1)` 、 `P(A2)` 和 `P(A3)` 怎么算呢？其实只要会算 `P(A1)` 其他的就都会算了。我们同样可以假设事件 `D1` 为小明选择 `L1` 路，事件 `E1` 为不堵车。那么 `P(A1)=P(D1)*P(E1)` 。但是在从表格中我们只知道 `P(D1)=0.5` ，怎么办呢？

回忆一下上面介绍的乘法定理，不难想到 `P(A1)=P(D1)*P(E1|D1)` 。从表格中可以看出 `P(E1|D1)=0.2` 。因此 `P(A1)=0.5*0.2=0.1` 。

然后依葫芦画瓢可以很快算出， `P(A2)=0.3*0.4=0.12` ， `P(A3)=0.2*0.7=0.14` 。所以 `P(C)=0.1+0.12+0.14=0.36` 。

什么是全概率公式

当为了达到某种目的，但是达到目的有很多种方式，如果想知道通过所有方式能够达到目的的概率是多少的话，就需要用到全概率公式（上面的例子就是这种情况！）。全概率公式的定义如下：

若事件$B_1,B_2,...,B_n$两两互不相容，并且其概率和为 `1` 。那么对于任意一个事件 `C` 都满足：

$$P(C)=P(B_1)P(C|B_1)+...+P(B_n)P(C|B_n)=\sum_{i=1}^{n}P(B_i)P(C|B_i)$$

引例中小明选择哪条路去公司的概率是**两两互不相容的**（只能选其中一条路去公司），**并且和为1**。所以小明不迟到的概率可以通过全概率公式来计算，而引例中的计算过程就是用的全概率公式。 ## 贝叶斯公式 当已知引发事件发生的各种原因的概率，想要算该事件发生的概率时，我们可以用**全概率公式**。但如果现在反过来，已知事件已经发生了，但想要计算引发该事件的各种原因的概率时，我们就需要用到**贝叶斯公式**了。 贝叶斯公式定义如下，其中A表示已经发生的事件，B_i为导致事件A发生的第i个原因：

$$P(B_i|A)=\frac{P(A|B_i)P(B_i)}{\sum_{i=1}^{n}P(A|B_i)P(B_i)}$$

贝叶斯公式看起来比较复杂，其实非常简单，分子部分是**乘法定理**，分母部分是**全概率公式**（分母等于$P(A)$）。 如果我们对贝叶斯公式进行一个简单的数学变换（两边同时乘以分母，再两边同时除以$P(B_i)$）。就能够得到如下公式：

$$P(A|B_i)=\frac{P(B_i|A)P(A)}{P(B_i)}$$

贝叶斯算法流程

在炎热的夏天你可能需要买一个大西瓜来解暑，但虽然你的挑西瓜的经验很老道，但还是会有挑错的时候。尽管如此，你可能还是更愿意相信自己经验。假设现在在你面前有一个纹路清晰，拍打西瓜后声音浑厚，按照你的经验来看这个西瓜是好瓜的概率有 `80` %，不是好瓜的概率有 `20` %。那么在这个时候你下意识会认为这个西瓜是好瓜，因为它是好瓜的概率大于不是好瓜的概率。

朴素贝叶斯分类算法的预测流程

朴素贝叶斯分类算法的预测思想和引例中挑西瓜的思想一样，会根据以往的经验计算出待预测数据分别为所有类别的概率，然后挑选其中概率最高的类别作为分类结果。

假如现在一个西瓜的数据如下表所示：

颜色	声音	纹理	是否为好瓜
绿	清脆	清晰	?

若想使用朴素贝叶斯分类算法的思想，根据这条数据中 `颜色` 、 `声音` 和 `纹理` 这三个特征来推断是不是好瓜，我们需要计算出这个西瓜是好瓜的概率和不是好瓜的概率。

假设事件 `A1` 为好瓜，事件 `B` 为绿，事件 `C` 为清脆，事件 `D` 为清晰，则这个西瓜是好瓜的概率为 `P(A1|BCD)` 。根据贝叶斯公式可知:

$$P(A_1|BCD) = \frac{P(A_1)P(B|A_1)P(C|A_1)P(D|A_1)}{P(BCD)}$$

同样，假设事件 `A2` 为好瓜，事件 `B` 为绿，事件 `C` 为清脆，事件 `D` 为清晰，则这个西瓜不是好瓜的概率为 `P(A2|BCD)` 。根据贝叶斯公式可知:

$$P(A_2|BCD) = \frac{P(A_2)P(B|A_2)P(C|A_2)P(D|A_2)}{P(BCD)}$$

朴素贝叶斯分类算法的思想是取概率最大的类别作为预测结果，所以如果满足下面的式子，则认为这个西瓜是好瓜，否则就不是好瓜：

$$\frac{P(A_1)P(B|A_1)P(C|A_1)P(D|A_1)}{P(BCD)} > \frac{P(A_2)P(B|A_2)P(C|A_2)P(D|A_2)}{P(BCD)}$$

从上面的式子可以看出， `P(BCD)` 是多少对于判断哪个类别的概率高没有影响，所以式子可以简化成如下形式：

$$P(A_1)P(B|A_1)P(C|A_1)P(D|A_1) > P(A_2)P(B|A_2)P(C|A_2)P(D|A_2)$$

所以在预测时，需要知道 `P(A1)` ， `P(A2)` ， `P(B|A_1)` ， `P(C|A_1)` ， `P(D|A_1)` 等于多少。而这些概率在训练阶段可以计算出来。

朴素贝叶斯分类算法的训练流程

训练的流程非常简单，主要是计算各种条件概率。假设现在有一组西瓜的数据，如下表所示：

编号	颜色	声音	纹理	是否为好瓜
1	绿	清脆	清晰	是
2	黄	浑厚	模糊	否
3	绿	浑厚	模糊	是
4	绿	清脆	清晰	是
5	黄	浑厚	模糊	是
6	绿	清脆	清晰	否

从表中数据可以看出：

`P(是好瓜)=4/6` `P(颜色绿|是好瓜)=3/4` `P(颜色黄|是好瓜)=1/4` `P(声音清脆|是好瓜)=1/2` `P(声音浑厚|是好瓜)=1/2` `P(纹理清晰|是好瓜)=1/2` `P(纹理模糊|是好瓜)=1/2` `P(不是好瓜)=2/6` `P(颜色绿|不是好瓜)=1/2` `P(颜色黄|是好瓜)=1/2` `P(声音清脆|不是好瓜)=1/2` `P(声音浑厚|不是好瓜)=1/2` `P(纹理清晰|不是好瓜)=1/2` `P(纹理模糊|不是好瓜)=1/2`

当得到以上概率后，训练阶段的任务就已经完成了。我们不妨再回过头来预测一下这个西瓜是不是好瓜。

颜色	声音	纹理	是否为好瓜
绿	清脆	清晰	?

假设事件 `A1` 为好瓜，事件 `B` 为绿，事件 `C` 为清脆，事件 `D` 为清晰。则有：

$$P(A_1)P(B|A_1)P(C|A_1)P(D|A_1) = \frac{4}{6} * \frac{3}{4} * \frac{1}{2} * \frac{1}{2} = \frac{1}{8}$$

假设事件 `A2` 为不是瓜，事件 `B` 为绿，事件 `C` 为清脆，事件 `D` 为清晰。则有：

$$P(A_2)P(B|A_2)P(C|A_2)P(D|A_2) = \frac{2}{6} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} = \frac{1}{24}$$

由于$\frac{1}{8} > \frac{1}{24}$，所以这个西瓜是好瓜。

sklearn中的朴素贝叶斯分类器

`MultinomialNB` 是 `sklearn` 中多项分布数据的朴素贝叶斯算法的实现。

在 `MultinomialNB` 实例化时 `alpha` 是一个常用的参数。

- `alpha`：平滑因子。当等于 `1` 时，做的是拉普拉斯平滑；当小于 `1` 时做的是 `Lidstone` 平滑；当等于 `0` 时，不做任何平滑处理。

`MultinomialNB` 类中的 `fit` 函数实现了朴素贝叶斯分类算法训练模型的功能， `predict` 函数实现了法模型预测的功能。

其中 `fit` 函数的参数如下：

- `X`：大小为 `[样本数量,特征数量]` 的 `ndarry`，存放训练样本
- `Y`：值为整型，大小为 `[样本数量]` 的 `ndarray`，存放训练样本的分类标签

而 `predict` 函数有一个向量输入：

- `X`：大小为 `[样本数量,特征数量]` 的 `ndarry`，存放预测样本

`MultinomialNB` 的使用代码如下：

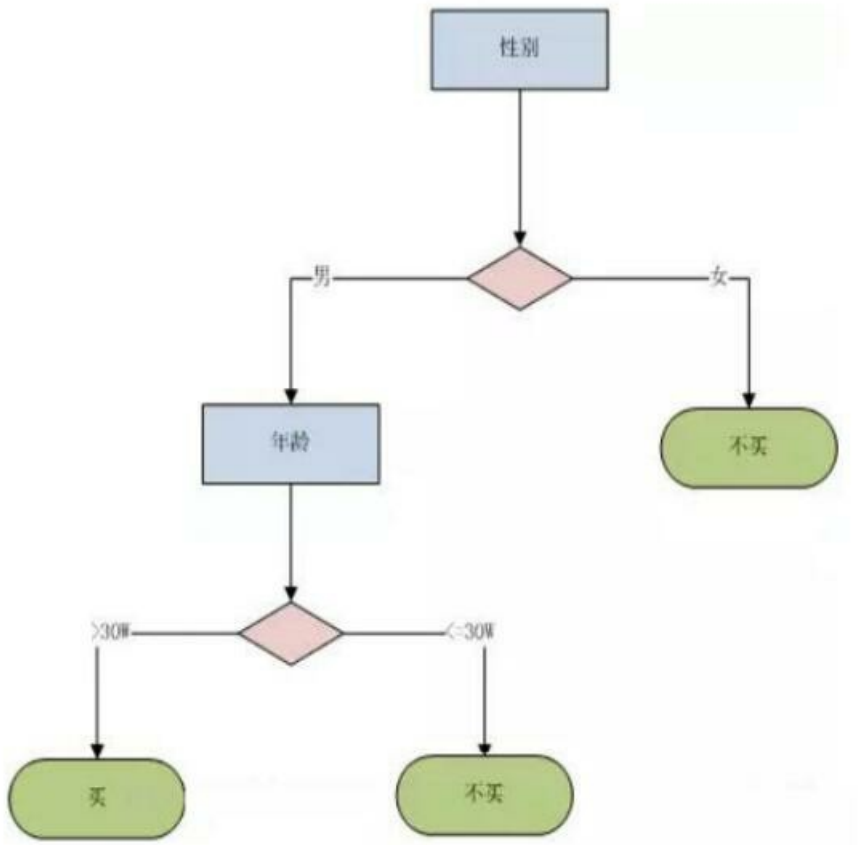
```
clf = tree.MultinomialNB()
clf.fit(X_train, Y_train)
result = clf.predict(X_test)
```

如果想动手实现朴素贝叶斯分类器算法，并掌握如何使用 sklearn 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/uyl5pk2q/challenges

5.2.5 最接近人类思维的算法---决策树

什么是决策树

决策树说白了就是一棵能够替我们做决策的树，或者说是我们人的脑回路的一种表现形式。比如我看到一个人，然后我会思考这个男人有没有买车。那我的脑回路可能是这样的：



其实这样一种脑回路的形式就是决策树。所以从图中能看出决策树是一个类似于人们决策过程的树结构，从根节点开始，每个分枝代表一个新的决策事件，会生成两个或多个分枝，每个叶子代表一个最终判定所属的类别。很明显，如果我现在已经构造好了一颗决策树的话，现在我得到一条数据`(男， 29)`，我就会认为这个人没有买过车。所以呢，关键问题就是怎样来构造决策树了。

构造决策树时会遵循一个指标，有的是按照信息增益来构建，这种叫**ID3**算法，有的是信息增益比来构建，这种叫**C4.5**算法，有的是按照基尼系数来构建的，这种叫**CART**算法。在这里主要介绍一下**ID3**算法。

ID3算法

整个**ID3**算法其实主要就是围绕着信息增益来的，所以要弄清楚**ID3**算法的流程，首先要弄清楚什么是信息增益，但要弄清楚信息增益之前有个概念必须要懂，就是熵。所以先看看什么是熵。

熵、条件熵、信息增益

在信息论和概率统计中呢，为了表示某个随机变量的不确定性，就借用了热力学的一个概念叫熵。如果假设 X 是一个有限个取值的离散型随机变量的话，很显然它的概率分布或者分布律就是这样的：

$$P(X=x_i)=p_i, i=1,2,...,n$$

有了概率分布后，则这个随机变量 X 的熵的计算公式就是（PS：这里的 log 是以 2 为底）：

$H(X)=-\sum_{i=1}^{n}p_ilogp_i$ 从这个公式也可以看出，如果我概率是 0 或者是 1 的时候，我的熵就是 0 。（因为这种情况下我随机变量的不确定性是最低的），那如果我的概率是 0.5 也就是五五开的时候，我的熵是最大也就是 1 。（就像扔硬币，你永远都猜不透你下次扔到的是正面还是反面，所以它的不确定性非常高）。所以呢，**熵越大，不确定性就越高**。 在实际情况下，要研究的随机变量基本上都是多随机变量的情况，所以假设有随便量(X,Y)，那么它的联合概率分布是这样的：

$$P(X=x_i,Y=y_j)=p_{ij}, i=1,2,...,n; j=1,2,...,m$$

那如果我想知道在我事件 X 发生的前提下，事件 Y 发生的熵是多少的话，这种熵叫它条件熵。条件熵 $H(Y|X)$ 表示随机变量 X 的条件下随机变量 Y 的不确定性。条件熵的计算公式是这样的：

$H(Y|X)=\sum_{i=1}^{n}p_iH(Y|X=x_i)$。

当然条件熵的一个性质也熵的性质一样，我概率越确定，条件熵就越小，概率越五五开，条件熵就越大。

现在已经知道了什么是熵，什么是条件熵。接下来就可以看看什么是信息增益了。所谓的信息增益就是表示我已知条件 X 后能得到信息 Y 的不确定性的减少程度。就好比，我在玩读心术。您心里想一件东西，我来猜。我一开始什么都没问你，我要猜的话，肯定是瞎猜。这个时候我的熵就非常高对不对。然后我接下来我会去试着问你是非题，当我问了是非题之后，我就能减小猜测你心中想到的东西的范围，这样其实就是减小了我的熵。那么我熵的减小程度就是我的信息增益。

所以信息增益如果套上机器学习的话就是，如果把特征 A 对训练集 D 的信息增益记为 $g(D,A)$ 的话，那么 $g(D,A)$ 的计算公式就是：$g(D,A)=H(D)-H(D|A)$。

如果看到这一堆公式可能会比较烦，那不如举个栗子来看看信息增益怎么算。假设我现在有这一个数据表，第一列是性别，第二列是活跃度， 第三列是客户是否流失的 $label$。

gender	act_info	is_lost
男	高	0
女	中	0
男	低	1
女	高	0
男	高	0
男	中	0
男	中	1
女	中	0
女	低	1
女	中	0
女	高	0
男	低	1
女	低	1
男	高	0
男	高	0

那如果我要算性别和活跃度这两个特征的信息增益的话，首先要先算总的熵和条件熵。(5/15 的意思是总共有 15 条样本里面 *label* 为 1 的样本有 5 条，3/8 的意思是性别为男的样本有 8 条，然后这 8 条里有 3 条是 *label* 为 1，其他的数值以此类推)

```
总熵= (-5/15)*log(5/15)-(10/15)*log(10/15)=0.9182
```

```
性别为男的熵= -(3/8)*log(3/8)-(5/8)*log(5/8)=0.9543
```

```
性别为女的熵= -(2/7)*log(2/7)-(5/7)*log(5/7)=0.8631
```

```
活跃度为低的熵= -(4/4)*log(4/4)-0=0
```

```
活跃度为中的熵= -(1/5)*log(1/5)-(4/5)*log(4/5)=0.7219
```

```
活跃度为高的熵= -0-(6/6)*log(6/6)=0
```

现在有了总的熵和条件熵之后就能算出性别和活跃度这两个特征的信息增益了。

```
性别的信息增益=总的熵-(8/15)*性别为男的熵-(7/15)*性别为女的熵=0.0064
```

```
活跃度的信息增益=总的熵-(6/15)*活跃度为高的熵-(5/15)*活跃度为中的熵-(4/15)*活跃度为低的熵=0.6776
```

那信息增益算出来之后有什么意义呢？回到读心术的问题，为了我能更加准确的猜出你心中所想，我肯定是问的问题越好就能猜得越准！换句话来说我肯定是要想出一个信息增益最大的问题来问你，对不对？其实**ID3**算法也是这么想的。**ID3**算法的思想是从训练集 D 中计算每个特征的信息增益，然后看哪个最大就选哪个作为当前节点。然后继续重复刚刚的步骤来构建决策树。

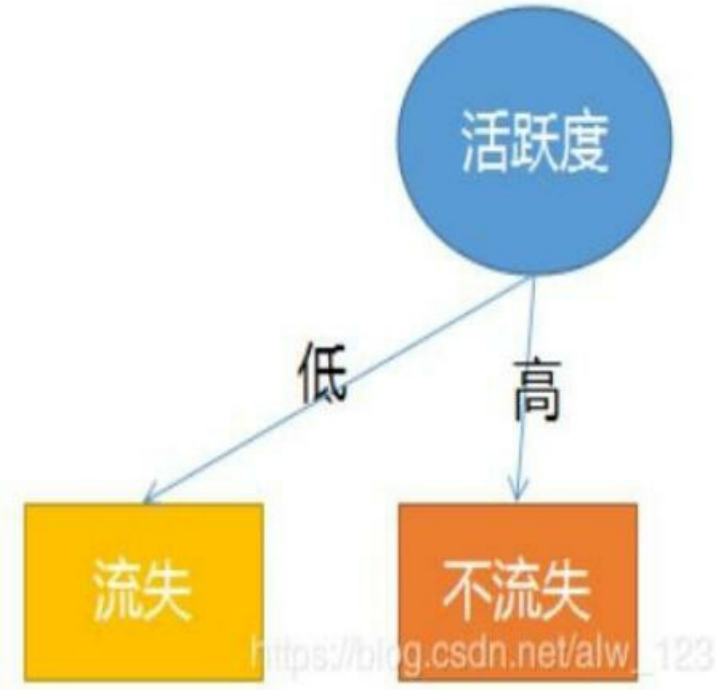
决策树构流程

ID3算法其实就是依据特征的信息增益来构建树的。具体套路就是从根节点开始，对节点计算所有可能的特征的信息增益，然后选择信息增益最大的特征作为节点的特征，由该特征的不同取值建立子节点，然后对子节点递归执行上面的套路直到信息增益很小或者没有特征可以继续选择为止。

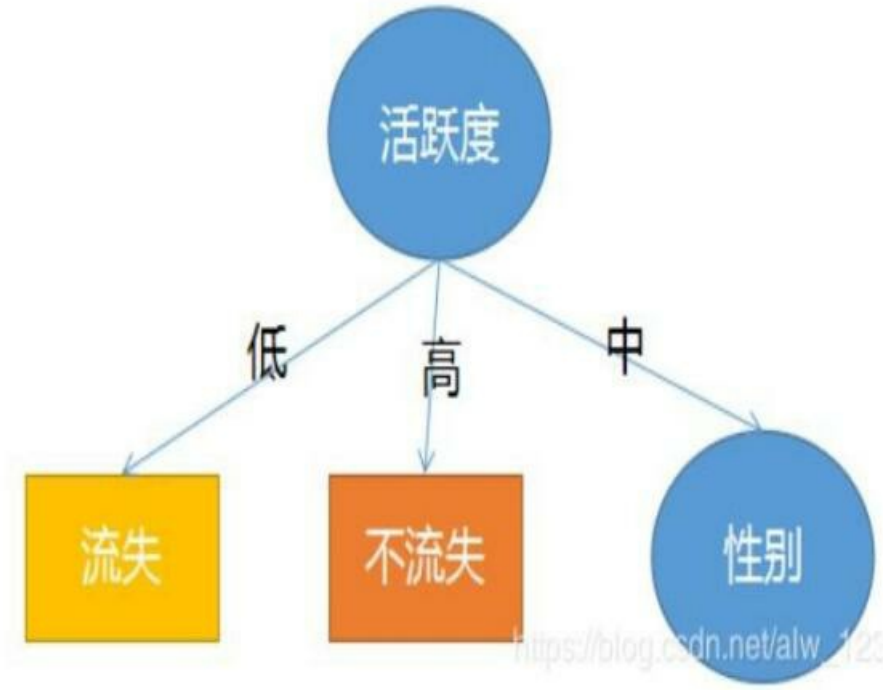
这样看上去可能会懵，不如用刚刚的数据来构建一颗决策树。

一开始已经算过信息增益最大的是活跃度，所以决策树的根节点是活跃度 。所以这个时候树是这样的：

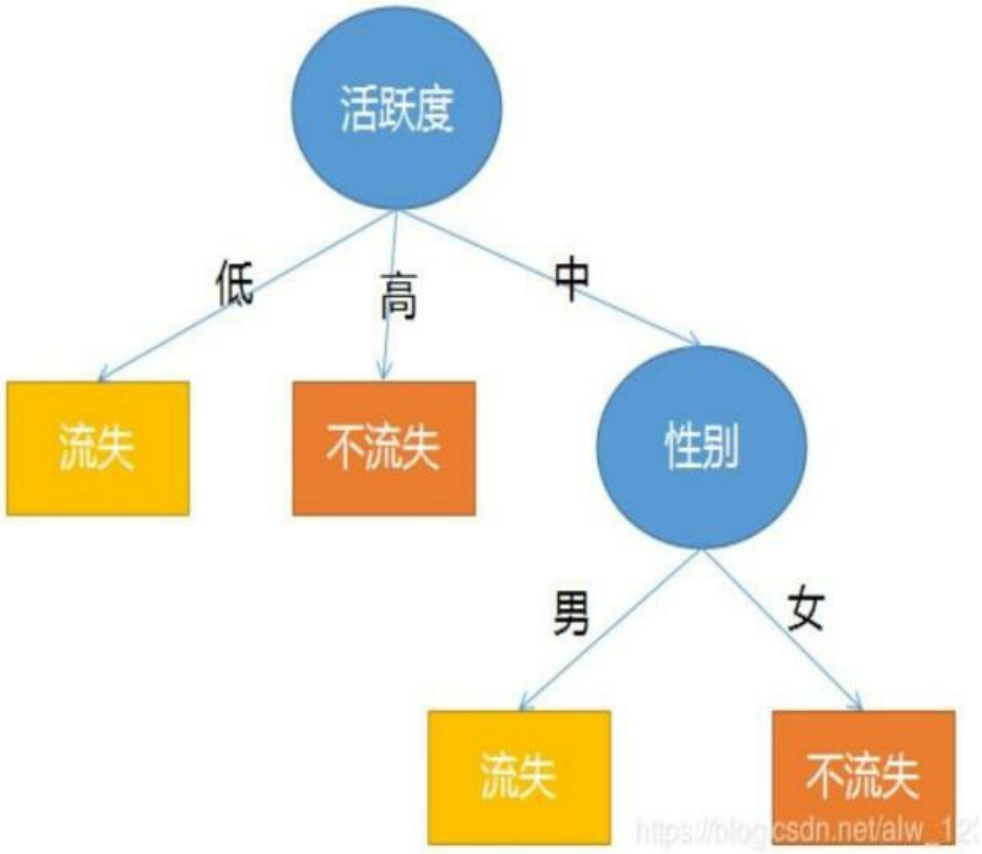
然后发现训练集中的数据表示当我活跃度低的时候一定会流失，活跃度高的时候一定不流失，所以可以先在根节点上接上两个叶子节点。



但是活跃度为中的时候就不一定流失了，所以这个时候就可以把活跃度为低和为高的数据屏蔽掉，屏蔽掉之后 5 条数据，接着把这 5 条数据当成训练集来继续算哪个特征的信息增益最高，很明显算完之后是性别这个特征，所以这时候树变成了这样：



这时候呢，数据集里没有其他特征可以选择了（总共就两个特征，活跃度已经是根节点了），所以就看我性别是男或女的时候那种情况最有可能出现了。此时性别为男的用户中有 1 个是流失，1 个是不流失，五五开。所以可以考虑随机选个结果当输出了。性别为女的用户中有全部都流失，所以性别为女时输出是流失。所以呢，树就成了这样：



好了，决策树构造好了。从图可以看出决策树有一个非常好的地方就是模型的解释性非常强！！很明显，如果现在来了一条数据 (男, 高) 的话，输出会是不流失。

sklearn中的决策树

`DecisionTreeClassifier` 的构造函数中有两个常用的参数可以设置：

- `criterion` :划分节点时用到的指标。有 `gini` （基尼系数）, `entropy` (信息增益)。若不设置，默认为 `gini`
- `max_depth` :决策树的最大深度，如果发现模型已经出现过拟合，可以尝试将该参数调小。若不设置，默认为 `None`

和 `sklearn` 中其他分类器一样， `DecisionTreeClassifier` 类中的 `fit` 函数用于训练模型， `fit` 函数有两个向量输入：

- `X` ：大小为 `[样本数量,特征数量]` 的 `ndarray` ，存放训练样本
- `Y` ：值为整型，大小为 `[样本数量]` 的 `ndarray` ，存放训练样本的分类标签

`DecisionTreeClassifier` 类中的 `predict` 函数用于预测，返回预测标签， `predict` 函数有一个向量输入：

- `X` ：大小为 `[样本数量,特征数量]` 的 `ndarray` ，存放预测样本

`DecisionTreeClassifier` 的使用代码如下：

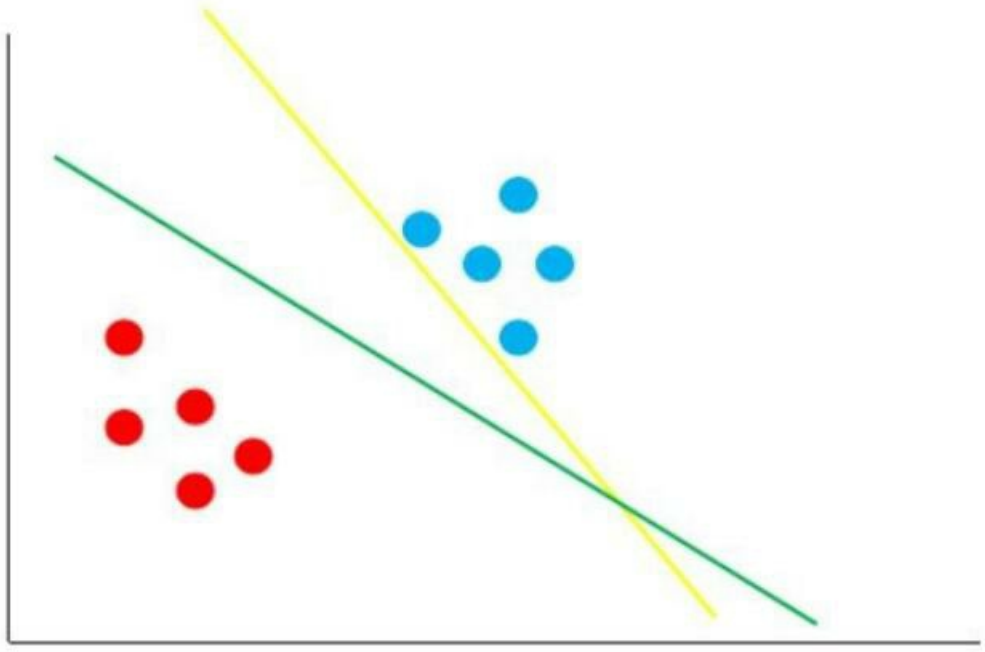
```
from sklearn.tree import DecisionTreeClassifier

clf = tree.DecisionTreeClassifier()
clf.fit(X_train, Y_train)
result = clf.predict(X_test)
```

如果想动手实现决策树算法，并掌握如何使用 sklearn 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/hl7wacq5/challenges

5.2.6 好还不够，我要最好---支持向量机

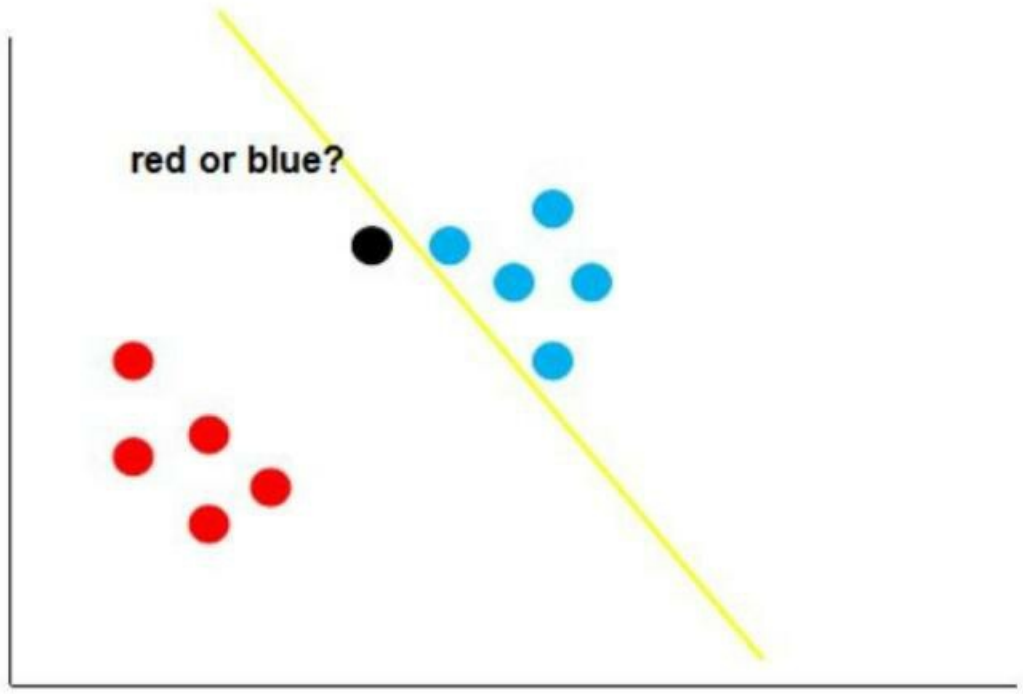
线性可分支持向量机

例如逻辑回归这种广义线性模型对数据进行分类时，其实本质就是找到一条 决策边界 ，然后将数据分成两个类别(边界的一边是一种类别，另一边是另一种类别)。如下图中的两条线是两种模型产生的 决策边界 ：



图中的绿线与黄线都能很好的将图中的红点与蓝点给区分开。但是，哪条线的泛化性更好呢？可能你不太了解泛化性，也就是说，这条直线，不仅需要在训练集 **(**已知的数据**)** 上能够很好的将红点跟蓝点区分开来，还要在测试集 **(**未知的数据**)** 上将红点和蓝点给区分开来。

假如经过训练，我们得到了黄色的这条决策边界用来区分数据，这个时候又来了一个数据，即黑色的点，那么你觉得黑色的点是属于红的这一类，还是蓝色的这一类呢？



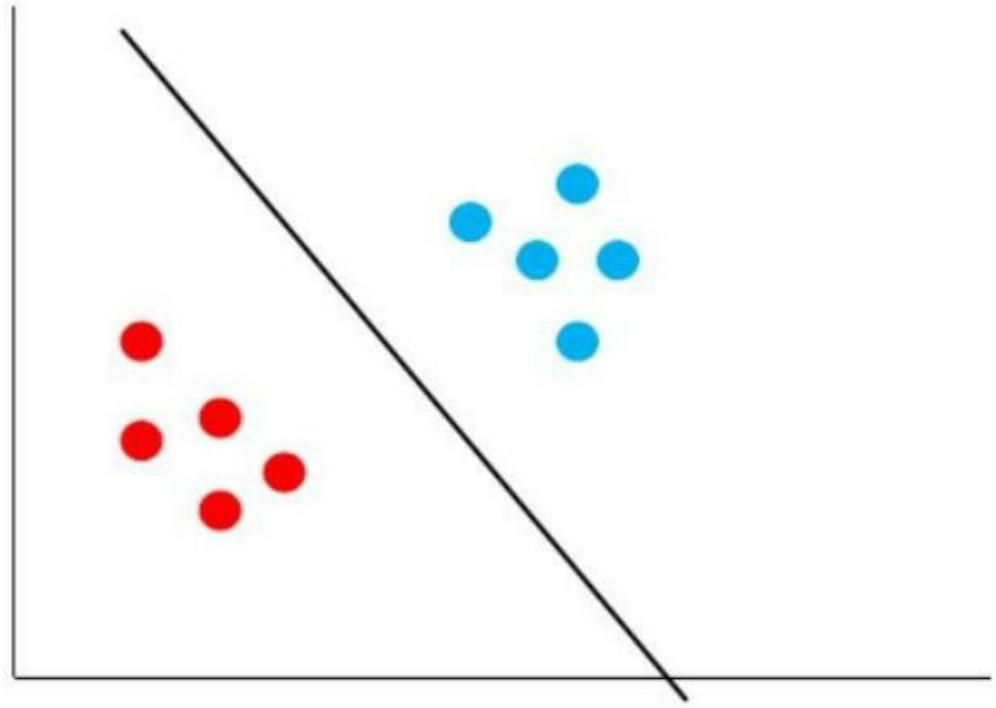
如上图，根据黄线的划分标准，黑色的点应该属于红色这一类。可是，肉眼很容易发现，黑点离蓝色的点更近，它应该是属于蓝色的点。这就说明，黄色的这条直线它的泛化性并不好，它对于未知的数据并不能很好的进行分类。那么，如何得到一条泛化性好的直线呢？这个就是支持向量机考虑的问题。

基本思想

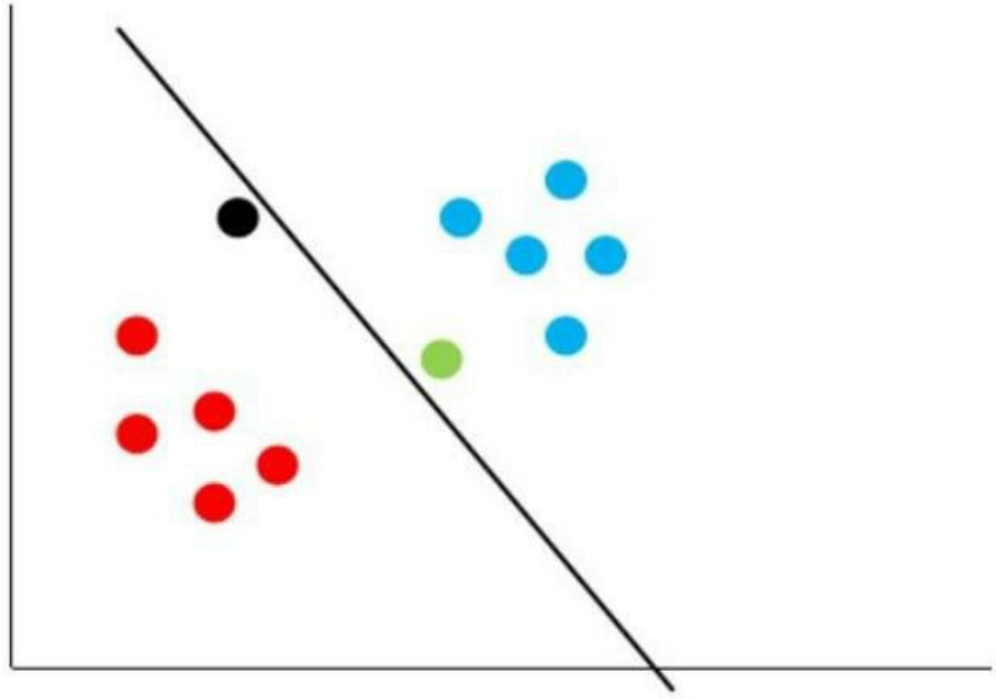
支持向量机的思想认为，一条决策边界它如果要有很好的泛化性，它需要满足一下以下两个条件：

- 能够很好的将样本划分
- 离最近的样本点最远

比如下图中的黑线：



它能够正确的将红点跟蓝点区分开来，而且，它还保证了对未知样本的容错率，因为它离最近的红点跟蓝点都很远，这个时候，再来一个数据，就不会出现之前黄色决策边界的错误了。



无论新的数据出现在哪个位置，黑色的决策边界都能够很好的给它进行分类，这个就是支持向量机的基本思想。

间隔与支持向量

在样本空间中，决策边界可以通过如下线性方程来描述：

$$w^Tx+b=0$$

其中$w=(w_1,w_2,..,w_d)$为法向量，决定了决策边界的方向。b为位移项，决定了决策边界与原点之间的距离。显然，决策边界可被法向量和位移确定，可以将其表示为(w,b)。样本空间中的任意一个点x，到决策边界(w,b)的距离可写为：

$$r=\frac{|w^Tx+b|}{||w||}$$

假设决策边界(w,b)能够将训练样本正确分类，即对于任何一个样本点(x_i,y_i)，若它为正类，即$y_i=+1$时，$w^Tx+b\geq+1$。若它为负类，即$y_i=-1$时，$w^Tx+b\leq-1$。 如图中，距离最近的几个点使两个不等式的等号成立，它们就被称为支持向量，即图中两条黄色的线。两个异类支持向量到超平面的距离之和为：

$$r=\frac{2}{||w||}$$

它被称为间隔，即蓝线的长度。欲找到具有“最大间隔”的决策边界，即黑色的线，也就是要找到能够同时满足如下式子的w与b: 显然，为了最大化间隔，仅需最大化$||w||^{-1}$，这等价于最小化$||w||^2$，于是，条件可以重写为：

$$min\frac{1}{2}||w||^2$$

$$s.t.y_i(w^Tx+b)\geq1$$

这就是线性可分支持向量机的基本型。

对偶问题

现在已经知道了支持向量机的基本型，问题本身是一个凸二次规划问题，可以用现成的优化计算包求解，不过可以用更高效的方法。支持向量机的模型如下：

$$min\frac{1}{2}||W||^2$$

$$s.t.y_i(w^Tx+b)\geq1$$

对两式子使用拉格朗日乘子法可得到其对偶问题。具体为，对式子的每条约束添加拉格朗日乘子$\alpha\geq0$，则该问题的拉格朗日函数可写为：

$$L(w,b,\alpha)=\frac{1}{2}||w||^2+\sum_{i=1}^m\alpha_i(1-y_i(w^Tx_i+b)).....(1)$$

其中$\alpha=(\alpha_1,\alpha_2,..,\alpha_m)$。令$L(w,b,\alpha)$对$w$和$b$的偏导为零可得：

$$w=\sum_{i=1}^m\alpha_iy_ix_i.....(2)$$

$$0=\sum_{i=1}^m\alpha_iy_i.....(3)$$

将式子2带入式子1，则可将w和b消去，再考虑式子3的约束，就可得到原问题的对偶问题：

$$max(\sum_{i=1}^m\alpha_i-\frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m\alpha_i\alpha_jy_iy_jx_i^Tx_j)$$

$$s.t. \sum_{i=1}^{m} \alpha_i y_i = 0$$

$$\alpha_i \geq 0$$

解出α后，求出w与b即可得到模型：

$$f(x) = w^T x + b = \sum_{i=1}^{m} \alpha_i y_i x_i^T x + b.....(4)$$

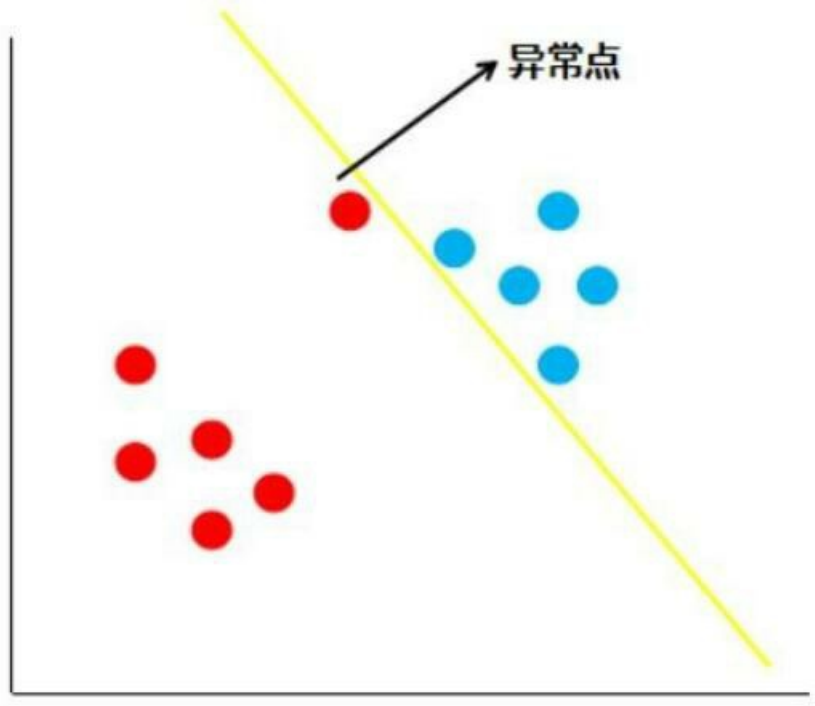
从对偶问题解出的α_i是拉格朗日乘子，它恰对应着训练样本(x_i, y_i)，由于原问题有不等式的约束，因此上述过程需满足KKT(Karush-Kuhn-Tucker)条件，即要求：

$$\begin{cases} \alpha_i \geq 0 \\ y_i f(x_i) - 1 \geq 0 \\ \alpha_i (y_i f(x_i) - 1) = 0 \end{cases}$$

于是，对任意训练样本(x_i, y_i)，总有$\alpha_i = 0$或$y_i f(x_i) = 1$。若$\alpha_i = 0$，则该样本将不会出现在式子4中，也就不会对$f(x)$，有任何影响。若$\alpha_i > 0$，则必有$y_i f(x_i) = 1$，所对应的样本点位于最大间隔边界上，是一个支持向量。这显示出支持向量机的一个重要性质: 训练完后，大部分的训练样本都不需要保留，最终模型仅与支持向量有关。

线性支持向量机

假如现在有一份数据分布如下图：



按照线性可分支持向量机的思想，黄色的线就是最佳的决策边界。很明显，这条线的泛化性不是很好，造成这样结果的原因就是数据中存在着异常点，那么如何解决这个问题呢，支持向量机引入了软间隔最大化的方法来解决。

所谓的软间隔，是相对于硬间隔说的，刚刚在间隔与支持向量中提到的间隔就是硬间隔。

接着我们再看如何可以软间隔最大化呢？SVM 对训练集里面的每个样本(x_i, y_i)引入了一个松弛变量$\xi_i \geq 0$，使函数间隔加上松弛变量大于等于1，也就是说：

$$y_i(w^T x + b) \geq 1 - \xi$$

对比硬间隔最大化，可以看到我们对样本到超平面的函数距离的要求放松了，之前是一定要大于等于1，现在只需要加上一个大于等于0的松弛变量能大于等于1就可以了。也就是允许支持向量机在一些样本上出错，

如下图： 当然，松弛变量不能白加，这是有成本的，每一个松弛变量ξ_i, 对应了一个代价ξ_i，这个就得到了我们的软间隔最大化的支持向量机，模型如下：

$$min\frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i$$

$$s.t. y_i(w^T x_i + b) \geq 1 - \xi_i$$

$$\xi_i \geq 0$$

这里，$C > 0$为惩罚参数，可以理解为一般回归和分类问题正则化时候的参数。C越大，对误分类的惩罚越大，C越小，对误分类的惩罚越小。也就是说，希望权值的二范数尽量小，误分类的点尽可能的少。C是协调两者关系的正则化惩罚系数。在实际应用中，需要调参来选择。 同样的，使用拉格朗日函数将软间隔最大化的约束问题转化为无约束问题，利用相同的方法得到数学模型：

$$max(\sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j x_i^T x_j)$$

$$s.t. \sum_{i=1}^{m}\alpha_i y_i = 0$$

$$0 \leq \alpha_i \leq C$$

这就是软间隔最大化时的线性支持向量机的优化目标形式，和之前的硬间隔最大化的线性可分支持向量机相比，我们仅仅是多了一个约束条件:

$$0 \leq \alpha_i \leq C$$

sklearn中的支持向量机

`SVR` 是 `sklearn` 中提供的支持向量机回归器， `SVR` 的构造函数中有四个常用的参数可以设置：

```
kernel：采用核函数类型，默认为`rbf`。可选参数有：`linear`（线性核函数）,`poly`（多项式核函数）,`rbf`：（径像核函数/高斯核）。
C：错误项的惩罚参数C，默认为1.0。
gamma:核函数系数，只对`rbf`,`poly`有效。如果gamma为auto，代表其值为样本特征数的倒数，即1/n_features。
epsilon:误差精度。
```

`SVR` 类中的 `fit` 函数用于训练模型， `fit` 函数有两个向量输入：

- `X`：大小为**[**样本数量,特征数量**]**的 `ndarray` ，存放训练样本
- `Y`：值为整型，大小为**[**样本数量**]**的 `ndarray` ，存放训练样本的标签值

`SVR` 类中的 `predict` 函数用于预测，返回预测值， `predict` 函数有一个向量输入：

- `X`：大小为**[**样本数量,特征数量**]**的 `ndarray`，存放预测样本

`SVR` 的使用代码如下：

```
from sklearn.svm import SVR
svr = SVR()
svr.fit(X_train, Y_train)
predict = lr.predict(X_test)
```

`SVC` 是 `sklearn` 中提供的支持向量机分类器，用法和 `SVR` 一致， `SVC` 的使用代码如下:

```
from sklearn.svm import SVC
svr = SVC()
svr.fit(X_train, Y_train)
predict = lr.predict(X_test)
```

如果想动手实现支持向量机算法，并掌握如何使用 sklearn 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/b6yi97f2/challenges

5.2.7 群众的力量是伟大的---随机森林

既然有决策树，那有没有用多棵决策树组成森林的算法呢？有！那就是随机森林。随机森林是一种叫**Bagging**的算法框架的变体。所以想要理解随机森林首先要理解**Bagging**。

Bagging

什么是**Bagging**

Bagging 是 Bootstrap Aggregating 的英文缩写，刚接触的您不要误认为 **Bagging** 是一种算法，**Bagging** 是集成学习中的学习框架， **Bagging** 是并行式集成学习方法。大名鼎鼎的随机森林算法就是在**Bagging**的基础上修改的算法。

Bagging 方法的核心思想就是三个臭皮匠顶个诸葛亮。如果使用 **Bagging** 解决分类问题，就是将多个分类器的结果整合起来进行投票，选取票数最高的结果作为最终结果。如果使用 **Bagging** 解决回归问题，就将多个回归器的结果加起来然后求平均，将平均值作为最终结果。

那么 **Bagging** 方法为什么如此有效呢，举个例子。狼人杀我相信您应该玩过，在天黑之前，村民们都要根据当天所发生的事和别人的发现来投票决定谁可能是狼人。

如果我们将每个村民看成是一个分类器，那么每个村民的任务就是二分类，假设 $h_i(x)$ 表示第 i 个村民认为 x 是不是狼人(-1 代表不是狼人， 1 代表是狼人)，$f(x)$ 表示 x 真正的身份(是不是狼人)，ϵ 表示为村民判断错误的错误率。则有 $P(h_i(x) \neq f(x)) = \epsilon$。

根据狼人杀的规则，村民们需要投票决定天黑前谁是狼人，也就是说如果有超过半数的村民投票时猜对了，那么这一轮就猜对了。那么假设现在有 T 个村民，$H(x)$ 表示投票后最终的结果，则有

$H(x) = sign(\sum_{i=1}^{T} h_i(x))$。

现在假设每个村民都是有主见的人，对于谁是狼人都有自己的想法，那么他们的错误率也是相互独立的。那么根据**Hoeffding**不等式可知，$H(x)$ 的错误率为：

$P(H(x) \neq f(x)) = \sum_{k=0}^{T/2} C_T^k (1-\epsilon)^k \epsilon^{T-k} \leq exp(-\frac{1}{2}T(1-2\epsilon)^2)$

根据上式可知，如果 5 个村民，每个村民的错误率为 0.33 ，那么投票的错误率为 0.749 ；如果 20 个村民，每个村民的错误率为 0.33 ，那么投票的错误率为 0.315 ；如果 50 个村民，每个村民的错误率为 0.33 ，那么投票的错误率为 0.056 ；如果 100 个村民，每个村民的错误率为 0.33 ，那么投票的错误率为 0.003 。从结果可以看出，村民的数量越大，那么投票后犯错的错误率就越小。这也是**Bagging**性能强的原因之一。

Bagging方法如何训练

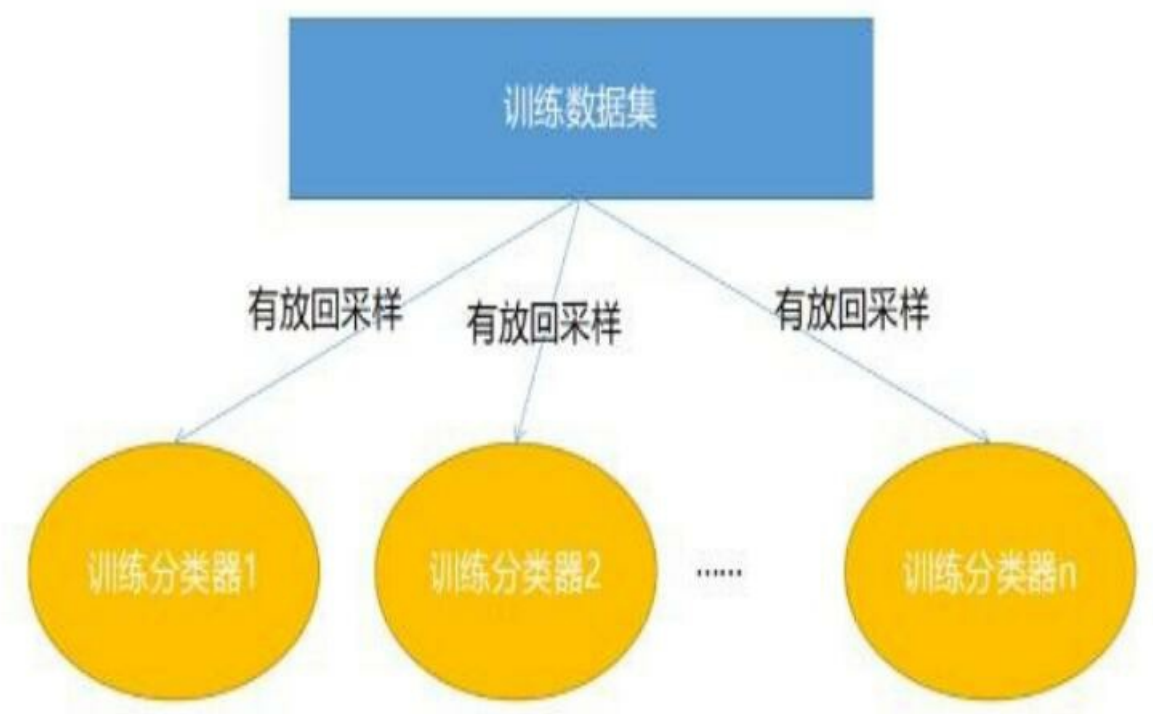
Bagging 在训练时的特点就是随机有放回采样和并行。

随机有放回采样： 假设训练数据集有 m 条样本数据，每次从这 m 条数据中随机取一条数据放入采样集，然后将其返回，让下一次采样有机会仍然能被采样。然后重复 m 次，就能得到拥有 m 条数据的采样集，该采样集作为 **Bagging** 的众多分类器中的一个作为训练数据集。假设有 T 个分类器（随便什么分类器），那

么就重复 T 此随机有放回采样，构建出 T 个采样集分别作为 T 个分类器的训练数据集。

并行： 假设有 10 个分类器，在**Boosting**中，1 号分类器训练完成之后才能开始2 号分类器的训练，而在**Bagging**中，分类器可以同时进行训练，当所有分类器训练完成之后，整个**Bagging**的训练过程就结束了。

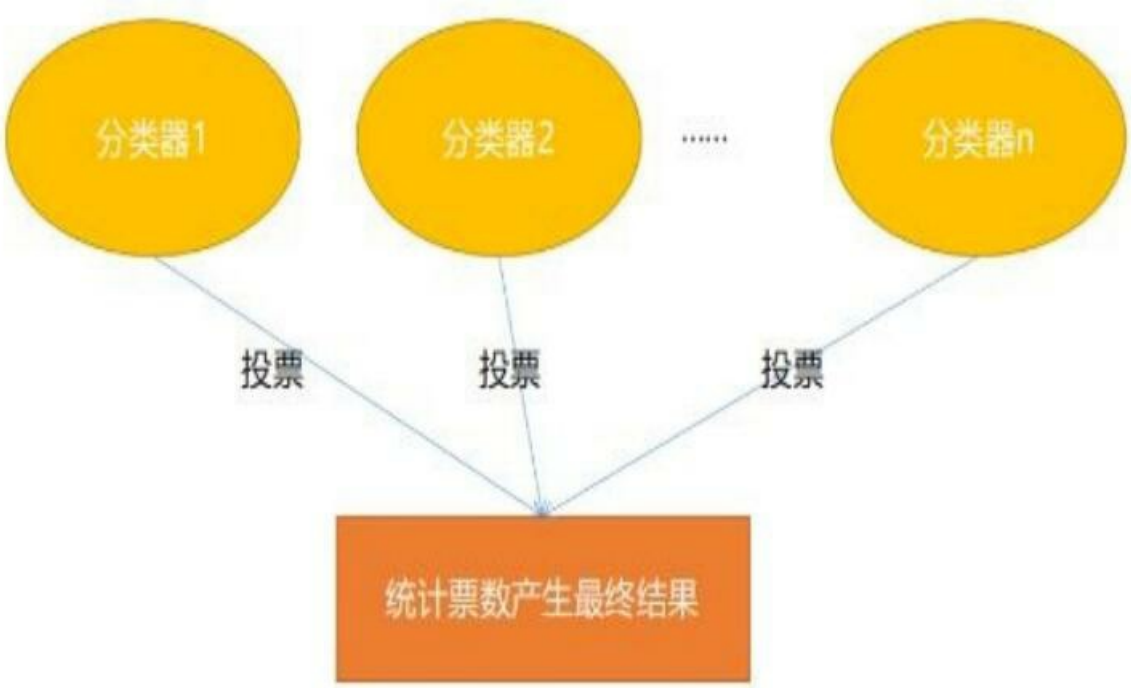
Bagging训练过程如下图所示：



Bagging方法如何预测

Bagging在预测时非常简单，就是投票！比如现在有 5 个分类器，有 3 个分类器认为当前样本属于 A 类，1 个分类器认为属于 B 类，1 个分类器认为属于 C 类，那么**Bagging**的结果会是 A 类（因为 A 类的票数最高）。

Bagging预测过程如下图所示:



随机森林

随机森林是**Bagging**的一种扩展变体，随机森林的训练过程相对与**Bagging**的训练过程的改变有：

- 基学习器：**Bagging**的基学习器可以是任意学习器，而随机森林则是以决策树作为基学习器。
- 随机属性选择：假设原始训练数据集有 10 个特征，从这 10 个特征中随机选取 k 个特征构成训练数据子集，然后将这个子集作为训练集扔给决策树去训练。其中 k 的取值一般为 $log2$ (特征数量)。

这样的改动通常会使得随机森林具有更加强的泛化性，因为每一棵决策树的训练数据集是随机的，而且训练数据集中的特征也是随机抽取的。如果每一棵决策树模型的差异比较大，那么就很容易能够解决决策树容易过拟合的问题。

sklearn中的随机森林

`RandomForestClassifier` 是 `skleran` 中随机森林分类器的实现，的构造函数中有两个常用的参数可以设置：

- `n_estimators` ： 森林中决策树的数量
- `criterion` ： 构建决策树时，划分节点时用到的指标。有 `gini` （基尼系数），`entropy` (信息增益)。若不设置，默认为 `gini`
- `max_depth` ： 决策树的最大深度，如果发现模型已经出现过拟合，可以尝试将该参数调小。若不设置，默认为 `None`
- `max_features` ： 随机选取特征时选取特征的数量，一般传入 `auto` 或者 `log2` ，默认为 `auto` ， `auto` 表示 `max_features=sqrt(训练集中特征的数量)` ； `log2` 表示 `max_features=log2(训练集中特征的数量)`

`RandomForestClassifier` 类中的 `fit` 函数实现了随机森林分类器训练模型的功能， `predict` 函数实现了模型预测的功能。

其中 `fit` 函数的参数如下：

- `X`：大小为 `[样本数量,特征数量]` 的 `ndarry`，存放训练样本
- `Y`：值为整型，大小为 `[样本数量]` 的 `ndarray`，存放训练样本的分类标签

而 `predict` 函数有一个向量输入：

- `X`：大小为 `[样本数量,特征数量]` 的 `ndarry`，存放预测样本

`RandomForestClassifier` 的使用代码如下：

```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(n_estimators=50)
clf.fit(X_train, Y_train)
result = clf.predict(X_test)
```

如果想动手实现随机森林算法，并掌握如何使用 sklearn 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/ya8h7utx/challenges

5.2.8 知错能改善莫大焉---Adaboost

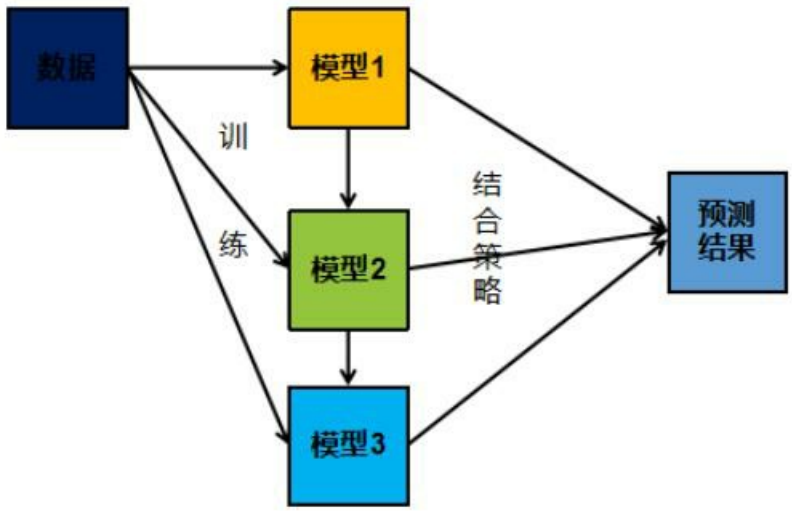
Boosting

提升方法基于这样一种思想：对于一个复杂任务来说，将多个专家的判断进行适当的综合所得出的判断，要比其中任何一个专家单独的判断好。

历史上，`Kearns` 和 `Valiant` 首先提出了强可学习和弱可学习的概念。指出：在 `PAC` 学习的框架中，一个概念，如果存在一个多项式的学习算法能够学习它，并且正确率很高，那么就称这个概念是强可学习的；一个概念，如果存在一个多项式的学习算法能够学习它，学习的正确率仅比随机猜测略好，那么就称这个概念是弱可学习的。非常有趣的是 `Schapire` 后来证明强可学习与弱可学习是等价的，也就是说，在 `PAC` 学习的框架下，一个概念是强可学习的充分必要条件是这个概念是弱可学习的。

这样一来，问题便成为，在学习中，如果已经发现了弱学习算法，那么能否将它提升为强学习算法。大家知道，发现弱学习算法通常要比发现强学习算法容易得多。那么如何具体实施提升，便成为开发提升方法时所要解决的问题。

与 `bagging` 不同， `boosting` 采用的是一个串行训练的方法。首先，它训练出一个弱分类器，然后在此基础上，再训练出一个稍好点的弱分类器，以此类推，不断的训练出多个弱分类器，最终再将这些分类器相结合，这就是 `boosting` 的基本思想，流程如下图：



可以看出，子模型之间存在强依赖关系，必须串行生成。 `boosting` 是利用不同模型的相加，构成一个更好的模型，求取模型一般都采用序列化方法，后面的模型依据前面的模型。

Adaboost原理

对提升方法来说，有两个问题需要回答：一是在每一轮如何改变训练数据的权值或概率分布；二是如何将弱分类器组合成一个强分类器。关于第 `1` 个问题，AdaBoost的做法是，提高那些被前一轮弱分类器错误分类样本的权值，而降低那些被正确分类样本的权值。这样一来，那些没有得到正确分类的数据，由于其

权值的加大而受到后一轮的弱分类器的更大关注。于是，分类问题被一系列的弱分类器“分而治之”。至于第 2 个问题，即弱分类器的组合，AdaBoost采取加权多数表决的方法，加大分类误差率小的弱分类器的权值，使其在表决中起较大的作用，减小分类误差率大的弱分类器的权值，使其在表决中起较小的作用。

Adaboost算法流程

AdaBoost 是 AdaptiveBoost 的缩写，表明该算法是具有适应性的提升算法。

算法的步骤如下：

1 . 给每个训练样本$(x_1, x_2, .., x_N)$分配权重，初始权重w_1均为$1/N$

2 . 针对带有权值的样本进行训练，得到模型G_m（初始模型为G_1）

3 . 计算模型G_m的误分率：

$e_m = \sum_i^N w_i I(y_i \neq G_M(X_i))$

其中：

$I(y_i \neq G_M(X_i)$

为指示函数，表示括号内成立时函数值为 1 ，否则为 0

4 . 计算模型G_m的系数：

$\alpha_m = \frac{1}{2}\log[\frac{1-e_m}{e_m}]$

5 . 根据误分率e和当前权重向量w_m更新权重向量：

$w_{m+1,i} = \frac{w_m}{z_m} exp(-\alpha_m y_i G_m(x_i))$

其中Z_m为规范化因子：

$z_m = \sum_{i=1}^m w_{mi} exp(-\alpha_m y_i G_m(x_i))$

6 . 计算组合模型$f(x) = \sum_{m=1}^M \alpha_m G_m(x_i)$的误分率

7 . 当组合模型的误分率或迭代次数低于一定阈值，停止迭代；否则，回到步骤 2

sklearn中的Adaboost

AdaBoostClassifier 是 skleran 中Adaboost分类器的实现， AdaBoostClassifier 的构造函数中有四个常用的参数可以设置：

- algorithm ：这个参数只有 AdaBoostClassifier 有。主要原因是scikit-learn 实现了两种 Adaboost 分类算法， SAMME 和 SAMME.R 。两者的主要区别是弱学习器权重的度量， SAMME.R 使用了概率度量的连续

值，迭代一般比 `SAMME` 快，因此 AdaBoostClassifier 的默认算法 `algorithm` 的值也是 `SAMME.R` 。

- `n_estimators` ：弱学习器的最大迭代次数。一般来说 `n_estimators` 太小，容易欠拟合， `n_estimators` 太大，又容易过拟合，一般选择一个适中的数值。默认是 `50` 。
- `learning_rate` ：AdaBoostClassifier 和 AdaBoostRegressor 都有，即每个弱学习器的权重缩减系数 `v` ，默认为 `1.0` 。
- `base_estimator` ：弱分类学习器或者弱回归学习器。理论上可以选择任何一个分类或者回归学习器，不过需要支持样本权重。我们常用的一般是 CART 决策树或者神经网络 MLP。

和 `sklearn` 中其他分类器一样，AdaBoostClassifier 类中的 `fit` 函数用于训练模型， `fit` 函数有两个向量输入：

- `X` ：大小为**[**样本数量,特征数量**]**的 `ndarray` ，存放训练样本
- `Y` ：值为整型，大小为**[**样本数量**]**的 `ndarray` ，存放训练样本的分类标签

AdaBoostClassifier 类中的 `predict` 函数用于预测，返回预测标签， `predict` 函数有一个向量输入：

`X` ：大小为**[**样本数量,特征数量**]**的 `ndarray` ，存放预测样本 AdaBoostClassifier的使用代码如下：

```
ada=AdaBoostClassifier(n_estimators=5,learning_rate=1.0)
ada.fit(train_data,train_label)
predict = ada.predict(test_data)
```

如果想动手实现 Adaboost 算法，并掌握如何使用 sklearn 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/xkv4b6yc/challenges

5.3.1 物以类聚人以群分---k Means

k Means是属于机器学习里面的非监督学习，通常是大家接触到的第一个聚类算法，其原理非常简单，是一种典型的基于距离的聚类算法。距离指的是每个样本到质心的距离。那么，这里所说的质心是什么呢？

其实，质心指的是样本每个特征的均值所构成的一个坐标。举个例子：假如有两个数据 $(1,1)$ 和$(2,2)$ 则这两个样本的质心为 $(1.5,1.5)$。

同样的，如果一份数据有 m 个样本，每个样本有 n 个特征，用 x_i^j 来表示第 j 个样本的第 i 个特征，则它们的质心为：$Cmass = (\frac{\sum_{j=1}^{m} x_1^j}{m}, \frac{\sum_{j=1}^{m} x_2^j}{m}, ..., \frac{\sum_{j=1}^{m} x_n^j}{m})$。

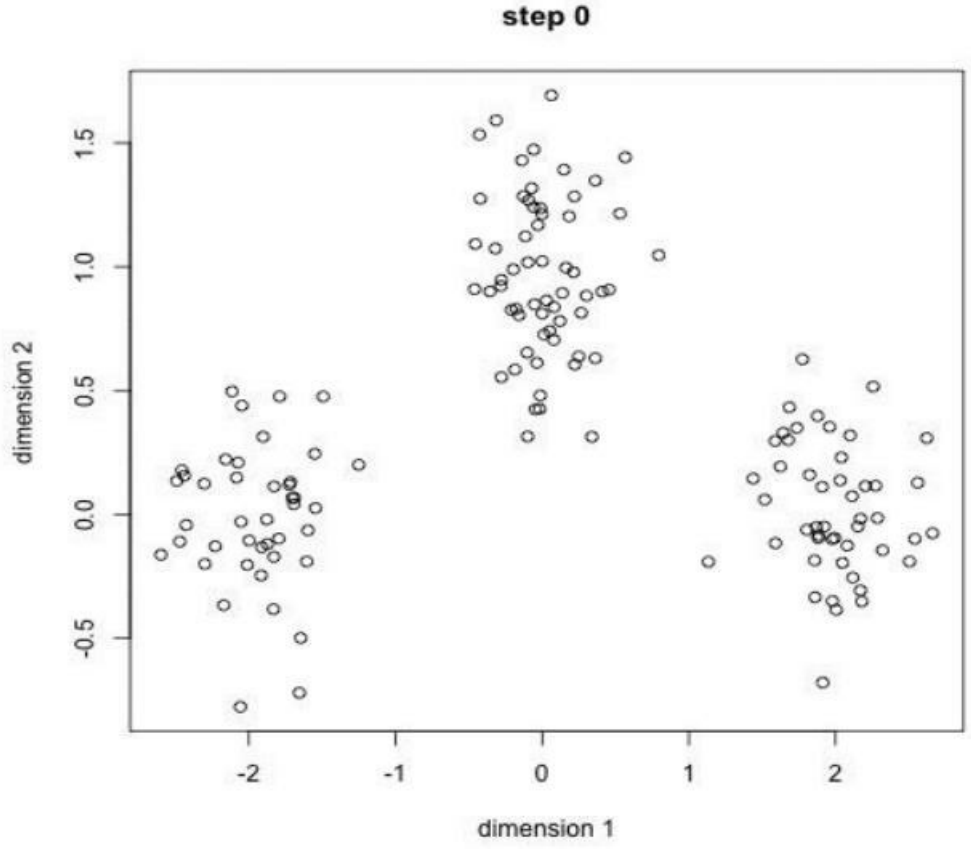
知道什么是质心后，就可以看看**k Means**算法的流程了。

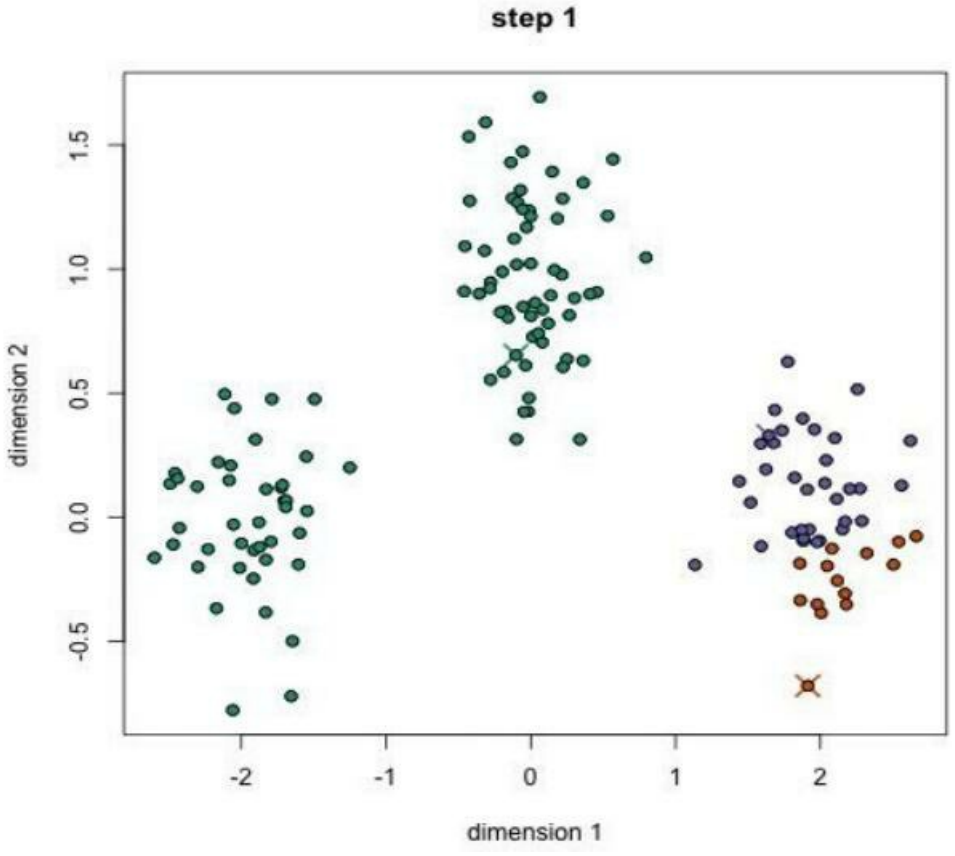
k Means算法流程

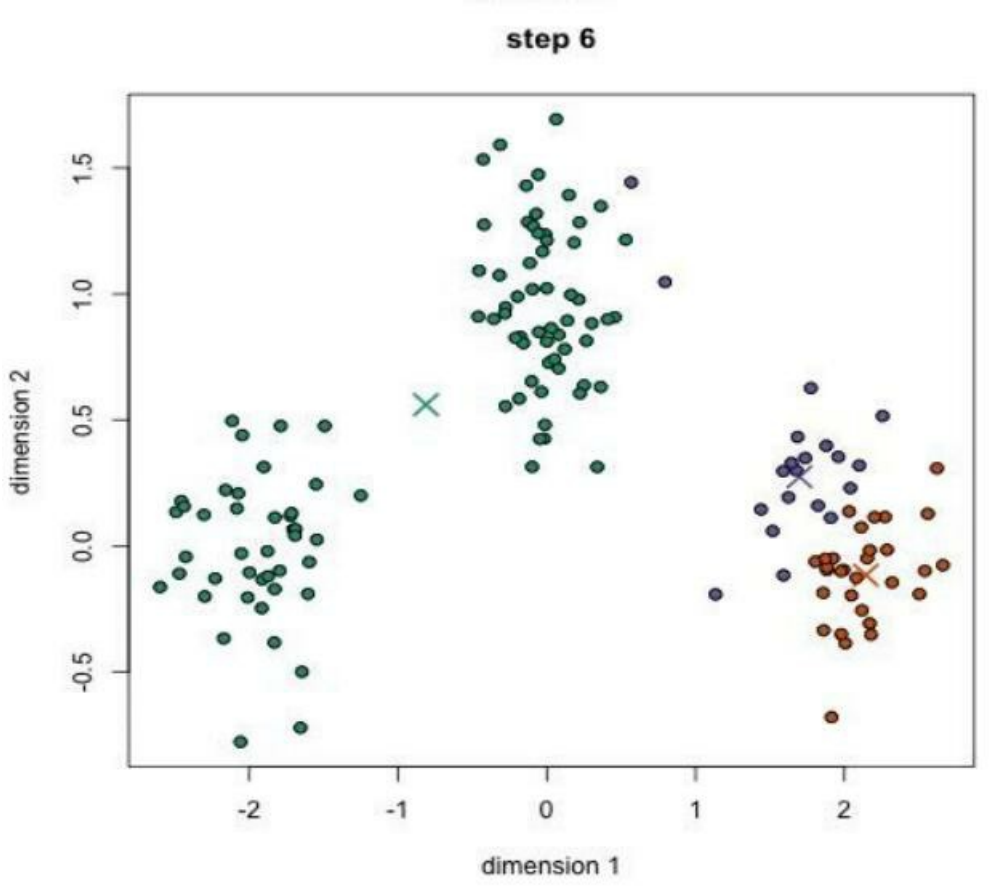
使用**k Means**来聚类时需要首先定义参数**k**，**k**的意思是我想将数据聚成几个类别。假设**k=3**，就是将数据划分成**3**个类别。接下来就可以开始**k Means**算法的流程了，流程如下：

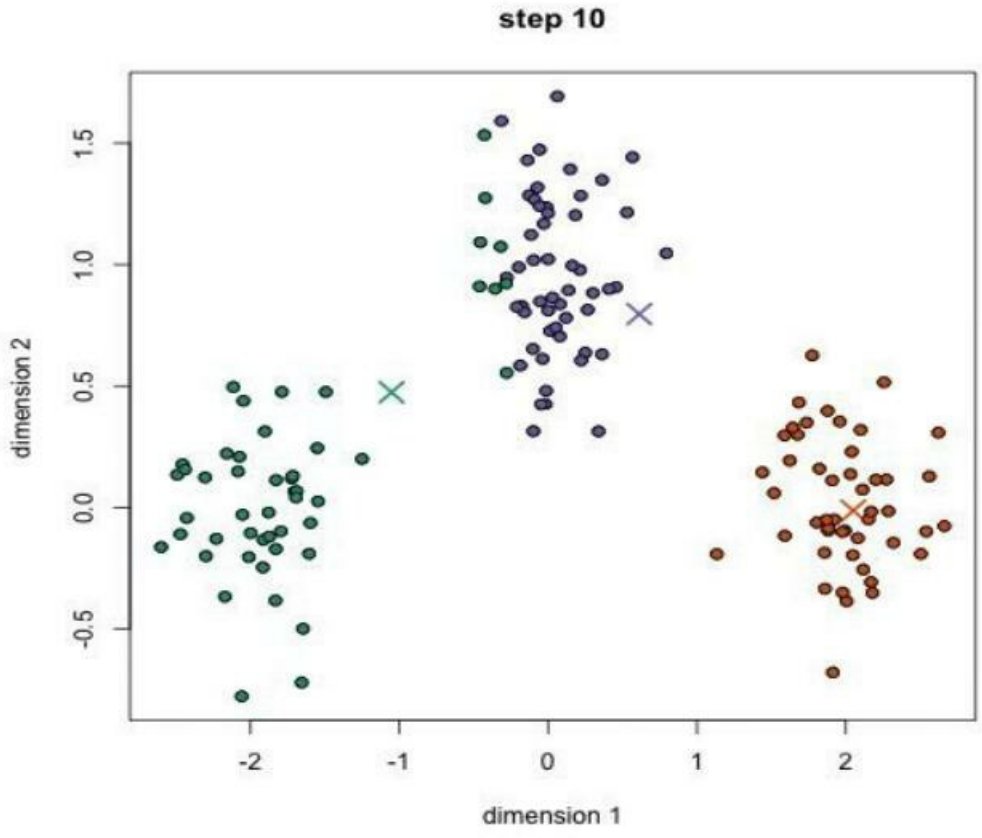
1. 随机初始**k**个样本，作为类别中心。 2. 对每个样本将其标记为距离类别中心最近的类别。 3. 将每个类别的质心更新为新的类别中心。 4. 重复步骤 2 、 3 ，直到类别中心的变化小于阈值。

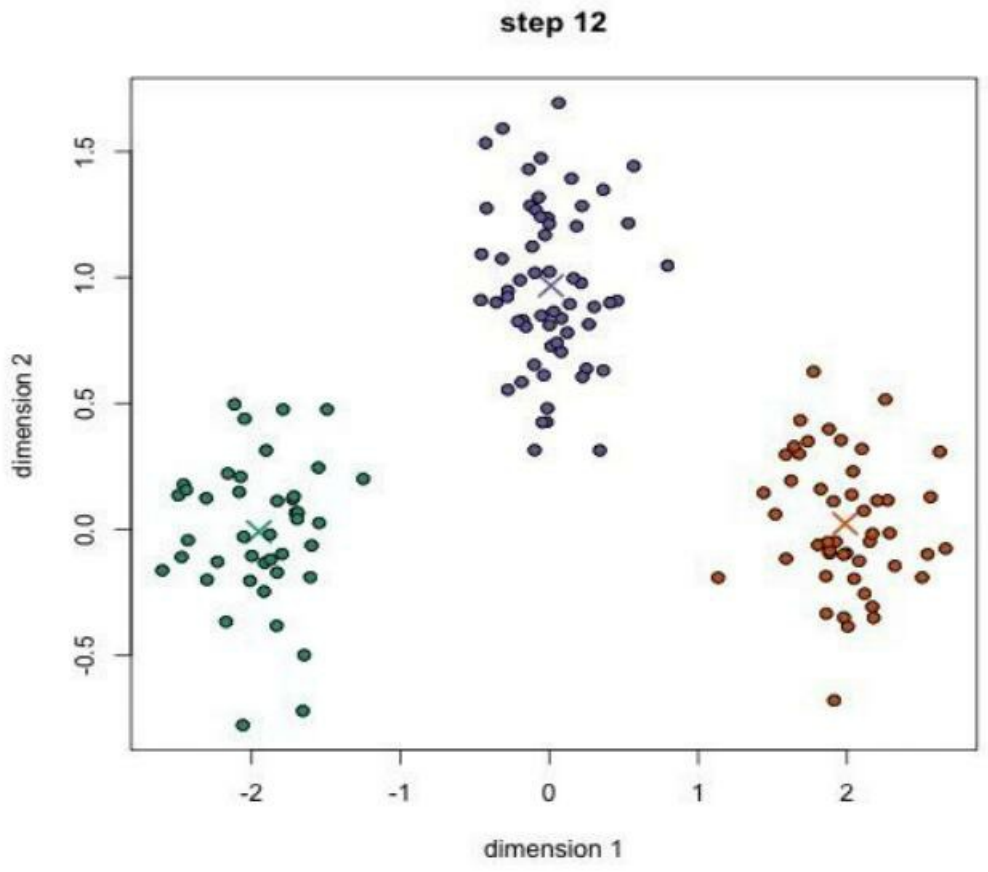
过程示意图如下（其中 X 表示类别的中心，数据点的颜色代表不同的类别，总共迭代 12 次，下图为部分迭代的结果）：


step 0
dimension 2
1.5
1.0
0.5
0.0
-0.5
-2
-1
0
1
2
dimension 1


step 1
dimension 2
1.5
1.0
0.5
0.0
-0.5
-2
-1
0
1
2
dimension 1


step 6
dimension 2
1.5
1.0
0.5
0.0
-0.5
-2
-1
0
1
2
dimension 1


step 10
dimension 2
1.5
1.0
0.5
0.0
-0.5
-2
-1
0
1
2
dimension 1



sklearn中的k Means

`KMeans` 是 `sklearn` 中k Means算法的实现， `KMeans` 的构造函数中有两个常用的参数可以设置：

- `n_clusters` ： 将结果聚成 `k` 个类。
- `random_state` :设置初始质心位置时，随机种子数值。

和 `sklearn` 中其他聚类器一样， `KMeans` 不允许对新的数据进行预测， `KMeans` 类中的 `fit_predict` 函数用于训练模型并获取聚类结果， `fit_predict` 函数有一个向量输入：

- `X` ： 大小为**[**样本数量,特征数量**]**的 `ndarray` ， 存放训练样本。

`KMeans` 的使用代码如下：

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters=5)
result = km.fit_predict(x)
```

如果想动手实现k Means算法，并掌握如何使用 sklearn 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/k6fp4saq/challenges

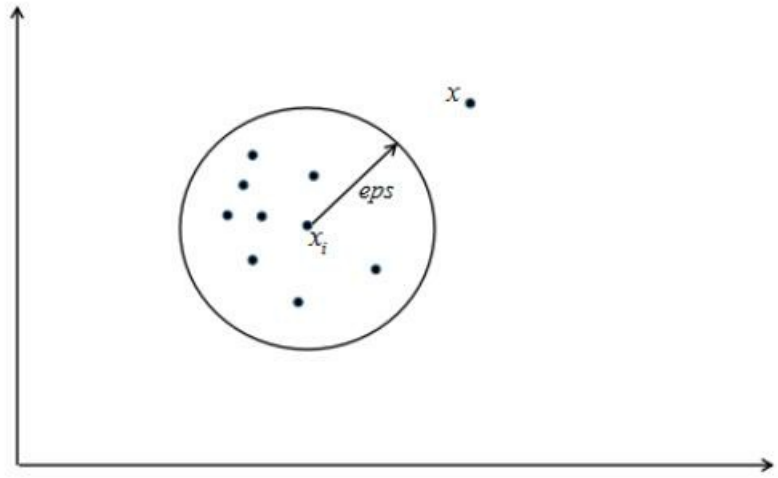
5.3.2 用密度来聚类---DBSCAN

基本概念

在 `DBSCAN` 算法中，有两个基本的领域参数，分别为 `eps` 邻域和 `Minpts` 。

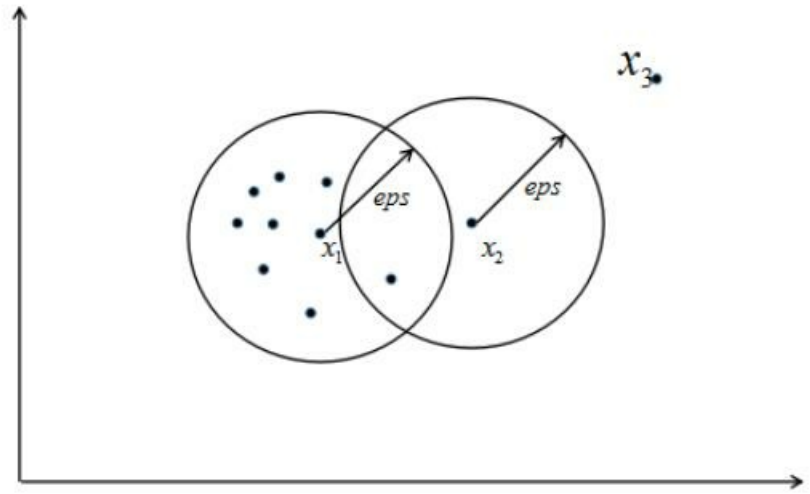
`eps` 邻域表示的是在数据集 `D` 中与样本点x_i的距离不大于 `eps` 的样本。

样本点x_i的 `eps` 邻域如图所示：



在图中，样本点x不在样本点x_i的 `eps` 邻域内。x_i的密度由x_i的 `eps` 邻域内的点的数量来估计。 `Minpts` 表示的是在样本点x_i的 `eps` 邻域内的最少样本点数目。基于邻域参数 **eps** 和 **Minpts**，在DBSCAN算法中将数据点分为一下三类：

1.核心点：若样本x_i的 `eps` 邻域内至少包含了 **Minpts** 个样本，则称样本点x_i为核心点。 2.边界点：若样本x_i的 `eps` 邻域内包含的样本数目小于 **Minpts**，但是它在其它核心点的邻域内，则称样本点x_i为边界点。 3.噪音点：指的是既不是核心点，又不是边界点的点。



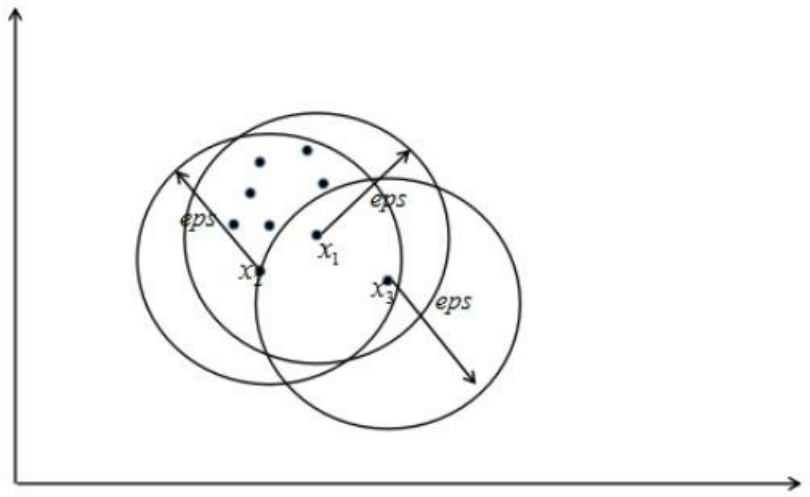
如上图，设置 **Minpts** 的值为 `8` ，对应的样本点x_1的 `eps` 邻域内包含了 `9` 个点，大于 **Minpts**，则样本点x_1为核心点。样本点x_2在样本点x_1的 `eps` 邻域内，且样本点x_2的 `eps` 邻域内只包含了 `2` 个样本点，小于 **Minpts**，则样本点x_2为边界点。样本点x_3为噪音点。

在 `DBSCAN` 算法中，还定义了如下的一些概念：

1.直接密度可达：若样本点x_j 在核心点x_i的 `eps` 邻域内，则称样本点x_j从样本点x_i直接密度可达。 2.密度可达：若在样本点x_1和样本点x_n之间存在一序列

$x_2, .., x_{n-1}$

且x_{i+1}从x_i直接密度可达，则称x_n从x_1密度可达。 3.密度相连：对于样本点x_i和样本点x_j，若存在样本点x_k，使得x_i和x_j都从x_k密度可达，则称x_i和x_j密度相连。



如上图，设置 **Minpts** 的值为 `8` ，则样本点x_1和x_2都是核心点，样本点x_3为边界点。样本点x_2在核心点x_1 `eps` 邻域内，则样本点x_2从样本点x_1直接密度可达。样本点x_3在在核心点x_2 `eps` 邻域内，则样本点x_3从样本点x_2直接密度可达。样本点x_1和x_3直接存在样本点x_2，且样本点x_2从样本点x_1直接密度可达，则样本点x_3从样本点x_1密度可达。

DBSCAN算法原理

基于密度的聚类算法通过寻找被低密度区域分离的高密度区域，并将高密度区域作为一个簇。在 `DBSCAN` 算法中，聚类簇定义为：由密度可达关系导出的最大的密度相连样本集合。

DBSCAN算法流程

在 `DBSCAN` 算法中，由核心对象出发，找到与该核心对象密度可达的所有样本形成一个聚类簇。 `DBSCAN` 算法流程如下：

`1` . 如果一个点的eps邻域包含多于MinPits个对象，则创建一个集合P作为核心对象的新簇。 `2` . 寻找核心对象的直接密度可达的对象，并合并为一个新的簇。 `3` . 直到没有点可以更新簇时算法结束。

DBSCAN算法优点

根据 `DBSCAN` 算法原理及流程可以发现， `DBSCAN` 算法在聚类时不需要自己设定簇的个数，而且能够发现任意形状的簇。还有一个优点就是， `DBSCAN` 算法对噪音点不敏感，所以 `DBSCAN` 算法也常用来找寻异常数据。

sklearn中的DBSCAN

`DBSCAN` 的构造函数中有两个常用的参数可以设置：

- `eps` ： `eps` 邻域半径大小
- `min_samples` ：即 Minpts， `eps` 邻域内样本最少数目

和 `sklearn` 中其他聚类器一样， `DBSCAN` 不允许对新的数据进行预测， `DBSCAN` 类中的 `fit_predict` 函数用于训练模型并获取聚类结果， `fit_predict` 函数有一个向量输入：

- `X` ：大小为 **[**样本数量,特征数量**]** 的 `ndarray` ，存放训练样本

`DBSCAN` 的使用代码如下：

```
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps=0.5,min_samples =10)
result = dbscan.fit_predict(x)
```

如果想动手实现 `DBSCAN` 算法，并掌握如何使用 `sklearn` 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/jfzo8xcu/challenges

5.3.3 分久必合---AGNES

`AGNES` 算法是一种聚类算法，最初将每个对象作为一个簇，然后这些簇根据某些距离准则被一步步地合并。两个簇间的相似度有多种不同的计算方法。聚类的合并过程反复进行直到所有的对象最终满足簇数目。所以理解 `AGNES` 算法前需要先理解一些距离准则。

为什么需要距离

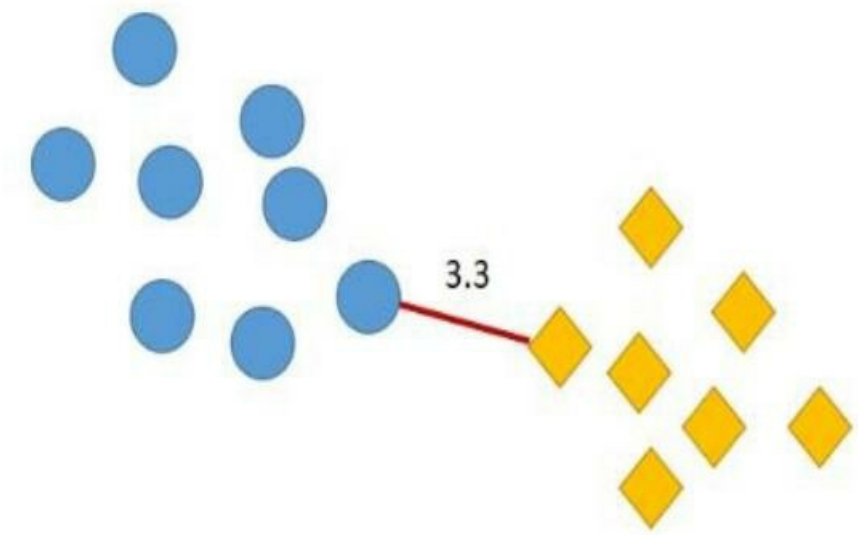
`AGNES` 算法是一种自底向上聚合的层次聚类算法，它先会将数据集中的每个样本看作一个初始簇，然后在算法运行的每一步中找出距离最近的两个簇进行合并，直至达到预设的簇的数量。所以 `AGNES` 算法需要不断的计算簇之间的距离，这也符合聚类的核心思想（物以类聚，人以群分），因此怎样度量两个簇之间的距离成为了关键。

距离的计算

衡量两个簇之间的距离通常分为最小距离、最大距离和平均距离。在 `AGNES` 算法中可根据具体业务选择其中一种距离作为度量标准。

最小距离

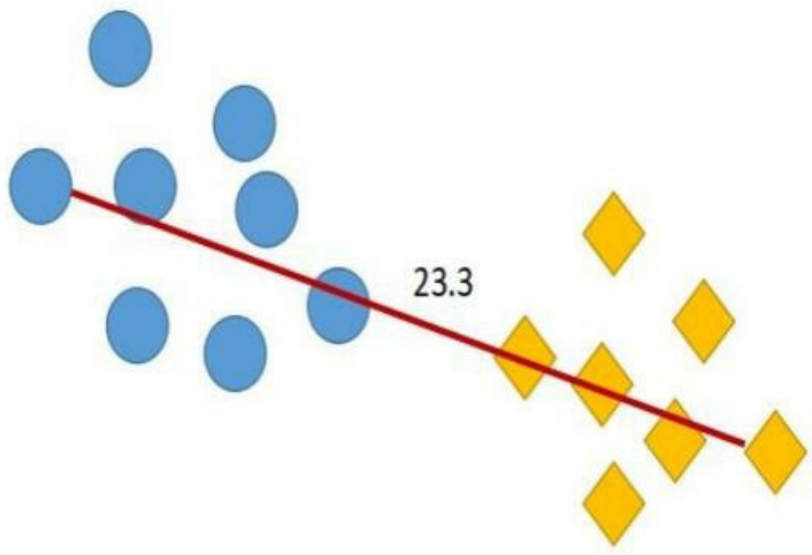
最小距离描述的是两个簇之间距离最近的两个样本所对应的距离。例如下图中圆圈和菱形分别代表两个簇，两个簇之间离得最近的样本的欧式距离为 `3.3` ，则最小距离为 `3.3` 。



假设给定簇C_i与C_j，则最小距离为：$d_{min} = min_{x \in i, z \in j} dist(x, z)$

最大距离

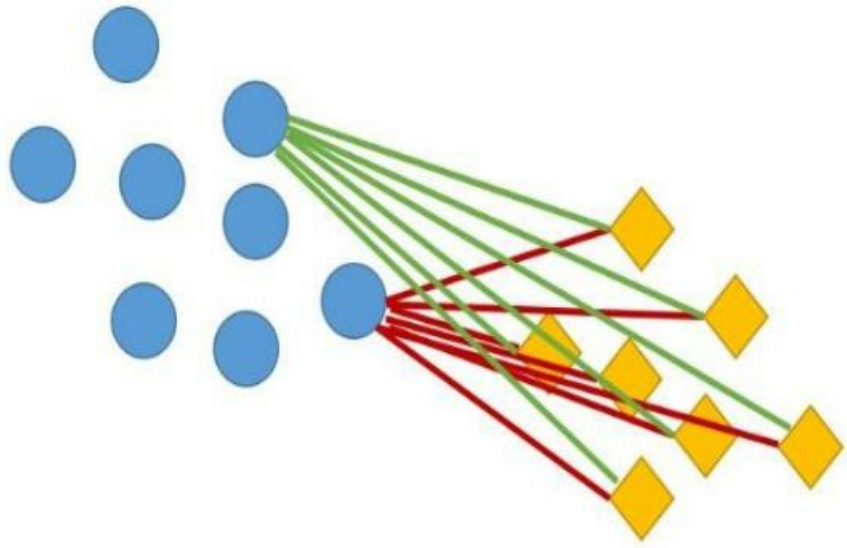
最大距离描述的是两个簇之间距离最远的两个样本所对应的距离。例如下图中圆圈和菱形分别代表两个簇，两个簇之间离得最远的样本的欧式距离为 `23.3` ，则最大距离为 `23.3` 。



假设给定簇C_i与C_j，则最大距离为：$d_{min} = max_{x \in i, z \in j} dist(x, z)$

平均距离

平均距离描述的是两个簇之间样本的平均距离。例如下图中圆圈和菱形分别代表两个簇，计算两个簇之间的所有样本之间的欧式距离并求其平均值。



假设给定簇C_i与C_j，$|C_i|, |C_j|$分别表示簇 i 与簇 j 中样本的数量，则平均距离为：

$d_{min} = \frac{1}{|C_i||C_j|} \sum_{x \in i} \sum_{z \in j} dist(x, z)$

AGNES算法流程

`AGNES` 算法是一种自底向上聚合的层次聚类算法，它先会将数据集中的每个样本看作一个初始簇，然后在算法运行的每一步中找出距离最近的两个簇进行合并，直至达到预设的簇的数量。

举个例子，现在先要将西瓜数据聚成两类，数据如下表所示：

编号	体积	重量
1	1.2	2.3
2	3.6	7.1
3	1.1	2.2
4	3.5	6.9
5	1.5	2.5

一开始，每个样本都看成是一个簇(`1` 号样本看成是 `1` 号簇， `2` 号样本看成是 `2` 号簇，...， `5` 号样本看成是 `5` 号簇)，假设簇的集合为 `C=[[1], [2], [3], [4], [5]]` 。

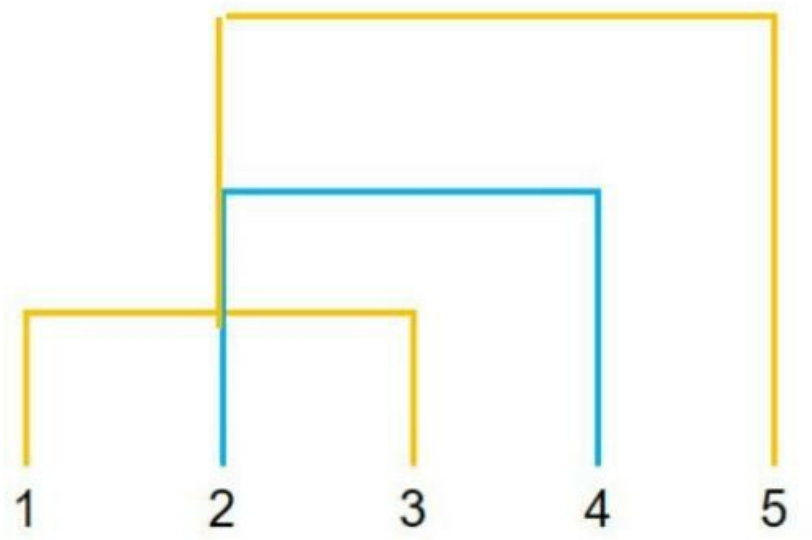
假设使用簇间最小距离来度量两个簇之间的远近，从表中可以看出 `1` 号簇与 `3` 号簇的簇间最小距离最小。因此需要将 `1` 号簇和 `3` 号簇合并，那么此时簇的集合 `C=[[1, 3], [2], [4], [5]]` 。

然后继续看这 `4` 个簇中哪两个簇之间的最小距离最小，我们发现 `2` 号簇与 `4` 号簇的最小距离最小，因此我们要进行合并，合并之后 `C=[[1, 3], [2, 4], [5]]` 。

然后继续看这 `3` 个簇中哪两个簇之间的最小距离最小，我们发现 `5` 号簇与 `[1, 3]` 簇的最小距离最小，因此我们要进行合并，合并之后 `C=[[1, 3, 5], [2, 4]]` 。

这个时候 `C` 中只有两个簇了，达到了我们的预期目标（想要聚成两类），所以算法停止。算法停止后会发现，我们已经将 `5` 个西瓜，聚成了两类，一类是小西瓜，另一类是大西瓜。

如果将整个聚类过程中的合并，与合并的次序可视化出来，就能看出为什么说 `AGNES` 是自底向上的层次聚类算法了。



所以 `AGNES` 伪代码如下：

```
#假设数据集为D，想要聚成的簇的数量为k
def AGNES(D, k):
    #C为聚类结果
```

```
    C = []
    #将每个样本看成一个簇
    for d in D:
        C.append(d)

    #C中簇的数量
    q=len(C)
    while q > k:
        寻找距离最小的两个簇a和b
        将a和b合并，并修改C
        q = len(C)
    return C
```

sklearn中的AGNES

`AgglomerativeClustering` 是 `sklearn` 中 `AGNES` 算法的实现， `AgglomerativeClustering` 的构造函数中有两个常用的参数可以设置：

- `n_clusters` ：将数据聚成 `n_clusters` 个类
- `linkage` ：设置 `AGNES` 聚类时使用最小簇间距离、最大簇间距离还是平均距离。传入 `ward` 表示最小簇间距离，传入 `complete` 表示最大簇间距离，传入 `average` 表示平均距离

`AgglomerativeClustering` 类中的 `fit_predict` 函数用于训练模型并获取聚类结果， `fit_predict` 函数有一个向量输入：

- `X` ：数据集，形状为**[**样本数量,特征数量**]**的 `ndarray`

`AgglomerativeClustering` 的使用代码如下：

```
from sklearn.cluster import AgglomerativeClustering
agnes = AgglomerativeClustering(k=5)

# 注意：x为ndarray
result = agnes.fit_predict(x)
```

如果想动手实现 `AGNES` 算法，并掌握如何使用 `sklearn` 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/qy9gozt8/challenges

5.3.4 最重要的才是我想要的---PCA

维数灾难

在机器学习中，我们不仅需要学习怎样进行分类、回归或者聚类，我们更要学习怎样对数据进行更好的处理，使得我们的数据能够更好的为我们的机器学习算法服务。而降维就是数据处理中的一环。

说到降维，那首先就要提到一个概念：维数灾难。维数灾难其实很好理解，举个例子。

我们现在玩个游戏，我告诉你一些信息，你猜一猜我所描述的是什么。

- 我：这个能在地球上才有，而且是犬科动物。
- 您：.....

如果您现在是一个动物的分类器，我相信您仅仅靠这两个特征（地球上才有，犬科动物）不大可能能够预测出我所说的是什么。也就是说，不管你用什么算法去分类，都很有可能发生欠拟合的现象。

- 我：这个是犬科动物，喜欢啃骨头，长得像狼， 比较二。
- 您：哈士奇！
- 我：猜的挺准。

当我给出的信息比较合适（这次有`4`个特征），您可能能够猜到所提供的特征数据所描述的是哈士奇。这个时候我们的分类算法能正常工作。

- 我：这个能在地球上才有，是犬科动物，有毛，有爪子，体型大，耳尖呈圆形，尾巴喜欢上翘，长得像狼，喜欢啃骨头，有时比较二但挺忠诚。
- 您：哈士奇！
- 我：不，我说的是阿拉斯加。
- 您：.....

这次我提供的信息比上面个两次都多（这次有`10`个特征），但是您可能将阿拉斯加误判成哈士奇。因为您可能看到长得像狼和比较二就认为是哈士奇了，也就是发生了过拟合的现象。这也说明了不是说数据的特征数量越多，我们的机器学习算法的效果就越强。当数据的特征数量变大时，和可能会造成机器学习算法的模型变得非常复杂，从而导致过拟合。而且如果我所提供的特征数量越多，比如有`10000`个特征，那么算法的训练过程中的时间成本会非常大。

所以维数灾难通常是指对于已知样本数目，存在一个特征数目的最大值，当实际使用的特征数目超过这个最大值时，机器学习算法的性能不是得到改善，而是退化。

降维

既然维数太大可能引发维数灾难，那么如果能有算法能够自动的帮我们把重要性比较高的特征维度保留下来，把其他的维度过滤掉就好了。那这个过程我们称之为降维。

从维数灾难的概念出发，我们就能知道降维的作用了。

- 降低机器学习算法的时间复杂度
- 节省了提取不必要特征的开销

- 缓解因为维数灾难所造成的过拟合现象

PCA与降维

降维的方法有很多，而最为常用的就是 `PCA` (主成分分析)。 `PCA` 是将数据从原来的坐标系转换到新的坐标系，新的坐标系的选择是由数据本身决定的。第一个新坐标轴选择的是原始数据中方差最大的方向，第二个新坐标轴的选择和第一个坐标轴正交且方差最大的方向。然后该过程一直重复，重复次数为原始数据中的特征数量。最后会发现大部分方差都包含在最前面几个新坐标轴中，因此可以忽略剩下的坐标轴，从而达到降维的目的。

PCA的算法流程

`PCA` 在降维时，需要指定将维度降至多少维，假设降至 `k` 维，则 `PCA` 的算法流程如下：

1. `demean`
2. 计算数据的协方差矩阵
3. 计算协方差矩阵的特征值与特征向量
4. 按照特征值，将特征向量从大到小进行排序
5. 选取前 `k` 个特征向量作为转换矩阵
6. `demean` 后的数据与转换矩阵做矩阵乘法获得降维后的数据

其中 **`demean`** ，协方差矩阵，特征值与特征向量的相关知识如下：

demean

`demean` 又称为零均值化，意思是将数据中每个维度上的均值变成 `0` 。那为什么要这样做呢？ `PCA` 实质上是找方差最大的方向，而方差的公式如下(其中μ为均值)：

$$Var(x) = \frac{1}{n}\sum_{i=1}^{n}(x-\mu)^2$$

如果将均值变成 `0` ，那么方差计算起来就更加方便，如下：

$$Var(x) = \frac{1}{n}\sum_{i=1}^{n}(x)^2$$

在 `numpy` 中想要 `demean` 很简单，代码如下：

```
import numpy as np

#计算样本各个维度的均值
u = np.mean(data, axis=0)
#demean
after_demean = data - u
```

协方差矩阵

协方差描述的是两个特征之间的相关性，当协方差为正时，两个特征呈正相关关系（同增同减）；当协方差为负时，两个特征呈负相关关系（一增一减）；当协方差为**0**时，两个特征之间没有任何相关关系。

协方差的数学定义如下(假设样本有 `x` 和 `y` 两种特征，而 `X` 就是包含所有样本的 `x` 特征的集合， `Y` 就是包含所有样本的 `y` 特征的集合)：

$$conv(X,Y) = \frac{\sum_{i=1}^{n}(x_i-\mu_x)\sum_{i=1}^{n}(y_i-\mu_y)}{n-1}$$

如果在算协方差之前做了 `demean` 操作，那么公式则为：

$$conv(X,Y) = \frac{\sum_{i=1}^{n}x_i\sum_{i=1}^{n}y_i}{n-1}$$

假设样本只有 `X` 和 `Y` 这两个特征，现在把 `X` 与 `X` ， `X` 与 `Y` ， `Y` 与 `X` ， `Y` 与 `Y` 的协方差组成矩阵，那么就构成了协方差矩阵。而协方差矩阵反应的就是特征与特征之间的相关关系。

	X	**Y**
X	conv(X,X)	conv(X,Y)
Y	conv(Y,X)	conv(Y,Y)

`NumPy` 提供了计算协方差矩阵的函数 `cov` ，示例代码如下：

```
import numpy as np

# 计算after_demean的协方差矩阵
# after_demean的行数为样本个数，列数为特征个数
# 由于cov函数的输入希望是行代表特征，列代表数据的矩阵，所以要转置
cov = np.cov(after_demean.T)
```

特征值与特征向量

特征值与特征向量的数学定义：如果向量 `v` 与矩阵 `A` 满足 `Av=λv` ，则称向量 `v` 是矩阵**A**的一个特征向量， `λ` 是相应的特征值。

因为协方差矩阵为方阵，所以我们可以计算协方差矩阵的特征向量和特征值。其实这里的特征值从某种意义上来说体现了方差的大小，特征值越大方差就越大。而特征值所对应的特征向量就代表将原始数据进行坐标轴转换之后的数据。

`numpy` 为我们提供了计算特征值与特征向量的接口 `eig` ，示例代码如下：

```
import numpy as np

#eig函数为计算特征值与特征向量的函数
#cov为矩阵，value为特征值，vector为特征向量
value, vector = np.linalg.eig(cov)
```

因此， `PCA` 算法伪代码如下：

```
#假设数据集为D，PCA后的特征数量为k
def pca(D, k):
    after_demean=demean(D)
    计算after_demean的协方差矩阵cov
    value, vector = eig(cov)
    根据特征值value将特征向量vector降序排序
    筛选出前k个特征向量组成映射矩阵P
    after_demean和P做矩阵乘法得到result
```

```
    return result
```

sklearn中的PCA

`PCA` 的构造函数中有一个常用的参数可以设置：

- `n_components` ：表示想要将数据降维至 `n_components` 个维度

`PCA` 类中有三个常用的函数分别为： `fit` 函数用于训练 `PCA` 模型； `transform` 函数用于将数据转换成降维后的数据，当模型训练好后，对于新输入的数据，也可以用 `transform` 方法来降维； `fit_transform` 函数用于使用数据训练 `PCA` 模型，同时返回降维后的数据。

其中 `fit` 函数中的参数:

- `X` ：大小为 `[样本数量,特征数量]` 的 `ndarray` ，存放训练样本

`transform` 函数中的参数：

- `X` ：大小为 `[样本数量,特征数量]` 的 `ndarray` ，存放训练样本

`fit_transform` 函数中的参数：

- `X` ：大小为 `[样本数量,特征数量]` 的 `ndarray` ，存放训练样本

`PCA` 的使用代码如下：

```
from sklearn.decomposition import PCA

#构造一个将维度降至11维的PCA对象
pca = PCA(n_components=11)
#对数据X进行降维，并将降维后的数据保存至newX
newX = pca.fit_transform(X)
```

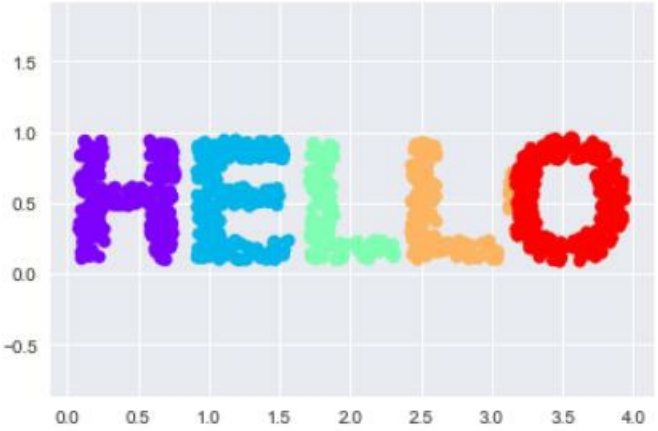
如果想动手实现 `PCA` 算法，并掌握如何使用 `sklearn` 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/rbnaxywe/challenges

5.3.5 不忘初心---多维缩放

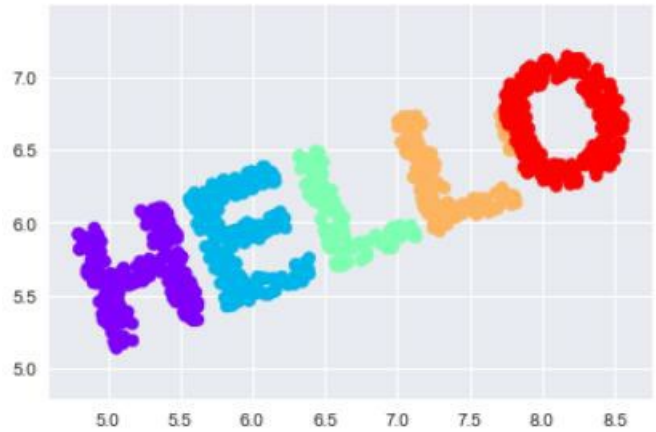
多维缩放（Multiple Dimensional Scaling,MDS）是一种经典的降维方法。其主要思想是保持样本在原空间和低维空间的距离不变。因为距离是样本之间的一个很好的分离属性，降维后，保持距离不变，那么就相当于保持了样本的相对空间关系不变。

MDS算法思想

假如，一份数据可视化后结果如下：



则将数据进行旋转，数据的特征值发生改变，但是每个点与数据中其他点的距离并没有发生改变：



所以，`MDS`算法认为，在数据样本中，每个样本的每个特征值并不是数据间关系的必要特征，真正的基础特征是每个点与数据集中其他点的距离。

假设存在距离矩阵，它的第`i`行第`j`列为$dist_{ij}$表示样本`i`到样本`j`之间的距离，我们要通过样本的坐标计算出距离矩阵非常简单。但是，反过来，我们想通过距离矩阵还原出每个样本的坐标就很困难了，而`MDS`算法就是用来解决这个问题的。它可以将一个数据集的距离矩阵还原成一个`D`维坐标来表示数据集。

MDS算法推导

假设存在 `m` 个样本，在原始空间中的距离矩阵$D \in R^{m \times m}$,其第 `i` 行 `j` 列为样本 `i` 到样本 `j` 之间的距离。目标是获得样本在d'维空间中的欧式距离等于原空间的欧氏距离，即：

$||z_i - z_j|| = dist_{ij}$

令$B = Z^T Z \in R^{m \times m}$，其中 `B` 为降维后的内积矩阵，$b_{ij} = z_i^T z_j$，则：

$dist_{ij}^2 = ||z_i - z_j||^2 = ||z_i||^2 + ||z_j||^2 - 2z_i^T z_j$

令降维后的样本被中心化，则：

$\sum_{i=1}^{m} dist_{ij}^2 = tr(B) + mb_{jj}$

$\sum_{j=1}^{m} dist_{ij}^2 = tr(B) + mb_{ii}$

$\sum_{i=1}^{m} \sum_{j=1}^{m} dist_{ij}^2 = 2mtr(B)$

令 $dist_{i.}^2 = \frac{1}{m} \sum_{j=1}^{m} dist_{ij}^2 ...(1)$

$dist_{.j}^2 = \frac{1}{m} \sum_{i=1}^{m} dist_{ij}^2 ...(2)$

$dist_{..}^2 = \frac{1}{m^2} \sum_{i=1}^{m} \sum_{j=1}^{m} dist_{ij}^2 ...(3)$

则

$b_{ij} = -\frac{1}{2}(dist_{ij}^2 - dist_{i.}^2 - dist_{.j}^2 + dist_{..}^2)$

由此即可通过降维前后保持不变的距离矩阵求取内积矩阵。

再将内积矩阵做特征分解，取 `d` 个最大的特征值所构成的对角矩阵Λ，并求相应的特征向量矩阵V，最后计算出 `Z`：

$Z = V\Lambda^{\frac{1}{2}}$

MDS算法流程

`1`. 计算式子 `1`， `2`， `3` `2`. 计算矩阵 `B` `3`. 对矩阵 `B` 进行特征分解 `4`. 取 `d` 个最大特征值所构成的对角矩阵，并求相应的特征向量矩阵 `5`. 计算矩阵 `Z`

sklearn中的多维缩放

在降维时， `MDS` 的构造函数中有一个常用的参数可以设置：

- `n_components`：即我们进行 `MDS` 降维时降到的维数。在降维时需要输入这个参数。

`MDS` 类中的 `fit_transform` 函数用于训练模型并进行降维，返回降维后的数据，`fit_transform` 函数有一个向量输入：

- `X`：大小为**[**样本数量,特征数量**]**的 `ndarray`，存放需降维的样本

`MDS` 的使用代码如下：

```
from sklearn.manifold import MDS
mds = MDS(2)
Z = mds.fit_transform(X)
```

如果想动手实现多维缩放算法，并掌握如何使用 `sklearn` 来解决实际问题，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/vry7x6op/challenges

6 综合实战

纸上得来终觉浅，绝知此事要躬行。当我们学会了机器学习算法原理，知道了怎样使用 `sklearn` 来快速编写机器学习程序之后，需要进行大量的实战演练来提升我们的技能和实操能力。所以在本章中提供了两个综合实战案例：一个是 `kaggle` 中最为经典的也是最适合入门的机器学习比赛---泰坦尼克生还预测。另一个是使用图像处理技术让程序自动识别人脸性别。希望通过这两个实战案例，能够提升您对机器学习的兴趣以及使用机器学习技术解决实际问题的能力。

6.1.1 泰坦尼克生还问题简介

泰坦尼克号的沉船事件是是历史上最臭名昭著的沉船事件之一。`1912`年`4`月`15`日，泰坦尼克在航线中与冰山相撞，`2224`名乘客中有`1502`名乘客丧生。

泰坦尼克号数据集是目标是给出一个模型来预测某位泰坦尼克号的乘客在沉船事件中是生还是死。而且该数据集是一个非常好的数据集，能够让您快速的开始数据科学之旅。

泰坦尼克生还预测实训已在`educoder`平台上提供，若感兴趣可以输入链接进行体验。

链接：https://www.educoder.net/shixuns/kz3fixv9/challenges

6.1.3 特征工程

什么是特征工程？其实每当我们拿到数据时，并不是所有的特征都是有用的，可能有许多冗余的特征需要删掉，或者根据 `EDA` 的结果，我们可以根据已有的特征来添加新的特征，这其实就是特征工程。

接下来我们来尝试对一些特征进行处理。

年龄离散化

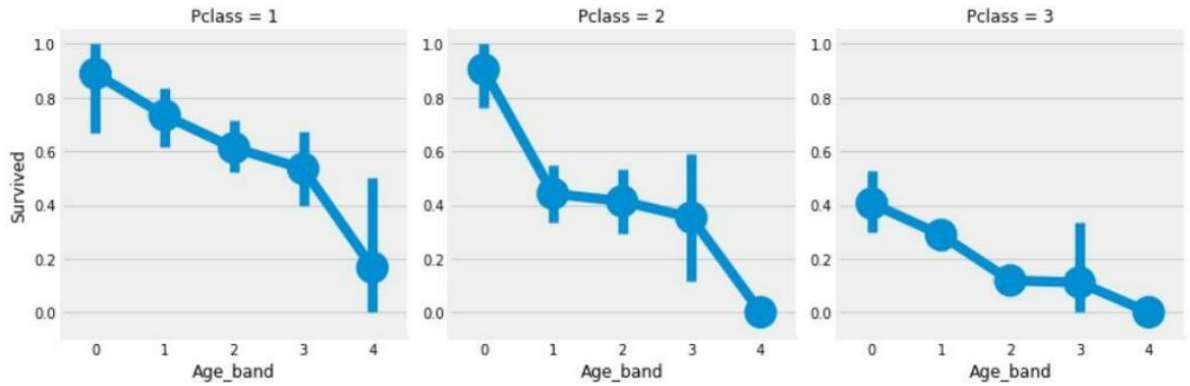
年龄是一个连续型的数值特征，有的机器学习算法对于连续性数值特征不太友好，例如决策树、随机森林等 `tree-base model` 。所以我们可以考虑将年龄转换成年龄段。例如将年龄小于 `16` 的船客置为 `0` ， `16` 到 `32` 岁之间的置为 `1` 等。

```
data['Age_band']=0
data.loc[data['Age']<=16,'Age_band']=0
data.loc[(data['Age']>16)&(data['Age']<=32),'Age_band']=1
data.loc[(data['Age']>32)&(data['Age']<=48),'Age_band']=2
data.loc[(data['Age']>48)&(data['Age']<=64),'Age_band']=3
data.loc[data['Age']>64,'Age_band']=4
```

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Initial	Age_band
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S	Mr	1
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C	Mrs	2

我们可以看一下转换成年龄段后，年龄段与生还率的关系。

```
sns.factorplot('Age_band','Survived',data=data,col='Pclass')
plt.show()
```



可以看出和我们之前 `EDA` 的结果相符，年龄越大，生还率越低。

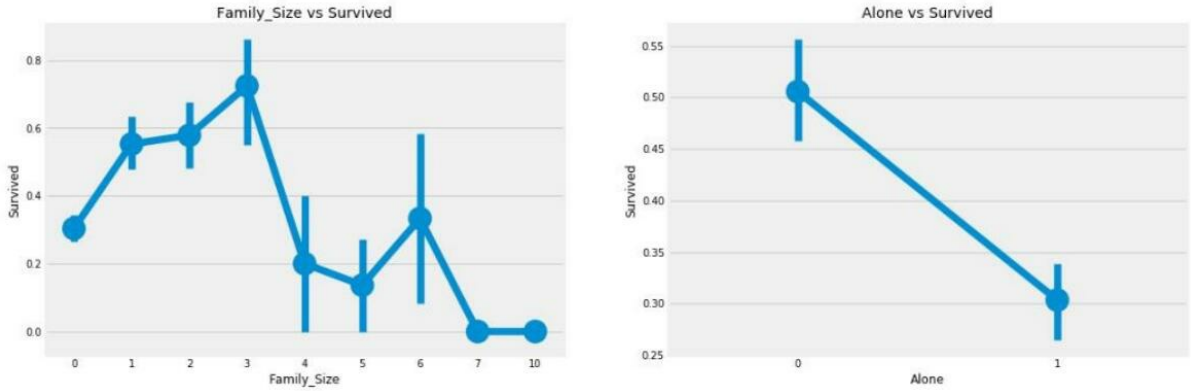
家庭成员数量与是否孤身一人

由于家庭成员数量和是否孤身一人好像对于是否生还有影响，所以我们不妨添加新的特征。

```
data['Family_Size']=0
data['Family_Size']=data['Parch']+data['SibSp']
data['Alone']=0
data.loc[data.Family_Size==0,'Alone']=1
```

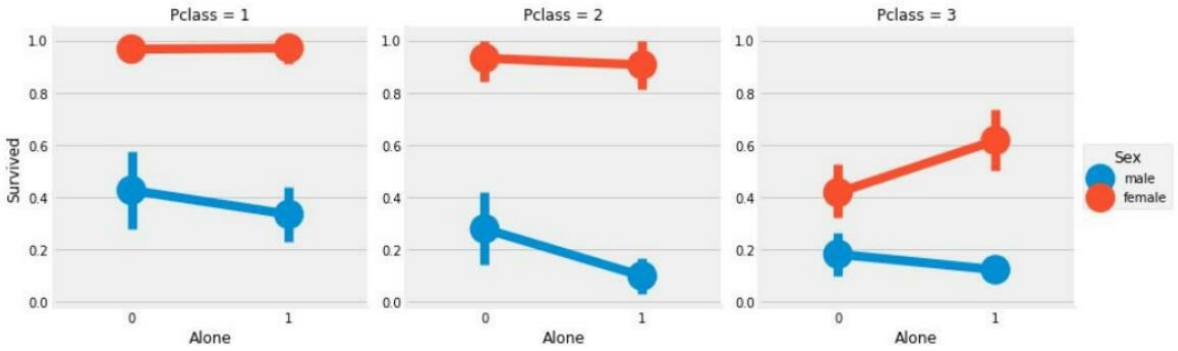
然后再可视化看一下

```
f,ax=plt.subplots(1,2,figsize=(18,6))
sns.factorplot('Family_Size','Survived',data=data,ax=ax[0])
ax[0].set_title('Family_Size vs Survived')
sns.factorplot('Alone','Survived',data=data,ax=ax[1])
ax[1].set_title('Alone vs Survived')
plt.close(2)
plt.close(3)
plt.show()
```



从图中可以很明显的看出，如果你是一个人，那么生还的几率比较低，而且对于人数大于 `4` 人的家庭来说生还率也比较低。感觉，这可能也是一个比较好的特征，可以再深入的看一下。

```
sns.factorplot('Alone','Survived',data=data,hue='Sex',col='Pclass')
plt.show()
```

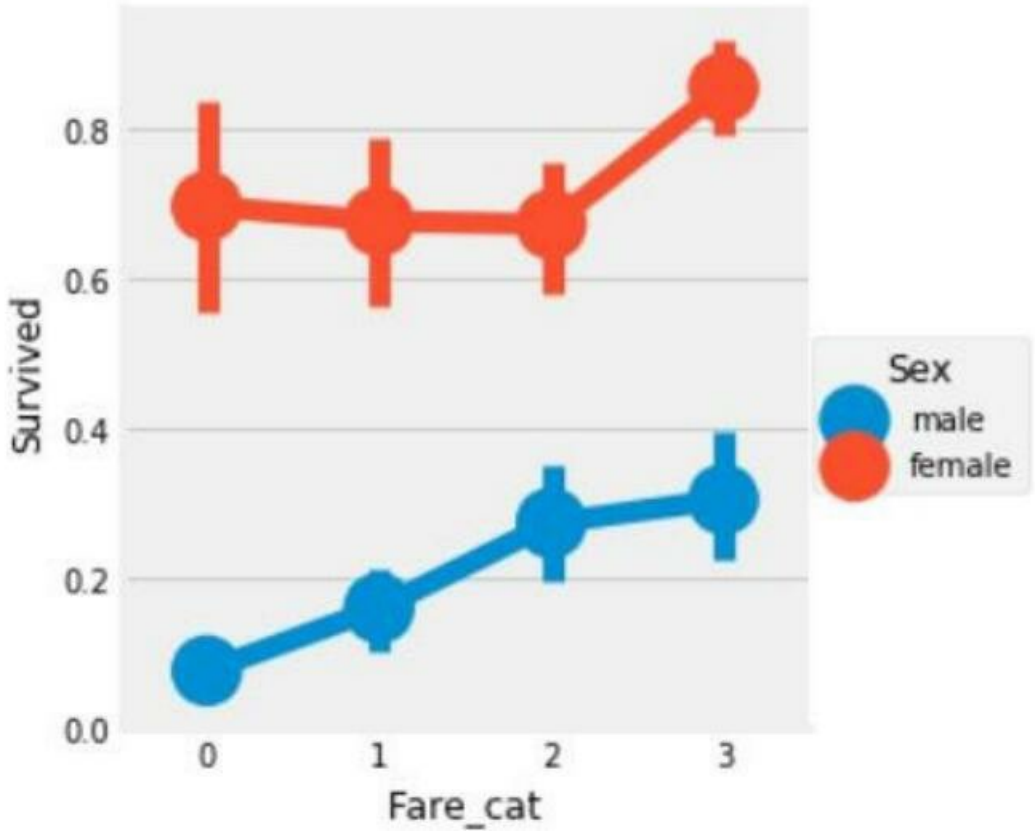


可以看出，除了三等舱的单身女性的生还率比非单身女性的生还率高外，单身并不是什么好事。

花费离散化

和年龄一样，花费也是一个连续性的数值特征，所以我们不妨将其离散化。

```
data['Fare_cat']=0
data.loc[data['Fare']<=7.91,'Fare_cat']=0
data.loc[(data['Fare']>7.91)&(data['Fare']<=14.454),'Fare_cat']=1
data.loc[(data['Fare']>14.454)&(data['Fare']<=31),'Fare_cat']=2
data.loc[(data['Fare']>31)&(data['Fare']<=513),'Fare_cat']=3

sns.factorplot('Fare_cat','Survived',data=data,hue='Sex')
plt.show()
```



很明显，花费越多生还率越高，金钱决定命运。

将字符串特征转换为数值型特征

由于我们的机器学习模型不支持字符串，所以需要将一些有用的字符串类型的特征转换成数值型的特征，比如：性别，口岸，姓名前缀。

```
data['Sex'].replace(['male','female'],[0,1],inplace=True)
data['Embarked'].replace(['S','C','Q'],[0,1,2],inplace=True)
data['Initial'].replace(['Mr','Mrs','Miss','Master','Other'],[0,1,2,3,4],inplace=True)
```

删掉没多大用处的特征

- 姓名：难道姓名和生死有关系？这也太玄乎了，我不信，所以把它删掉。
- 年龄：由于已经根据年龄生成了新的特征“年龄段”，所以这个特征也需要删除。
- 票：票这个特征感觉是一堆随机的字符串，所以删掉。
- 花费：和年龄一样，删掉。
- 船舱：由于有很多缺失值，不好填充，所以可以考虑删掉。
- 船客 `ID` ： `ID` 和生死应该没啥关系，所以删掉。

```
data.drop(['Name','Age','Ticket','Fare','Cabin','PassengerId'],axis=1,inplace=True)
```

6.1.4 构建模型进行预测

做好数据预处理后，可以将数据喂给机器学习模型来进行训练和预测了。不过在构建模型之前，要使用处理训练集数据的方式来处理测试集。

```
test_data=pd.read_csv('./Titanic/test.csv')

test_data['Initial']=0
for i in test_data:
    test_data.loc[:, 'Initial'] = test_data.Name.str.extract('([A-Za-z]+)\.',expand=False) #lets extract the
Salutations

test_data.loc[:, 'Initial'].replace(['Mlle','Mme','Ms','Dr','Major','Lady','Countess','Jonkheer','Col','Rev',
'Capt','Sir','Don'],['Miss','Miss','Miss','Other','Mr','Mrs','Mrs','Other','Other','Other','Mr','Mr','Mr'],in
place=True)

test_data.loc[(test_data.Age.isnull())&(test_data.Initial=='Mr'),'Age']=33
test_data.loc[(test_data.Age.isnull())&(test_data.Initial=='Mrs'),'Age']=36
test_data.loc[(test_data.Age.isnull())&(test_data.Initial=='Miss'),'Age']=22
test_data.loc[(test_data.Age.isnull())&(test_data.Initial=='Other'),'Age']=46

test_data['Embarked'].fillna('S', inplace=True)

test_data['Age_band']=0
test_data.loc[test_data['Age']<=16,'Age_band']=0
test_data.loc[(test_data['Age']>16)&(test_data['Age']<=32),'Age_band']=1
test_data.loc[(test_data['Age']>32)&(test_data['Age']<=48),'Age_band']=2
test_data.loc[(test_data['Age']>48)&(test_data['Age']<=64),'Age_band']=3
test_data.loc[test_data['Age']>64,'Age_band']=4

test_data['Family_Size']=0
test_data['Family_Size']=test_data['Parch']+test_data['SibSp']+1
test_data['Alone']=0
test_data.loc[test_data.Family_Size==1,'Alone']=1

test_data['Fare_cat']=0
test_data.loc[test_data['Fare']<=7.91,'Fare_cat']=0
test_data.loc[(test_data['Fare']>7.91)&(test_data['Fare']<=14.454),'Fare_cat']=1
test_data.loc[(test_data['Fare']>14.454)&(test_data['Fare']<=31),'Fare_cat']=2
test_data.loc[(test_data['Fare']>31)&(test_data['Fare']<=513),'Fare_cat']=3

test_data['Sex'].replace(['male','female'],[0,1],inplace=True)
test_data['Embarked'].replace(['S','C','Q'],[0,1,2],inplace=True)
test_data['Initial'].replace(['Mr','Mrs','Miss','Master','Other'],[0,1,2,3,4],inplace=True)
test_data['Cabin'].replace(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'T'], [0, 1, 2, 3, 4, 5, 6, 7], inplace=True)

test_data.drop(['Name','Age','Ticket','Fare','Fare_Range','PassengerId'],axis=1,inplace=True)
```

然后可以使用机器学习模型来训练并预测了，这里使用的是随机森林。

```
Y_train = data['Survived']
X_train = data.drop(['Survived'], axis=1)

Y_test = test_data['Survived']
X_test = test_data.drop(['Survived'], axis=1)
```

```
clf = RandomForestClassifier(n_estimators=10)
clf.fit(X_train, Y_train)
predict = clf.predict(X_test)
print(accuracy_score(Y_test, predict))
```

此时看到预测的准确率达到了 0.8275 。

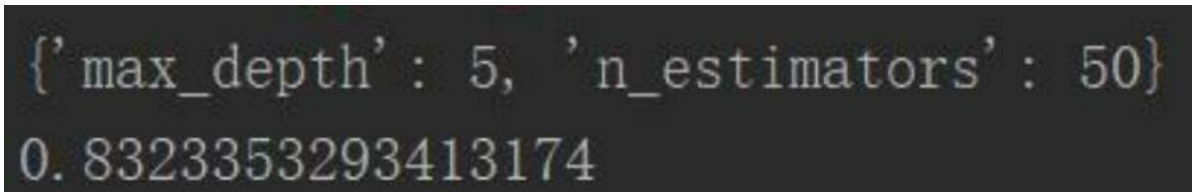
5.1.5 调参

很多机器学习算法有很多可以调整的参数(即超参数)，例如我们用的随机森林需要我们指定森林中有多少棵决策树，每棵决策树的最大深度等。这些超参数都或多或少的会影响这模型的性能。那么怎样才能找到合适的超参数，来让我们的模型性能达到比较好的效果呢？可以使用网格搜索!

网格搜索的意思其实就是遍历所有我们想要尝试的参数组合，看看哪个参数组合的性能最高，那么这组参数组合就是模型的最佳参数。

`sklearn` 为我们提供了网格搜索的接口，能很方便的进行网格搜索。

```
from sklearn.model_selection import GridSearchCV

# 想要调整的参数的字典，字典的key为参数名字，value为想要尝试参数值
param_grid = {'n_estimators': [10, 20, 50, 100, 150, 200],'max_depth': [5, 10, 15, 20, 25, 30]}

# 采用5折验证的方式进行网格搜索，分类器为随机森林
grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=5)
grid_search.fit(X_train, Y_train)

# 打印最佳参数组合
print(grid_search.best_params_)
# 打印最佳参数组合时模型的最佳性能
print(grid_search.best_score_)
```



可以看到经过调参之后，随机森林模型的性能提高到了 `0.8323` ，提升了接近 `1%` 的准确率。然后使用最佳参数构造随机森林，并对测试集测试会发现，测试集的准确率达到了 `0.8525` 。

```
Y_train = data['Survived']
X_train = data.drop(['Survived'], axis=1)

Y_test = test_data['Survived']
X_test = test_data.drop(['Survived'], axis=1)

clf = RandomForestClassifier(n_estimators=50, max_depth=5)
clf.fit(X_train, Y_train)
predict = clf.predict(X_test)
print(accuracy_score(Y_test, predict))
```

6.2 人脸性别识别

人脸性别识别可以根据图像中的特征识别出人脸所代表的性别信息。一般应用仔一些如安防，监控等场景。本案例只要介绍怎样使用 Opencv 和 sklearn 来实现人脸性别识别。

实训已在 `educoder` 平台上提供，若感兴趣可以输入链接进行体验。

链接：https://www.educoder.net/paths/130

6.2.1 OpenCV入门

什么是OpenCV

OpenCV 是计算机视觉领域应用最广泛的开源工具包，基于 C/C++ ，支持 Linux/Windows/MacOS/Android/iOS ，并提供了 Python，Matlab 和 Java 等语言的接口，因为其丰富的接口，优秀的性能和商业友好的使用许可，不管是学术界还是业界中都非常受欢迎。 OpenCV 最早源于 Intel 公司 1998 年的一个研究项目，当时在 Intel 从事计算机视觉的工程师盖瑞·布拉德斯基(Gary Bradski)访问一些大学和研究组时发现学生之间实现计算机视觉算法用的都是各自实验室里的内部代码或者库，这样新来实验室的学生就能基于前人写的基本函数快速上手进行研究。于是 OpenCV 旨在提供一个用于计算机视觉的科研和商业应用的高性能通用库。

第一个 alpha 版本的 OpenCV 于 2000 年的 CVPR 上发布，在接下来的5年里，又陆续发布了 5 个 beta 版本， 2006 年发布了第一个正式版。 2009 年随着盖瑞加入了 Willow Garage ， OpenCV 从 Willow Garage 得到了积极的支持，并发布了 1.1 版。 2010 年 OpenCV 发布了 2.0 版本，添加了非常完备的 C++ 接口，从 2.0 开始的版本非常用户非常庞大，至今仍在维护和更新。 2015 年 OpenCV 3 正式发布，除了架构的调整，还加入了更多算法，更多性能的优化和更加简洁的 API ，另外也加强了对 GPU 的支持，现在已经在许多研究机构和商业公司中应用开来。

安装OpenCV

想要安装 OpenCV 非常简单，只需在有网的环境下，在命令行中输入指令: pip install opencv-python 即可安装。

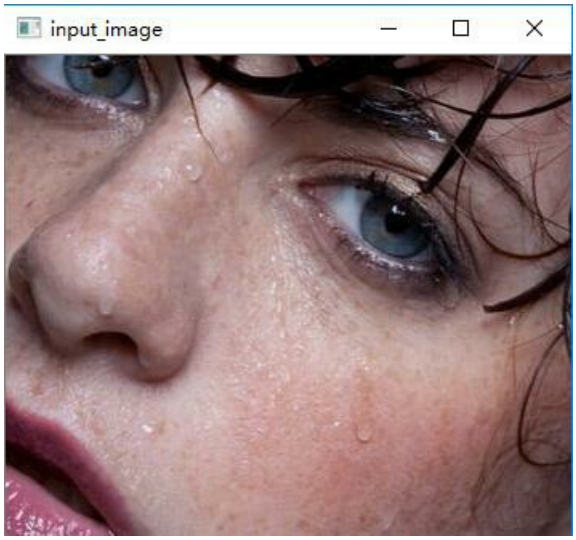
安装好后，使用如下代码，若能成功运行，则说明安装成功。

```
import cv2
print(cv2.__version__)
```

读取并显示图片

假设在当前目录有一张图片，图片名为 example.png ，那么可以使用 imread 函数读取图片，使用 namedWindow 函数创建显示图片的窗口，使用 imshow 函数显示图片。

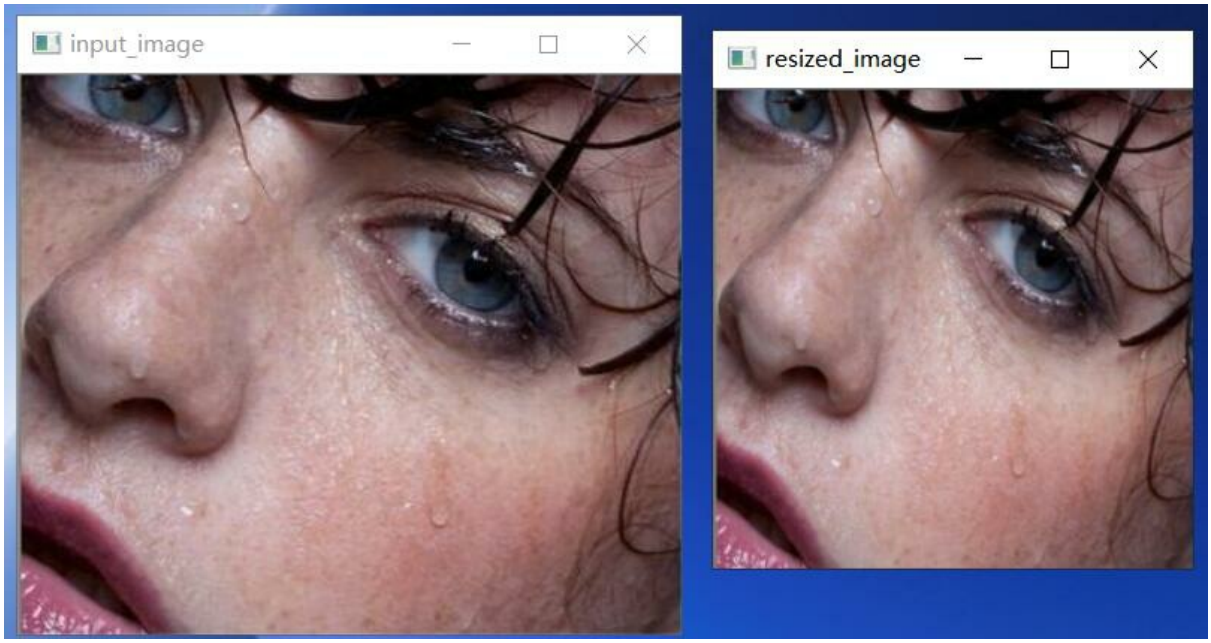
```
import cv2
# 读取图片
src=cv2.imread('./example.png')
# 创建一个标题为input_image的窗口
cv2.namedWindow('input_image', cv2.WINDOW_AUTOSIZE)
# 在标题为input_image的窗口上显示图片
cv2.imshow('input_image', src)
cv2.waitKey(0)
# 销毁窗口
cv2.destroyAllWindows()
```



图像缩放

有的时候，得到的原始图像数据集中的图像大小不一致，如果想要将图像缩放至固定的宽高，则可以使用 `resize` 函数实现功能。

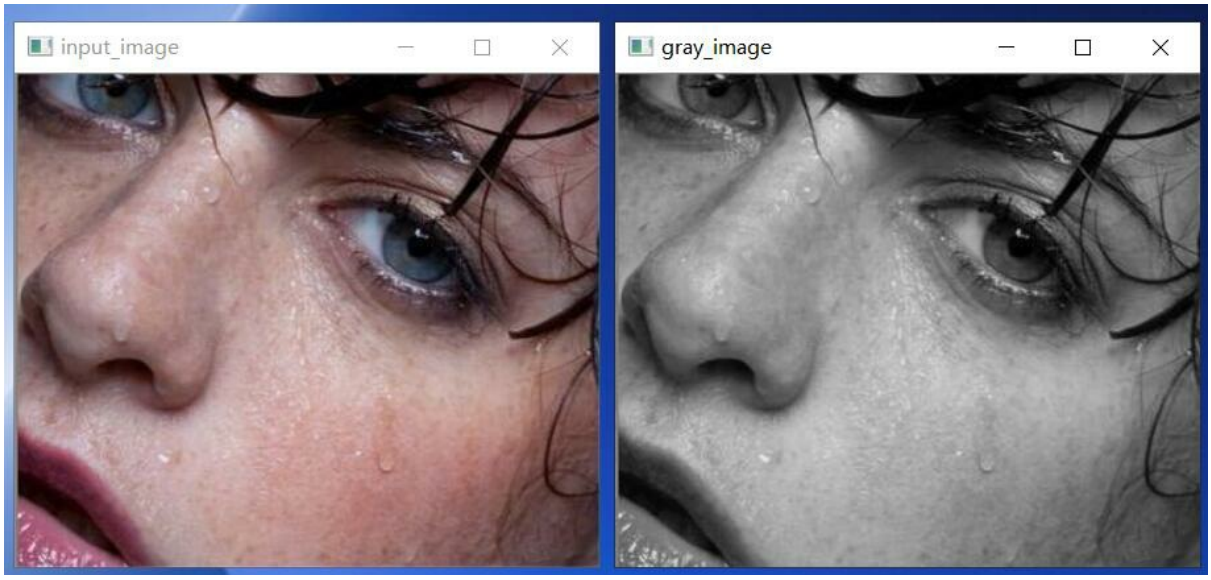
```
import cv2
# 读取图片
src=cv2.imread('./example.png')
# 将src图片缩放成宽为64个像素，高为64个像素的图像，并保存在dst变量中
dst = cv2.resize(src, (246, 256))
# 创建一个标题为input_image的窗口
cv2.namedWindow('input_image', cv2.WINDOW_AUTOSIZE)
# 创建一个标题为resized_image的窗口
cv2.namedWindow('resized_image', cv2.WINDOW_AUTOSIZE)
# 在标题为input_image的窗口上显示图片src
cv2.imshow('input_image', src)
# 在标题为resized_image的窗口上显示图片dst
cv2.imshow('resized_image', dst)
cv2.waitKey(0)
# 销毁窗口
cv2.destroyAllWindows()
```



彩色图转灰度图

有的时候颜色信息对于我们的任务来说并不是特别重要，此时可以使用 `cvtColor` 函数实现将彩色图转换成灰度图。

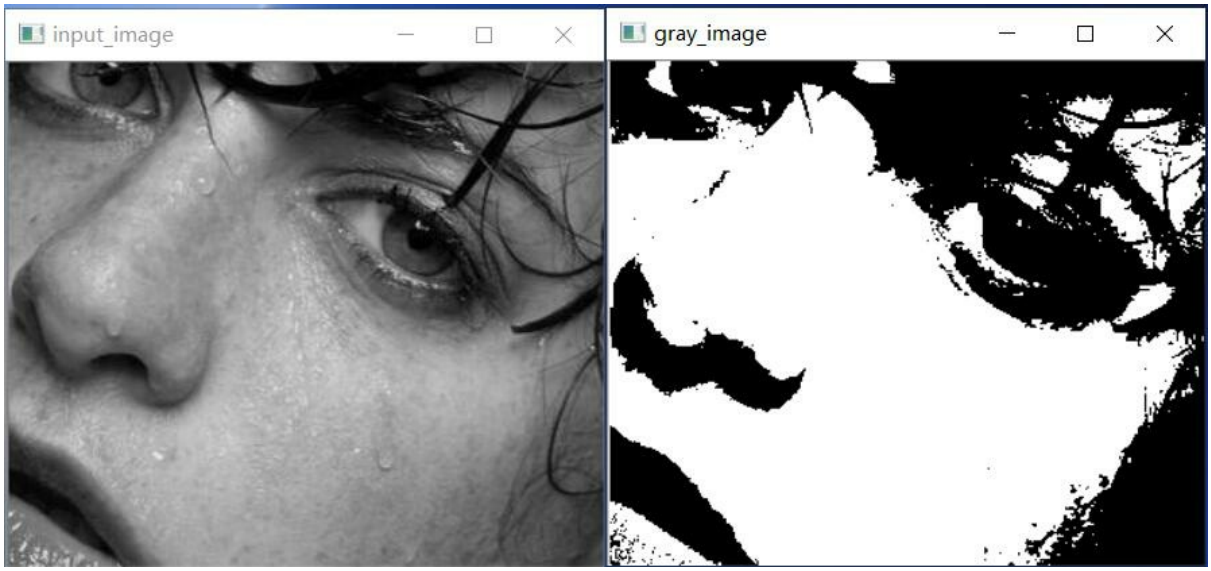
```
import cv2
# 读取图片
src=cv2.imread('./example.png')
# 将图片src从BGR的彩色图转换成灰度图，并保存到dst
dst = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
# 创建一个标题为input_image的窗口
cv2.namedWindow('input_image', cv2.WINDOW_AUTOSIZE)
# 创建一个标题为resized_image的窗口
cv2.namedWindow('gray_image', cv2.WINDOW_AUTOSIZE)
# 在标题为input_image的窗口上显示图片src
cv2.imshow('input_image', src)
# 在标题为resized_image的窗口上显示图片dst
cv2.imshow('gray_image', dst)
cv2.waitKey(0)
# 销毁窗口
cv2.destroyAllWindows()
```



二值化

图像二值化就是将图像上的像素点的灰度值设置为 `0` 或 `255` ，也就是将整个图像呈现出明显的黑白效果的过程。二值化的原理就是找到一个合适的阈值 `t` ，然后遍历整张图片中所有的像素点，当像素点的灰度值高于阈值 `t` 时，将该像素点的灰度值置为 `255` ，否则置成 `0` 。 `OpenCV` 中提供了实现二值化的函数 `threshold` 。

```
import cv2
# 读取图片，0表示读取到的图片为灰度图
src=cv2.imread('./example.png', 0)
# 对图片scr进行二值化，cv2.THRESH_OTSU表示自动找出合适的阈值，并将二值化后的图保存到dst
_, dst = cv2.threshold(src, 127, 255, cv2.THRESH_OTSU)
# 创建一个标题为input_image的窗口
cv2.namedWindow('input_image', cv2.WINDOW_AUTOSIZE)
# 创建一个标题为resized_image的窗口
cv2.namedWindow('gray_image', cv2.WINDOW_AUTOSIZE)
# 在标题为input_image的窗口上显示图片src
cv2.imshow('input_image', src)
# 在标题为resized_image的窗口上显示图片dst
cv2.imshow('gray_image', dst)
cv2.waitKey(0)
# 销毁窗口
cv2.destroyAllWindows()
```



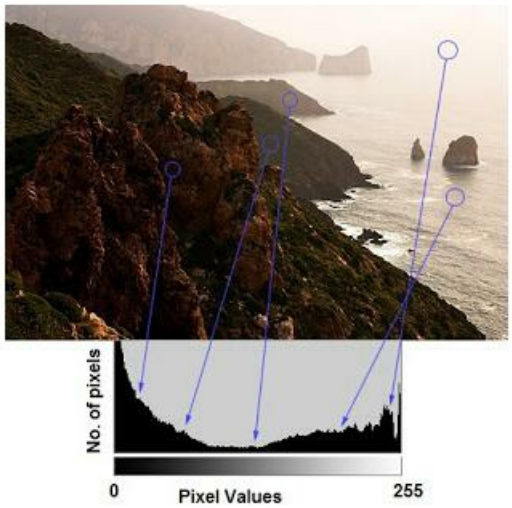
直方图均衡

对曝光过度或者逆光拍摄的图片可以通过直方图均衡化的方法用来增强局部或者整体的对比度。

具体思路是通过找出图像中最亮和最暗的像素值将之映射到纯黑和纯白之后再将其他的像素值按某种算法映射到纯黑和纯白之间的值。另一种方法是寻找图像中像素的平均值作为中间灰度值，然后扩展范围以达到尽量充满可显示的值。

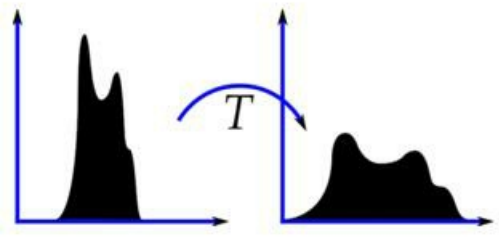
什么是直方图?你可以把直方图看作一个图或图，它给你一个关于图像的强度分布的总体思路。它是一个带有像素值的图(从 `0` 到 `255`，不总是)在 `x` 轴上，在 `y` 轴上的图像对应的像素个数。

这只是理解图像的另一种方式。通过观察图像的直方图，你可以直观地了解图像的对比度、亮度、亮度分布等。今天几乎所有的图像处理工具都提供了直方图上的特征。



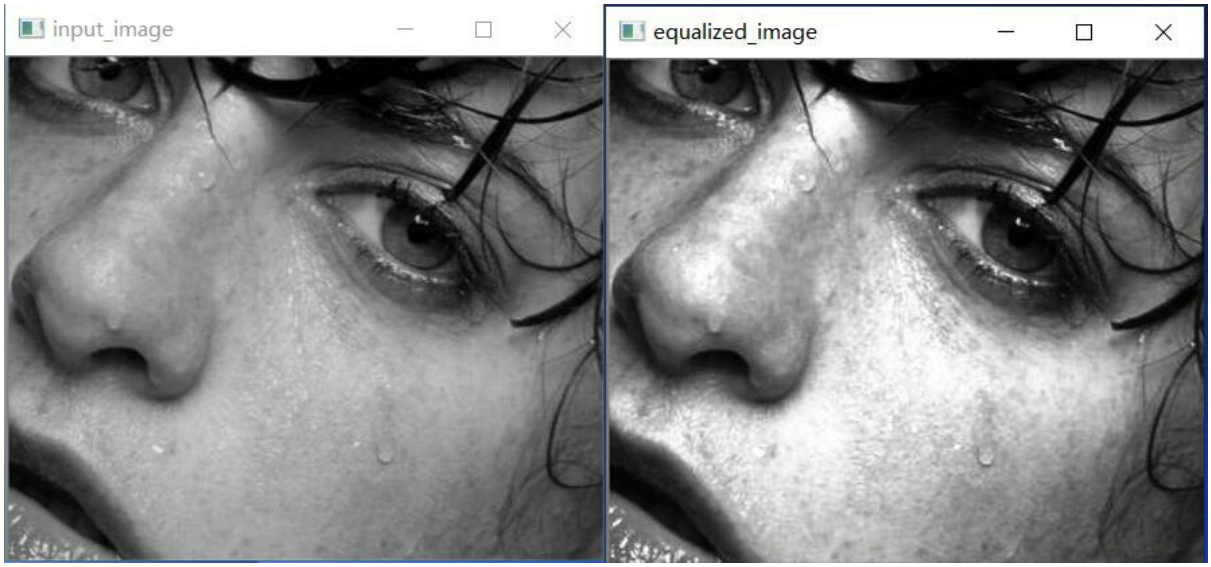
你可以看到图像和它的直方图。（这个直方图是用灰度图像绘制的，而不是彩色图像）。直方图的左边部分显示了图像中较暗像素的数量，右边区域显示了更明亮的像素。从直方图中可以看到，深色区域的像素数量比亮色区域更多，而中间色调的数量(中值大约在 `127` 左右)则少得多。

考虑一个图像，其像素值仅限制在特定的值范围内。例如，更明亮的图像将使所有像素都限制在高值中。但是一个好的图像会有来自图像的所有区域的像素。所以你需要把这个直方图拉伸到两端(如下图所给出的)，这就是直方图均衡的作用(用简单的话说)。这通常会改善图像的对比度。



`OpenCV` 中提供了实现直方图均衡的函数 `equalizeHist` 。

```
import cv2
# 读取图片，0表示读取到的图片为灰度图
src=cv2.imread('./example.png', 0)
# 对图片scr进行直方图均衡，并将结果保存到dst
dst = cv2.equalizeHist(src)
# 创建一个标题为input_image的窗口
cv2.namedWindow('input_image', cv2.WINDOW_AUTOSIZE)
# 创建一个标题为resized_image的窗口
cv2.namedWindow('equalized_image', cv2.WINDOW_AUTOSIZE)
# 在标题为input_image的窗口上显示图片src
cv2.imshow('input_image', src)
# 在标题为resized_image的窗口上显示图片dst
cv2.imshow('equalized_image', dst)
cv2.waitKey(0)
# 销毁窗口
cv2.destroyAllWindows()
```



6.2.2 人脸位置检测

OpenCV 的人脸检测，使用 Harr分类器 。该分类器采用的 Viola-Jones人脸检测算法 。它是在 2001 年由 Viola 和 Jones 提出的基于机器学习的人脸检测算法。

算法首先需要大量的积极图片（包含人脸的图片）和消极图片（不包含人脸的图片）。然后从中提取类 Harr特征(Harr-like features) ，之所以称为 Harr 分类器，是正是因为它使用了类 Harr 特征。最后，训练出一个级联检测器，用其来检测人脸。

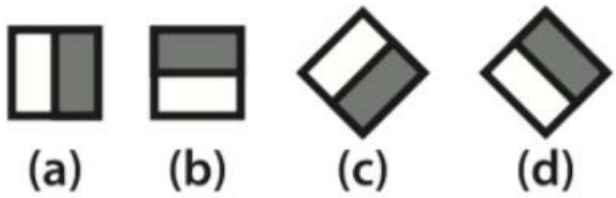
类Harr特征

图像中的特征通常是指，图片的像素点经过一系列的运算之后得到的结果，这些结果可能是向量、矩阵和多维数据等等。类 Harr 特征是一种反映图像的灰度变化的，像素分模块求差值的一种特征。

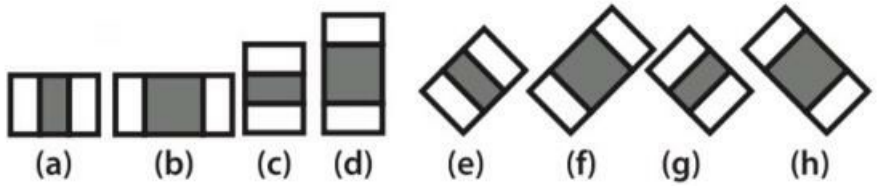
Harr特征类别

它分为三类：边缘特征、线性特征、中心特征和对角线特征。用黑色两种矩形框组成为特征模板。

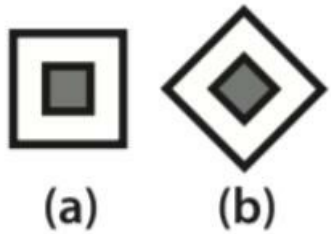
1 .边缘特征



2 .线性特征



3 .中心特征和对角线特征



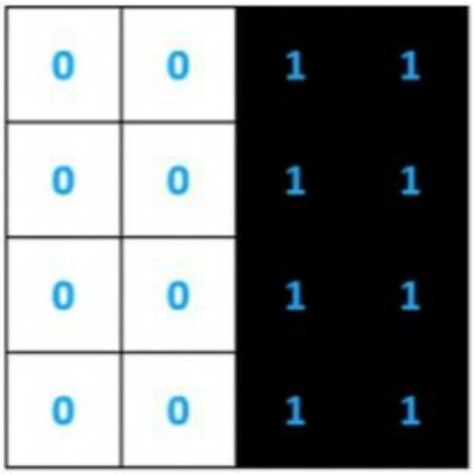
特征值计算

特征模板的特征值计算的方式，是用黑色矩形像素总和的均值减去白色矩形像素总和的均值。

$\Delta = \frac{1}{n}\sum_{dark}^{n} I(x) - \frac{1}{n}\sum_{white}^{n} I(x)$

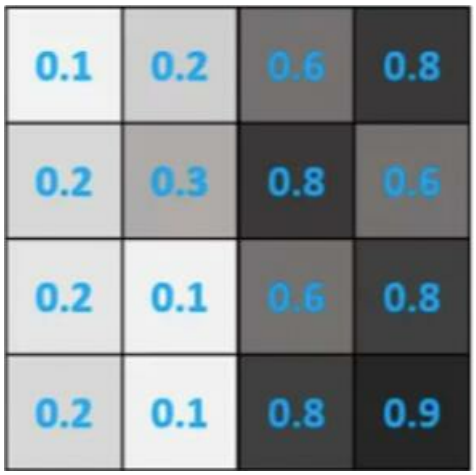
例如，对于 `4x4` 的像素块。

理想情况下，黑色和白色的像素块分布如下：



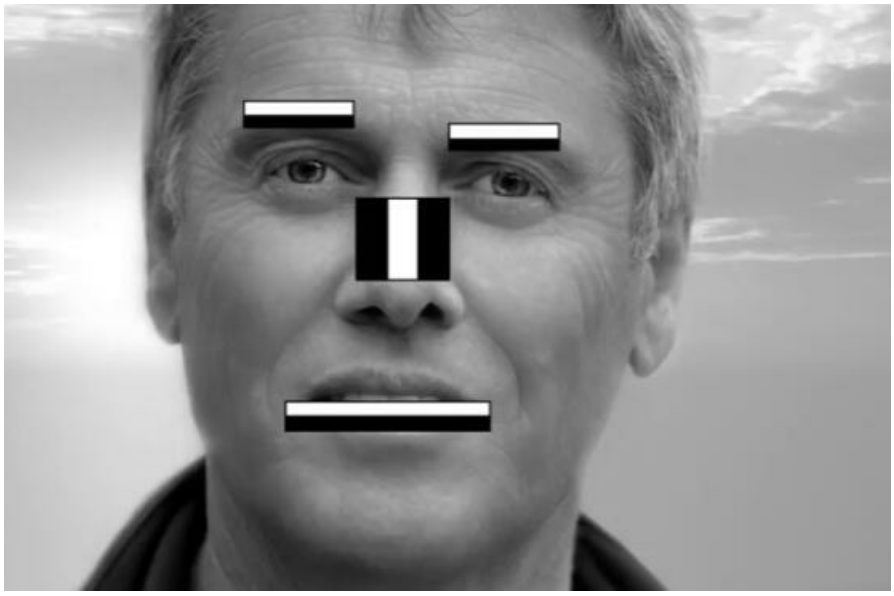
符合边缘特征的情况(`a`)。

但是通常情况，一张灰阶照片的黑白分布并非如此的明显，例如：



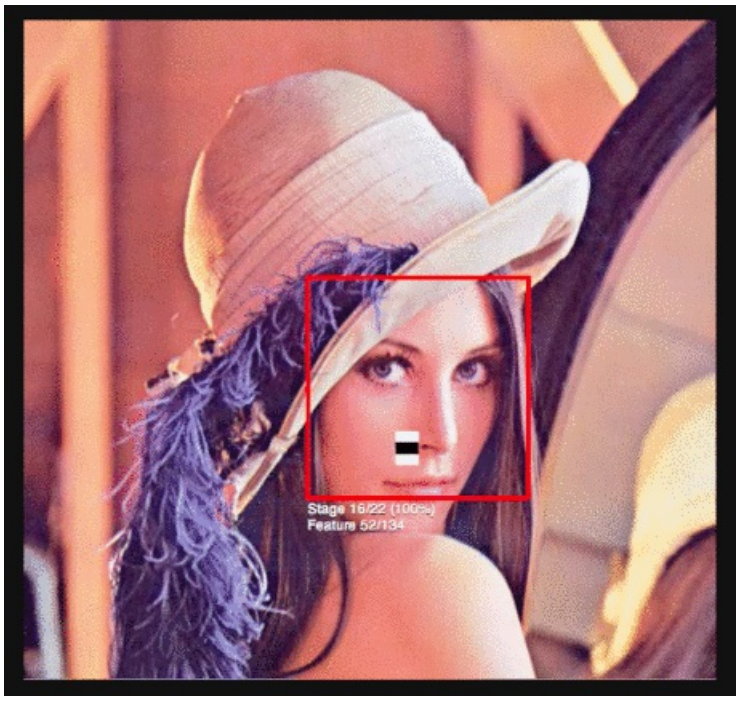
根据公式，第一张图特征值为 `1` ，第二张图特征值为 `0.75-0.18=0.56` 。

一张图中，对于识别人脸，只有部分特征是有效的。例如，用下图中的特征模板可以看出，眉毛区域比额头要亮，鼻梁区域比眼镜区域要亮。嘴唇区域比牙齿区域要暗。这样的类 `Harr` 特征能很好的识别出人脸。



为简化特征值计算，可以使用积分图算法。得到类 `Harr` 特征后，使用 `AdaBoost` 的方法选择出有效特征。最后再使用瀑布型级联检测器提高检测速度。其中，瀑布的每一层都是一个由 `Adaboost算法` 训练得到的强分类器。

`Harr` 人脸检测一个简单的动画过程如下：



红色的搜索框不断移动，检测出是否包含人脸。一般来说，输入的图片会大于样本，为了检索出不同大小的目标，分类器可以按比例的改变自己的尺寸，对输入图片进行多次的扫描。

训练**Harr**分类器

训练 `Harr` 分类器的主要步骤如下：

- 搜集制作大量的“消极”图像

- 搜集制作大量“积极”图像，确保这些图像中包含要检测的对象
- 创建“积极”向量文件
- 使用 `OpenCV` 训练 `Harr` 分类器

因为训练需要花费较多的资源和时间，所以我们学习时，先使用 `OpenCV` 中已经训练好的 `Harr` 分类器。

使用**Harr**分类器检测人脸

声明分类器：

```
CascadeClassifier(模型文件路径)
```

调用分类函数：

```
detectMultiScale(图片对象,scaleFactor, minNeighbors, minSize)
```

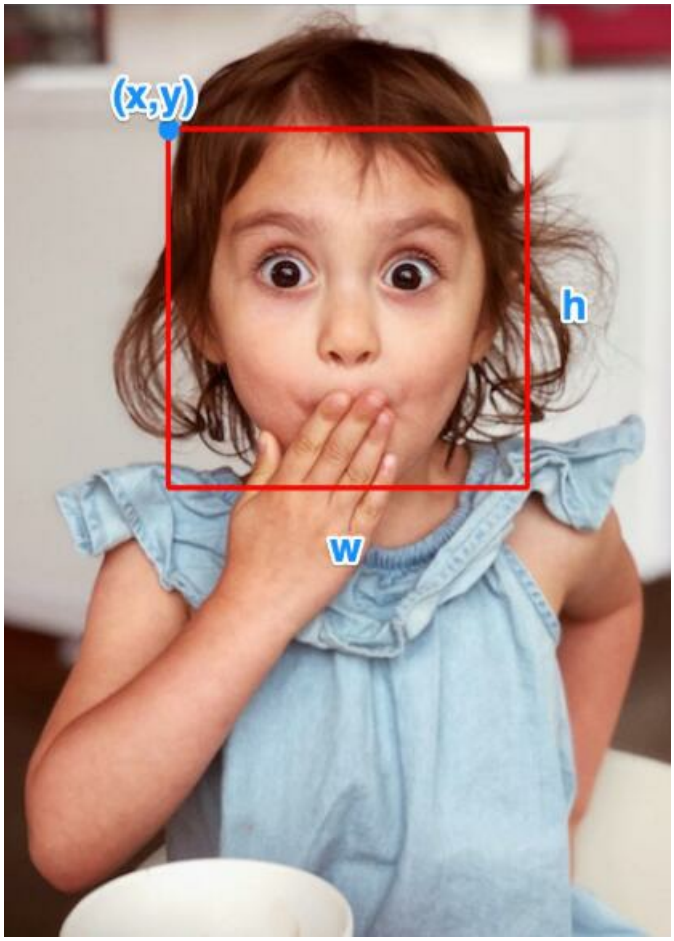
参数说明：

1. 图片对象：待识别图片对象；
2. `scaleFactor`：图像缩放比例；
3. `minNeighbors`：对特征检测点周边多少有效点同时检测，这样可避免因选取的特征检测点太小而导致遗漏；
4. `minSize`：特征检测点的最小尺寸，可选参数。

示例如下：

```
import numpy as np
import cv2 as cv
# 读取图片
img = cv.imread('face.jpg')
# 转换为灰度图片
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
# 人脸检测器
face_cascade = cv.CascadeClassifier('Harrcascade_frontalface_default.xml')

# 识别人脸
faces = face_cascade.detectMultiScale(gray, 1.3, 5)

for (x,y,w,h) in faces:
    print(x,y,w,h)
```

其中，图片坐标系以左上角为原点，`x,y` 代表人脸区域左上角坐标，`w` 代表宽度，`h` 代表高度。



如果想动手实现人脸检测，可以尝试进入链接进行实战：https://www.educoder.net/shixuns/hl7wacq5/challenges

6.2.3 人脸性别识别

截取人脸ROI

在进行人脸性别识别时，不需要知道除了人脸之外的信息，所以需要根据上一节中检测到的人脸位置将人脸的 `ROI` (感兴趣区域)截取下来。

假设检测到的人脸框为 `face_rect` ，那么可以根据人脸框得到人脸图像。

```
x, y, w, h = face_rect
f_h, f_w = gray.shape[:2]
# 防止溢出
x = max(0, x)
y = max(0, y)
w = min(f_w - x, w)
h = min(f_h - y, h)
# 截取图像
face = frame[y: y + h, x: x + w, :]
# 保存图像
cv2.imwrite('face.jpg', face)
```

原图：

提取到的人脸ROI：

提取**HoG**特征

方向梯度直方图（`Histogram of Oriented Gradient, HOG`）是一种在计算机视觉和图像处理中用来进行物体检测的特征描述方法。所谓特征描述方法，就是通过从图像中提取有用信息和丢弃无关信息来简化图像，并将信息运用于算法。

特征描述方法从图像中提取有用信息，那么在`HOG`中提取了图像那些有用信息呢？

其实，在图像中，物体的边缘和角落包含了关于物体形状的更多信息。也就是我们平时能看到的物体轮廓。如果能够提取出图像边缘和角落的信息，将非常有利于物体检测。

而对于图像，边缘和角落周围的梯度这个特征，变化幅度很大，所以在`HOG`特征描述方法中，使用梯度方向的分布（直方图）作为它的特征。

`OpenCV`中提供了提取`HOG`特征的接口。首先，使用`cv2.HOGDescriptor()`函数声明HOG特征描述方法，代码如下：

```
hog = cv2.HOGDescriptor()
```

然后计算加载的图像的`HOG`特征值，代码如下：

```
h = hog.compute(img)
```

完整的代码示例如下：

```
import cv2
hog = cv2.HOGDescriptor()
img = cv2.imread(sample)
feature = hog.compute(img)
```

训练性别分类器

提取了人脸ROI的 `HOG` 特征后，就可以使用 `sklearn` 训练性别分类器了。

```
from sklearn.svm import SVC

clf = SVC()
# hog_features为训练集中人脸ROI的HOG特征
clf.fit(hog_features, label)
pred = clf.predict(test_features)
```