

NumPy的使用

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。

NumPy 是一个运行速度非常快的数学库，主要用于数组计算，包含：

- 一个强大的N维数组对象 ndarray
- 广播功能函数
- 整合 C/C++/Fortran 代码的工具
- 线性代数、傅里叶变换、随机数生成等功能

2、为什么要用Numpy

NumPy是Python中的一个运算速度非常快的一个数学库，它非常重视数组。它允许你在Python中进行向量和矩阵计算，并且由于许多底层函数实际上是用C编写的，因此你可以体验在原生Python中永远无法体验到的速度。

```
import numpy as np
import time

# 用python自带方法处理
def func(values):
    result = []
    for v in values:
        result.append(v * v)
    return result
data = range(10000)
%timeit func(data)
运行结果:
1.07 ms ± 20.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

# 用numpy中的方法处理
arr = np.arange(0,10000)
%timeit arr ** arr
运行结果:
397 µs ± 32.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

NumPy绝对是科学Python成功的关键之一，如果你想要进入Python中的数据科学或机器学习，你就要必须学习它。从最后的执行结果来看numpy的处理速度要比python的处理速度快上十几倍，当然这只是它的其中一项优势，下面就通过一些具体的操作来看一看numpy的用法与优势。

3、怎么用Numpy

安装方法：

```
pip install numpy
```

引用方式：

```
import numpy as np
```

这是官方认证的导入方式，可能会有人说为什么不用 `from numpy import *`，是因为在numpy当中有一些方法与Python中自带的一些方法，例如 `max`、`min` 等冲突，为了避免这些麻烦大家就约定俗成的都使用这种方法。

Numpy的核心特征就是N-维数组对——`ndarray`。

3.1、为什么要用ndarray?

numpy所有的操作都是围绕着数组展开的，这个数组的名字就叫做 `ndarray`，在学习`ndarray`数组之前肯定有人会说这个东西和Python中的列表差不多啊，为什么不用列表呢，列表还要方便些。其实列表`list`本身是为了处理更广泛、更通用的目的而构建的，其实从这一方面来看`ndarray`对于处理这个数组类型结构的数据会更加方便。

接下来我们可以通过具体的实例来展示一下`ndarray`的优势。

现在有这么一个需求：

已知若干家跨国公司的市值（美元），将其换算为人民币

按照Python当中的方法

第一种：是将所有的美元通过for循环依次迭代出来，然后用每个公司的市值乘以汇率

第二种：通过`map`方法和`lambda`函数映射

这些方法相对来说也挺好用的，但是再来看通过`ndarray`对象是如何计算的

[

```
In [15]: import numpy as np
import random
b = [random.uniform(100,200) for _ in range(10000)] # 随机生成一万个小数
c = 6.9 # 汇率
```

```
In [14]: b = np.array(b) # 生成多维数组对象
b
```

```
Out[14]: array([113.58735517, 168.58120313, 123.84486298, ..., 133.07183826,
184.61710718, 170.9773161 ])
```

```
In [13]: b * c
```

```
Out[13]: array([ 783.75275067, 1163.21030159, 854.52955457, ..., 918.19568398,
1273.85803956, 1179.7434811 ])
```

]

通过`ndarray`这个多维数组对象可以让这些批量计算变得更加简单，当然这只它其中一种优势，接下来就通过具体的操作来发现。

3.2、ndarray-创建

方法	描述
<code>array()</code>	将列表转换为数组，可选择显式指定 <code>dtype</code>
<code>arange()</code>	<code>range</code> 的numpy版，支持浮点数
<code>linspace()</code>	类似 <code>arange()</code> ，第三个参数为数组长度
<code>zeros()</code>	根据指定形状和 <code>dtype</code> 创建全0数组
<code>ones()</code>	根据指定形状和 <code>dtype</code> 创建全1数组
<code>empty()</code>	根据指定形状和 <code>dtype</code> 创建空数组（随机值）
<code>eye()</code>	根据指定边长和 <code>dtype</code> 创建单位矩阵

[

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

data

1
2
3

data

1
2
3

.max() = 3

]

1、arange():

```
np.arange(1.2,10,0.4)
```

执行结果:

```
array([1.2, 1.6, 2. , 2.4, 2.8, 3.2, 3.6, 4. , 4.4, 4.8, 5.2, 5.6, 6. ,
       6.4, 6.8, 7.2, 7.6, 8. , 8.4, 8.8, 9.2, 9.6])
```

在进行数据分析的时候通常我们遇到小数的机会远远大于遇到整数的机会，这个方法与Python内置的range的使用方法一样

2、linspace()

```
np.linspace(1,10,20)
```

执行结果:

```
array([ 1.          ,  1.47368421,  1.94736842,  2.42105263,  2.89473684,
        3.36842105,  3.84210526,  4.31578947,  4.78947368,  5.26315789,
        5.73684211,  6.21052632,  6.68421053,  7.15789474,  7.63157895,
        8.10526316,  8.57894737,  9.05263158,  9.52631579, 10.          ])
```

这个方法与arange有一些区别，arange是顾头不顾尾，而这个方法是顾头又顾尾，在1到10之间生成的二十个数每个数字之间的距离相等的，前后两个数做减法肯定相等

3、zeros()

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

执行结果:

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

会用0生成三行四列的一个多维数组

4、ones()

```
np.ones((3,4))
```

执行结果:

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

会用1生成三行四列的一个多维数组

5、empty()

```
np.empty(10)
```

执行结果:

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

这个方法只申请内存，不给它赋值

6、eye()

```
np.eye(5)
```

执行结果：

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
# 对角矩阵
```

3.3、ndarray是一个多维数组列表

接下来就多维数组举个例子：

[

```
In [21]: li = [1, 2, 3]
         np.array(li)

Out[21]: array([1, 2, 3])
```

```
In [18]: li1 = [
         [1, 2, 3],
         [4, 5, 6]
         ]
         np.array(li1)

Out[18]: array([[1, 2, 3],
                [4, 5, 6]])
```

```
In [19]: li2 = [[[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]]]
         np.array(li2)

Out[19]: array([[[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]]])
```

]

为了创建一个2维数组，我们是传递了一个列表的列表给这个array()函数。如果我们想要的是一个三维数组，我们就必须要一个列表的列表的列表（也就是三层列表），以此类推。

[

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



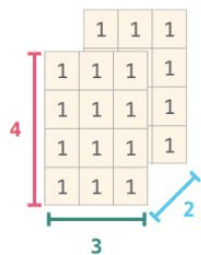
	5	6
1	2	8
3	4	

]

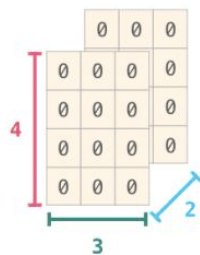
很多情况下，处理一个新的维度只需要在numpy函数的参数中添加一个逗号

[

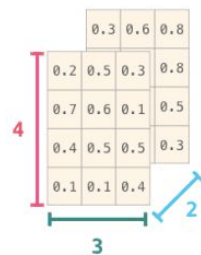
`np.ones((4,3,2))`



`np.zeros((4,3,2))`



`np.random.random((4,3,2))`



]

有的人可能会说了，这个数组跟Python中的列表很像啊，它和列表有什么区别呢？

```
'''
```

在python中列表是可以存任意类型的值的，但是在数组当中的元素必须类型必须相同。这是因为列表中存的只是每个元素的地址，不管运行多少次，值的位置是不会改变的，不需要在意数据的类型；而在ndarray当中存的是具体的值，每一次执行都是重新存放。

```
'''
```

```
l1 = ['1','2',4]
na = np.array(l1)
print(f"ndarray:{id(na[0])}")
print(f"list:{id(l1[0])}")
```

```
> ndarray:2140960887632
    list:2140897577592
```

```
'''
```

通过多次执行其实就可以发现，ndarray数组的id值一直在不停的换，而list的id值始终保持不变

```
'''
```

- 数组对象内的元素类型必须相同
- 数组大小不可修改

3.4、常用属性

属性	描述
T	数组的转置（对高维数组而言）
dtype	数组元素的数据类型
size	数组元素的个数
ndim	数组的维数
shape	数组的维度大小（以元组形式）
itemsize	每个项占用的字节数
nbytes	数组中的所有数据消耗掉的字节数

T:转置 转置是一种特殊的数据重组形式，可以返回底层数据的视图而不需要复制任何内容。
通俗点说，转置就是将数据旋转**90度**，行变成列，列变成行。

```
l11 = [  
    [1,2,3],  
    [4,5,6]  
]  
a = np.array(l11)  
a.T  
执行结果:  
array([[1, 4],  
       [2, 5],  
       [3, 6]])
```

[

data

1	2
3	4
5	6

data.T

1	3	5
2	4	6

]

```
# dtype: 返回当前数据的数据类型  
arr = np.arange(10)  
arr.dtype  
执行结果:  
dtype('int32')
```

```
# size: 返回当前数组内存在的元素个数  
l1 = [[1,2,3],  
      [4,5,6]],  
      [[7,8,9],  
      [1,5,9]]  
arr1 = np.array(l1)  
arr1.size  
执行结果:  
12
```

```
# ndim: 返回当前数组维度  
l1 = [[1,2,3],  
      [4,5,6]],  
      [[7,8,9],  
      [1,5,9]]  
arr1 = np.array(l1)  
arr1.ndim  
执行结果:
```

3

shape: 返回数组维度大小

```
l1 = [[1,2,3,4],
      [4,5,6,5],
      [6,8,3,6]],
      [[7,8,9,7],
      [1,5,9,7],
      [4,6,8,4]
      ]]
```

```
arr1 = np.array(l1)
```

```
arr1.shape
```

执行结果:

```
(2, 3, 4)
```

"""

最终三个参数代表的含义依次为: 二维维度, 三维维度, 每个数组内数据大小

要注意这些数组必须要是相同大小才可以

"""

3.5、数据类型

- dtype

类型	描述
布尔型	bool_
整型	int_ int8 int16 int32 int 64
无符号整型	uint8 uint16 uint32 uint64
浮点型	float_ float16 float32 float64

整型:

int32只能表示 $(-2^{31}, 2^{31}-1)$,因为它只有32个位, 只能表示 2^{32} 个数

无符号整型:

只能用来存正数, 不能用来存负数

"""

补充:

astype()方法可以修改数组的数据类型

示例:

```
data.astype(np.float)
```

"""

3.6、向量化数学运算

- 数组和标量(数字)之间运算

```
li1 = [
    [1,2,3],
    [4,5,6]
]
a = np.array(li1)
a * 2
```

运行结果:

```
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} * 1.6 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} * \begin{bmatrix} 1.6 \\ 1.6 \end{bmatrix} = \begin{bmatrix} 1.6 \\ 3.2 \end{bmatrix}$$

与标量之间进行向量化运算，多维数组与一维数组没有任何区别，都会将你要运算的数字映射到数组中的每一个元素上进行运算

- 同样大小数组之间的运算

```
# 12数组
12 = [
    [1,2,3],
    [4,5,6]
]
a = np.array(12)

# 13数组
13 = [
    [7,8,9],
    [10,11,12]
]
b = np.array(13)

a + b # 计算
```

执行结果:

```
array([[ 8, 10, 12],
       [14, 16, 18]])
```

$$\begin{array}{|c|} \hline \text{data} \\ \hline \end{array}
 \begin{array}{|c|} \hline 1 \\ \hline \end{array}
 -
 \begin{array}{|c|} \hline \text{ones} \\ \hline \end{array}
 \begin{array}{|c|} \hline 1 \\ \hline \end{array}
 =
 \begin{array}{|c|} \hline 0 \\ \hline \end{array}
 \begin{array}{|c|} \hline 1 \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline \text{data} \\ \hline \end{array}
 \begin{array}{|c|} \hline 1 \\ \hline \end{array}
 *
 \begin{array}{|c|} \hline \text{data} \\ \hline \end{array}
 \begin{array}{|c|} \hline 1 \\ \hline \end{array}
 =
 \begin{array}{|c|} \hline 1 \\ \hline \end{array}
 \begin{array}{|c|} \hline 4 \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline \text{data} \\ \hline \end{array}
 \begin{array}{|c|} \hline 1 \\ \hline \end{array}
 /
 \begin{array}{|c|} \hline \text{data} \\ \hline \end{array}
 \begin{array}{|c|} \hline 1 \\ \hline \end{array}
 =
 \begin{array}{|c|} \hline 1 \\ \hline \end{array}
 \begin{array}{|c|} \hline 1 \\ \hline \end{array}$$

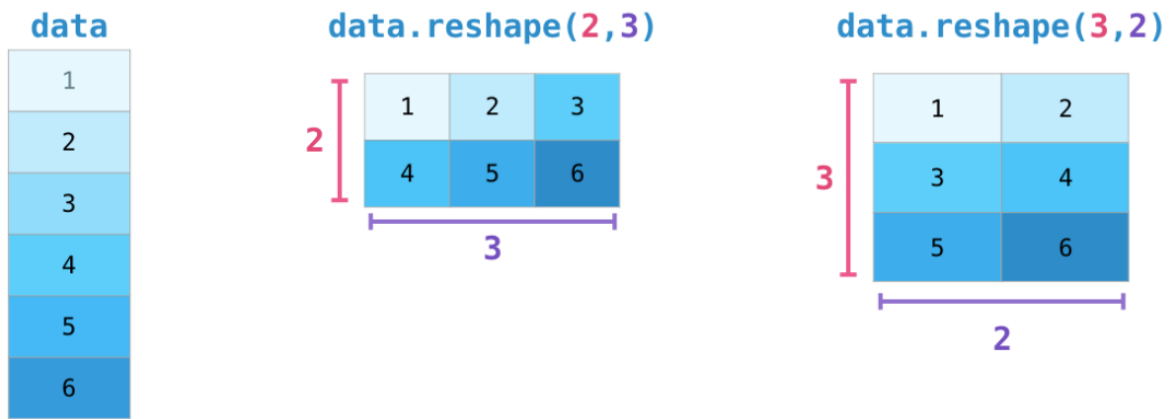
数组与数组之间的向量化运算，两个运算的数组必须相同大小，否则会报错

3.7、索引和切片

- 索引

一维索引使用与python本身的列表没有任何区别，所以接下来主要针对大的是多维数组

```
# np重塑
arr = np.arange(30).reshape(5,6) # 后面的参数6可以改为-1，相当于占位符，系统可以自动帮忙
算几列
```

```
# 将二维变一维
arr.reshape(30)
```

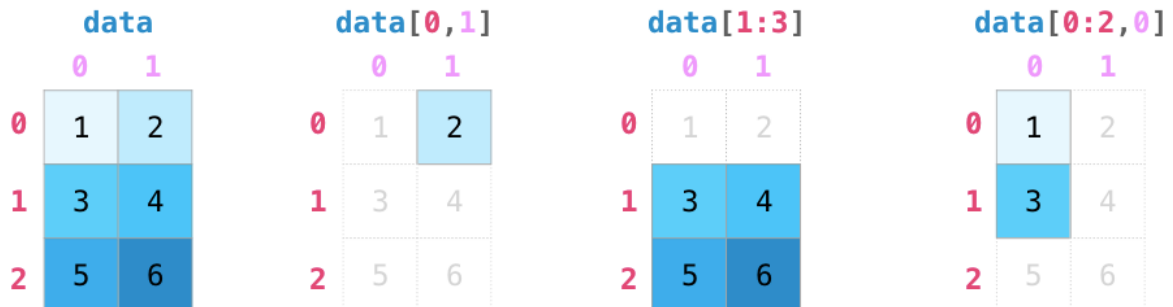
索引使用方法

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29]])
```

现在有这样一组数据，需求：找到20

列表写法：arr[3][2]

数组写法：arr[3,2] # 中间通过逗号隔开就可以了



• 切片

arr数组

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29]])
```

```
arr[0,1:4] # >>array([1, 2, 3])
```

```
arr[1:4,0] # >>array([ 6, 12, 18])
```

```
arr[:,2::2] # >>array([[ 0,  2,  4],
                      [12, 14, 16],
                      [24, 26, 28]])
```

```
arr[:,1] # >>array([ 1,  7, 13, 19, 25])
```

切片不会拷贝，直接使用的原视图，如果硬要拷贝，需要在后面加.copy()方法

```
In [25]: arr = np.arange(10) # 随机生成一个一维数组
arr
```

```
Out[25]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [26]: b = arr[5:] # 切片 5以后的
b
```

```
Out[26]: array([5, 6, 7, 8, 9])
```

```
In [27]: b[0] = 10 # 修改切片之后的数据
arr
```

```
Out[27]: array([ 0,  1,  2,  3,  4, 10,  6,  7,  8,  9])
```

最后会发现修改切片后的数据影响的依然是原数据。有的人可能对一点机制有一些不理解的地方，像Python中内置的都有赋值的机制，而Numpy却没有，其实是因为NumPy的设计目的是处理大数据，所以你可以想象一下，假如NumPy坚持要将数据复制来复制去的话会产生何等的性能和内存问题。

- 布尔型索引

现在有这样需求：给一个数组，选出数组中所有大于5的数。

```
li = [random.randint(1,10) for _ in range(30)]
a = np.array(li)
a[a>5]
执行结果：
array([10,  7,  7,  9,  7,  9, 10,  9,  6,  8,  7,  6])
```

原理：

a>5会对a中的每一个元素进行判断，返回一个布尔数组

a > 5的运行结果：

```
array([False,  True, False,  True,  True, False,  True, False, False,
        False, False, False, False, False, False,  True, False,  True,
        False, False,  True,  True,  True,  True,  True, False, False,
        False, False,  True])
```

布尔型索引：将同样大小的布尔数组传进索引，会返回一个有True对应位置的元素的数组

布尔型索引是numpy当中的一个非常常用的用法，通过布尔型索引取值方便又快捷。

4、通用函数

能对数组中所有元素同时进行运算的函数就是通用函数

4.1、常见通用函数

能够接受一个数组的叫做一元函数，接受两个数组的叫二元函数，结果返回的也是一个数组

- 一元函数：

函数	功能
abs、fabs	分别是计算整数和浮点数的绝对值
sqrt	计算各元素的平方根
square	计算各元素的平方
exp	计算各元素的指数 e^{**x}
log	计算自然对数
sign	计算各元素的正负号
ceil	计算各元素的ceiling值
floor	计算各元素floor值，即小于等于该值的最大整数
rint	计算各元素的值四舍五入到最接近的整数，保留dtype
modf	将数组的小数部分和整数部分以两个独立数组的形式返回，与Python的divmod方法类似
isnan	判断数组中的缺失值
isinf	表示那些元素是无穷的布尔型数组
cos, sin, tan	普通型和双曲型三角函数

以下简单演示常用的几个一元函数：

```
# 计算整数绝对值
arr = np.random.randint(-10,10,20)
np.abs(arr)
> array([9, 7, 3, 1, 5, 8, 7, 9, 4, 2, 7, 3, 4, 6, 6, 9, 2, 5, 8, 1])
-----

# 计算浮点数绝对值
arr = np.random.randn(2,5)
np.fabs(arr)
> array([[0.09892302, 0.06200835, 1.0324653 , 1.58089607, 0.44506652],
        [0.34897694, 1.04843539, 0.83976969, 0.4731551 , 0.92229931]])
-----

# 计算各元素的平方根
arr = np.arange(10)
np.sqrt(arr)
> array([0.          , 1.          , 1.41421356, 1.73205081, 2.          ,
        2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.          ])
```

- 二元函数：

函数	功能
add	将数组中对应的元素相加
subtract	从第一个数组中减去第二个数组中的元素
multiply	数组元素相乘
divide、floor_divide	除法或向下圆整除法（舍弃余数）
power	对第一个数组中的元素A，根据第二个数组中的相应元素B计算A**B
maximum, fmax	计算最大值，fmax忽略NAN
miximum, fmix	计算最小值，fmin忽略NAN
mod	元素的求模计算（除法的余数）

```

arr = np.random.randint(0,10,5)
arr1 = np.random.randint(0,10,5)
arr,arr1
> (array([0, 1, 8, 2, 6]), array([5, 4, 1, 7, 0]))
-----
# add 将数组中对应的元素相加
np.add(arr,arr1)
> array([5, 5, 9, 9, 6])
-----
# subtract 从第一个数组中减去第二个数组中的元素
np.subtract(arr,arr1)
> array([-5, -3, 7, -5, 6])
-----
# multiply 数组元素相乘
np.multiply(arr,arr1)
> array([ 0, 4, 8, 14, 0])
-----
...

```

补充内容：浮点数特殊值

浮点数：float

浮点数有两个特殊值：

- 1、nan(Not a Number):不等于任何浮点数（nan != nan）

- 2、inf(infinity):比任何浮点数都大

- Numpy中创建特殊值：np.nan、np.inf
- 数据分析中，nan常被用作表示数据缺失值

以上函数使用非常方便，使用这些方法可以让数据分析的操作更加便捷。

4.2、数学统计方法

函数	功能
sum	求和
cumsum	求前缀和
mean	求平均数
std	求标准差
var	求方差
min	求最小值
max	求最大值
argmin	求最小值索引
argmax	求最大值索引

```
arr = np.random.randint(0,10,10)
arr
> array([2, 9, 6, 5, 4, 2, 9, 8, 0, 5])
-----
# sum 求和
np.sum(arr)
> 50
-----
# cumsum 求前缀和
np.cumsum(arr)
array([ 2, 11, 17, 22, 26, 28, 37, 45, 45, 50], dtype=int32) # 依次累加
-----
# mean 求平均数
np.mean(arr)
> 5.0
-----
# 由于此档内容太过简单，后续就不一一展示了
...
```

4.3、随机数

我们有学过python中生成随机数的模块random，在numpy中也有一个随机数生成函数，它在 `np.random` 的子包当中。在Python自带的random当中只能生成一些简单、基础的随机数，而在 `np.random` 当中是可以生成一些高级的随机数的。`np.random` 要比Python自带的random包更加灵活。

函数	功能
rand	返回给定维度的随机数组（0到1之间的数）
randn	返回给定维度的随机数组
randint	返回给定区间的随机整数
choice	给定的一维数组中随机选择
shuffle	原列表上将元素打乱（与random.shuffle相同）

uniform 函数	给定形状产生随机数组 功能
seed	设定随机种子（使相同参数生成的随机数相同）
standard_normal	生成正态分布的随机样本数

```
# rand 返回给定维度的随机数组
np.random.rand(2,2,2)
> array([[0.37992696, 0.18115096],
        [0.78854551, 0.05684808]],

        [[0.69699724, 0.7786954 ],
        [0.77740756, 0.25942256]])

-----

# randn 返回给定维度的随机数组
np.random.randn(2,4)
> array([[ 0.76676877,  0.21752554,  2.08444169,  1.51347609],
        [-2.10082473,  1.00607292, -1.03711487, -1.80526763]])

-----

# randint 返回给定区间的随机整数
np.random.randint(0,20)
> 15

-----

# choice 给定的一维数组中随机选择
np.random.choice(9,3) # 从np.range(9)中，（默认）有放回地等概率选择三个数
> array([5, 8, 2])
np.random.choice(9,3,replace=False) # 无放回地选择
> array([1, 2, 0])

-----

# shuffle 原列表上将元素打乱（与random.shuffle相同）
arr = np.arange(10)
np.random.shuffle(arr)
arr
> array([4, 5, 0, 3, 7, 8, 9, 1, 2, 6])

-----

# uniform 给定形状产生随机数组
np.random.uniform(0,1,[3,3,3])
> array([[0.80239474, 0.37170323, 0.5134832 ],
        [0.42046889, 0.40245839, 0.0812019 ],
        [0.8788738 , 0.48545176, 0.73723353]],

        [[0.79057724, 0.80644632, 0.65966656],
        [0.43833643, 0.53994887, 0.46762885],
        [0.44472436, 0.08944074, 0.34148912]],

        [[0.7042795 , 0.58397044, 0.13061102],
        [0.22925123, 0.97745023, 0.14823085],
        [0.6960559 , 0.07936633, 0.91221842]])

-----

# seed 设定随机种子（使相同参数生成的随机数相同）
np.random.seed(0) # 当seed(0)时生成以下随机数组
np.random.rand(5)
> array([0.5488135 , 0.71518937, 0.60276338, 0.54488318, 0.4236548 ])
np.random.seed(2) # send(5)时生成以下随机数组
np.random.rand(5)
> array([0.5488135 , 0.71518937, 0.60276338, 0.54488318, 0.4236548 ])
np.random.seed(0) # 再次使用send(0)会发现返回最开始的随机数组
```

```
np.random.rand(5)
> array([0.5488135 , 0.71518937, 0.60276338, 0.54488318, 0.4236548 ])
-----
# standard_normal 生成正态分布的随机样本数
np.random.standard_normal([2,10])
> array([[ -0.17937969, -0.69277058,  1.13782687, -0.16915725, -0.76391367,
          -0.4980731 , -0.36289111,  0.26396031, -0.62964191, -0.4722584 ],
        [ -1.51336104,  1.10762468,  0.17623875, -0.94035354,  0.92959433,
          -1.06279492, -0.88640627,  1.92134696, -0.45978052, -1.08903444]])
```

np.random和Python原生的random的区别：

比较内容	random	np.random
输入类型	非空的列表类型（包括列表、字符串和元组）	非空的列表类型（包括列表、字符串和元组）+ numpy.array类型
输出维度	一个数或一个list（多个数）	可指定复杂的size
指定(a,b)范围	可以	整数可指定，浮点数不行，需自行转换
批量输出	不可	可。通过指定size参数
特定分布	涵盖了常用的几个分布；只能单个输出	几乎涵盖了所有分布；可批量输出

只要能把以上所有的内容掌握，数据分析这门功夫你就算是打通了任督二脉了，学起来轻松又愉快。

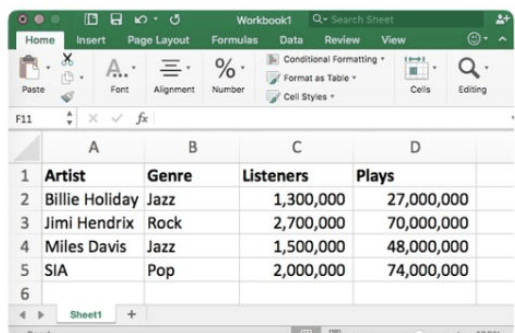
5、数据表示

所有需要处理和构建模型所需的数据类型（电子表格、图像、音频等），其中很多都适合在 n 维数组中表示：

表格和电子表格

表格和电子表格是二维矩阵。电子表格中的每个工作表都可以是它自己的变量。python 中最流行的抽象是 pandas 数据帧，它实际上使用了 NumPy 并在其之上构建。

music.csv



➔

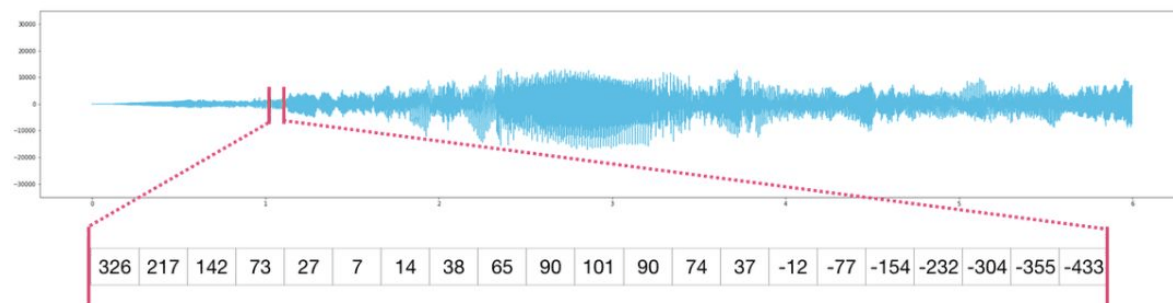
pandas.read_csv('music.csv')

	Artist	Genre	Listeners	Plays
0	Billie Holiday	Jazz	1,300,000	27,000,000
1	Jimi Hendrix	Rock	2,700,000	70,000,000
2	Miles Davis	Jazz	1,500,000	48,000,000
3	SIA	Pop	2,000,000	74,000,000

音频和时间序列

音频文件是样本的一维数组。每个样本都是一个数字，代表音频信号的一小部分。CD 质量的音频每秒包含 44,100 个样本，每个样本是 -65535 到 65536 之间的整数。这意味着如果你有一个 10 秒的 CD 质量 WAVE 文件，你可以将它加载到长度为 $10 * 44,100 = 441,000$ 的 NumPy 数组中。如果想要提取音频的第一秒，只需将文件加载到 audio 的 NumPy 数组中，然后获取 `audio[:44100]`。

以下是一段音频文件：



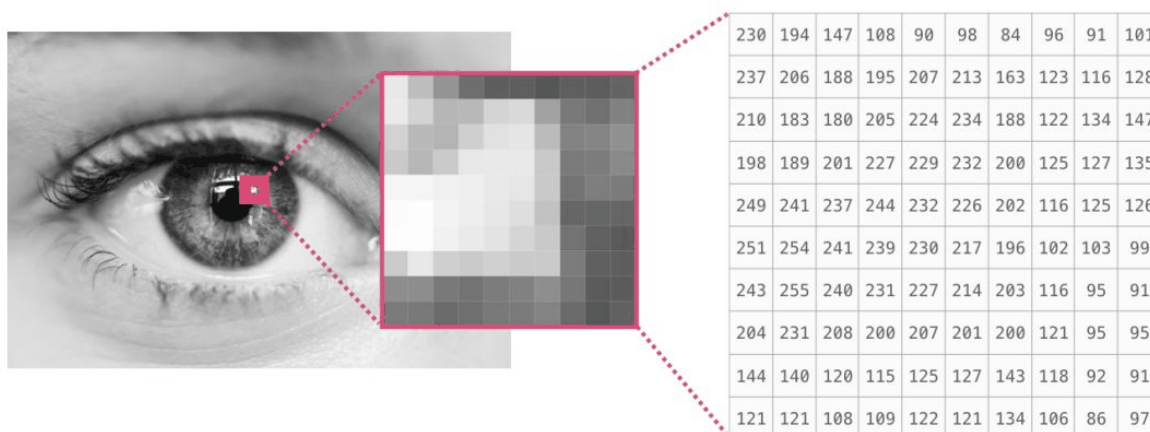
时间序列数据也是如此（如股票价格随时间变化）。

图像

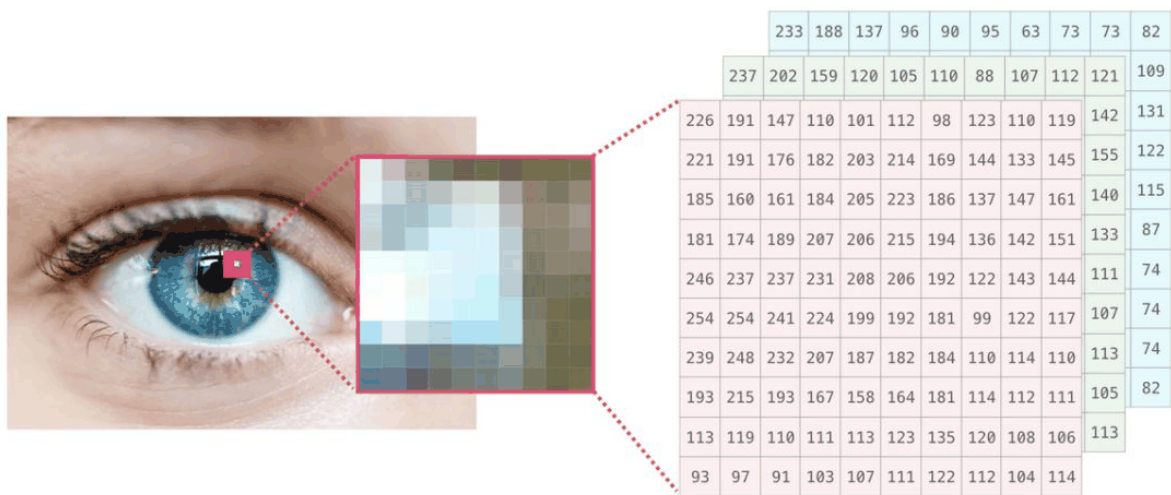
图像是尺寸（高度 × 宽度）的像素矩阵。

如果图像是黑白（即灰度）的，则每个像素都可以用单个数字表示（通常在 0（黑色）和 255（白色）之间）。想要裁剪图像左上角 10 × 10 的像素吗？在 NumPy 写入即可。

下图是一个图像文件的片段：



如果图像是彩色的，则每个像素由三个数字表示——红色、绿色和蓝色。在这种情况下，我们需要一个三维数组（因为每个单元格只能包含一个数字）。因此彩色图像由尺寸为（高 × 宽 × 3）的 ndarray 表示：



6、总结

使用numpy，可以为我们提供一组丰富而又灵活的数据结构，以金融的角度来看的话下面几种类型是最重要的：

基本数据类型

在金融量化当中，整数、浮点数和字符串给我们提供了原子数据类型

标准数据结构

元组、列表、字典和集合，这些在金融当中有许多应用领域，列表通常是最为常用的

数组

说到数组肯定就是今天所学习的numpy中的ndarray数组，它对于数据的处理性能更高，代码更简洁、方便

scipy的使用

Scipy是一个用于数学、科学、工程领域的常用软件包，可以处理插值、积分、优化、图像处理、常微分方程数值解的求解、信号处理等问题。它用于有效计算Numpy矩阵，使Numpy和Scipy协同工作，高效解决问题。

Scipy是由针对特定任务的子模块组成：

模块名	应用领域
scipy.cluster	向量计算/Kmeans
scipy.constants	物理和数学常量
scipy.fftpack	傅立叶变换
scipy.integrate	积分程序
scipy.interpolate	插值
scipy.io	数据输入输出
scipy.linalg	线性代数程序
scipy.ndimage	n维图像包
scipy.odr	正交距离回归
scipy.optimize	优化
scipy.signal	信号处理
scipy.sparse	稀疏矩阵
scipy.spatial	空间数据结构和算法
scipy.special	一些特殊的数学函数
scipy.stats	统计

scipy.io

- 载入和保存matlab文件

```
from scipy import io as spio
from numpy as np
x = np.ones((3,3))
spio.savemat('f.mat',{'a':a})
data = spio.loadmat('f.mat',struct_as_record=True)
data['a']
```

读取图片

```
from scipy import misc
misc.imread('picture')
```

scipy.special

special库中的特殊函数都是超越函数，所谓超越函数是指变量之间的关系不能用有限次加、减、乘、除、乘方、开方 运算表示的函数。如初等函数中的三角函数、反三角函数与对数函数、指数函数都是初等超越函数，一般来说非初等函数都是超越函数。

初等函数：指由基本初等函数经过有限次四则运算与复合运算所得到的函数

下面我们对scipy.special中的部分常用函数进行说明:

- γ 函数

γ 函数，也称伽玛函数，是由欧拉积分定义的函数，是阶乘函数在实数域与复数域上的扩

展，当 $\text{Re}(z) > 0$ 时， γ 函数可以定义为：
$$\gamma(z) = \int_0^{+\infty} \frac{t^{z-1}}{e^t} dt$$

由[解析延拓原理](#)可以将这个定义拓展到整个复数域上，非正整数除外。

在scipy.special中使用scipy.special.gamma()实现 γ 函数的计算，如果对于精度有更高要求，可以使用采用对数坐标的scipy.special.gammaln()函数进行计算。

- 贝塞尔函数

在介绍贝塞尔函数之前，先对贝塞尔方程进行说明，一般来说，我们将形如

$$x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} + (x^2 - \alpha^2)y = 0$$

的二阶线性常微分方程称为贝塞尔方程，而贝塞尔方程的标准解函数 $y(x)$ 就是贝塞尔函数。

其中，参数 α 被称为对应贝塞尔函数的阶。贝塞尔方程是一个二阶线性齐次常微分方程，必然存在两个线性无关的解，然而通常情况下它的解无法用初等函数表示，但是当 $\alpha = n$ 时，注意到 $x = 0$ 是贝塞尔方程的[正则奇点](#)，则由常微分方程的广义幂级数解法可以得出贝塞尔方程的两个广义幂级数解：

$$y_1 = J_n(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{\gamma(n+k+1)\gamma(k+1)} \left(\frac{x}{2}\right)^{2k+n}$$

$$y_2 = J_{-n}(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{\gamma(-n+k+1)\gamma(k+1)} \left(\frac{x}{2}\right)^{2k-n}$$

其中 y_1 被称为第一类贝塞尔函数， y_2 被称为第二类贝塞尔函数（诺依曼函数）。（求解方法参考[常微分方程教程](#) 7.4 广义幂级数解法）。

贝塞尔方程是在[柱坐标](#)或[球坐标](#)下使用[分离变量法](#)求解拉普拉斯方程和[亥姆霍兹方程](#)时得到的，因此贝塞尔函数在波动问题以及各种涉及有势场的问题中占有非常重要的地位，其应用有：

在圆柱形波导中的电磁波传播问题

圆柱体中的热传导问题

圆形或环形薄膜的震动膜态分析问题

信号处理中的调频合成 ([FM synthesis](#))

波动声学

在scipy.special中使用scipy.special.jn()计算 n 阶贝塞尔函数

- 椭圆函数

椭圆函数是定义在有限复平面上[亚纯](#)的双周期函数。所谓的双周期是指具有两个基本周期的单复变函数，即存在 ω_1, ω_2 两个非0复数，对任意整数 m, n 有：

$$f(z + n\omega_1 + m\omega_2) = f(z)$$

- 于是

- $\{n\omega_1 + m\omega_2 : n, m \in \mathbb{Z}\}$ 构成 $f(z)$ 的全部周期。

在scipy.special中使用scipy.special.ellipj()函数计算椭圆函数。

- Erf(高斯曲线的面积)

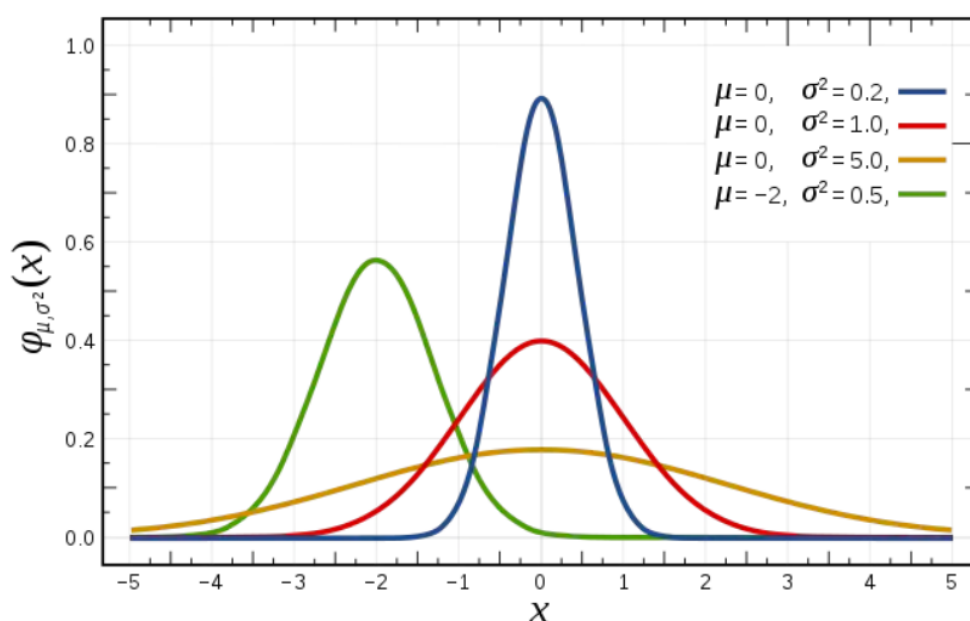
高斯曲线是指高斯分布也就是我们常说的正态分布

$$X \sim N(\mu, \sigma^2)$$

其概率密度函数为：

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

其概率密度函数曲线就是高斯曲线也叫钟形曲线，如下：



不同参数下的高斯曲线

当

时，高斯分布被称为标准正态分布。

scipy.special使用scipy.special.erf()计算高斯曲线的面积。

scipy.linalg

- scipy.linalg.det():计算方阵的行列式
- scipy.linalg.inv():计算方阵的逆
- scipy.linalg.svd():[奇异值](#)分解

scipy.fftpack

[快速傅立叶变换](#)（FFT），是快速计算序列的离散傅立叶变换（DFT）或其逆变换的方法。FFT会通过把DFT矩阵分解为稀疏因子之积来快速计算此类变换。



傅立叶变换将函数的时域与频域相关联



傅立叶变换将函数的时域与频域相关联

scipy.fftpack使用：

- `scipy.fftpack.fftfreq()`:生成样本序列
- `scipy.fftpack.fft()`:计算快速傅立叶变换

scipy.optimize

`scipy.optimize`模块提供了函数最值、曲线拟合和求根的算法。

- 函数最值

以寻找函数 $f(x) = x^2 + 10\sin(x)$ 的最小值为例进行说明：

首先绘制目标函数的图形：

```

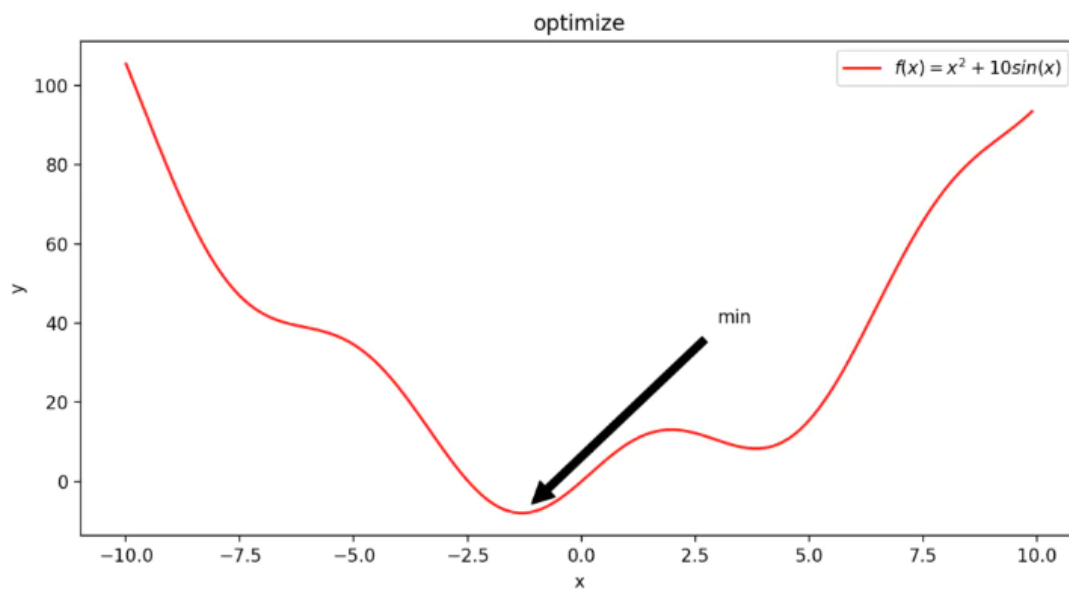
from scipy import optimize
import numpy as np
import matplotlib.pyplot as plt

#定义目标函数
def f(x):
    return x**2+10*np.sin(x)

#绘制目标函数的图形
plt.figure(figsize=(10,5))
x = np.arange(-10,10,0.1)
plt.xlabel('x')
plt.ylabel('y')
plt.title('optimize')
plt.plot(x,f(x),'r-',label='$f(x)=x^2+10sin(x)$')
#图像中的最低点函数值
a = f(-1.3)
plt.annotate('min',xy=(-1.3,a),xytext=(
3,40),arrowprops=dict(facecolor='black',shrink=0.05))
plt.legend()
plt.show()

```

图形输出如下:



先确定最小值所在区间

先确定最小值所在区间

显然这是一个非凸优化问题，对于这类函数得最小值问题一般是从给定的初始值开始进行一个梯度下降，在optimize中一般使用bfgs算法。

```
optimize.fmin_bfgs(f,0)
```

```
Optimization terminated successfully.
Current function value: -7.945823
Iterations: 5
Function evaluations: 18
Gradient evaluations: 6
```

运行结果

结果显示在经过五次迭代之后找到了一个局部最低点-7.945823，显然这并不是函数的全局最小值，只是该函数的一个局部最小值，这也是拟牛顿算法（BFGS）的局限性，如果一个函数有多个局部最小值，拟牛顿算法可能找到这些局部最小值而不是全局最小值，这取决于初始点的选取。在我们不知道全局最低点，并且使用一些临近点作为初始点，那将需要花费大量的时间来获得全局最优。此时可以采用暴力搜寻算法，它会评估范围网格内的每一个点。对于本例，如下：

```
grid = (-10, 10, 0.1)
xmin_global = optimize.brute(f, (grid,))
print(xmin_global)
```

搜寻结果如下：

```
[-1.30641113]
```

全局最小值

但是当函数的定义域大到一定程度时，[scipy.optimize.brute\(\)](#) 变得非常慢。[scipy.optimize.anneal\(\)](#) 提供了一个解决思路，使用模拟退火算法。

可以使用[scipy.optimize.fminbound](#)(function,a,b)得到指定范围([a,b])内的局部最低点。

○ 函数零点

[scipy.optimize.fsolve\(f,x\)](#):函数可以求解 $f=0$ 的零点， x 是根据函数图形特征预先估计的一个零点。

○ 曲线拟合

[scipy.optimize.curve_fit\(\)](#):非线性最小二乘拟合

```
from scipy import optimize

xdata = np.linspace(-10, 10, num=20)
ydata = f(xdata) + np.random.randn(xdata.size)
def f2(x, a, b):
    return a*x**2 + b*np.sin(x)

guess = [2, 2]
params, params_covariance = optimize.curve_fit(f2, xdata, ydata, guess)
print(params)
```

[scipy.optimize.leastsq\(\)](#):最小二乘法拟合

```

'''
使用最小二乘法拟合直线
'''

import numpy as np
from scipy.optimize import leastsq
import matplotlib.pyplot as plt

#训练数据
xi = np.array([8.19,2.72,6.39,8.71,4.7,2.66,3.78])
yi = np.array([7.01,2.78,6.47,6.71,4.1,4.23,4.05])

#定义拟合函数形式
def func(p,x):
    k,b = p
    return k*x+b

#定义误差函数
def error(p,x,y,s):
    print(s)
    return func(p,x)-y

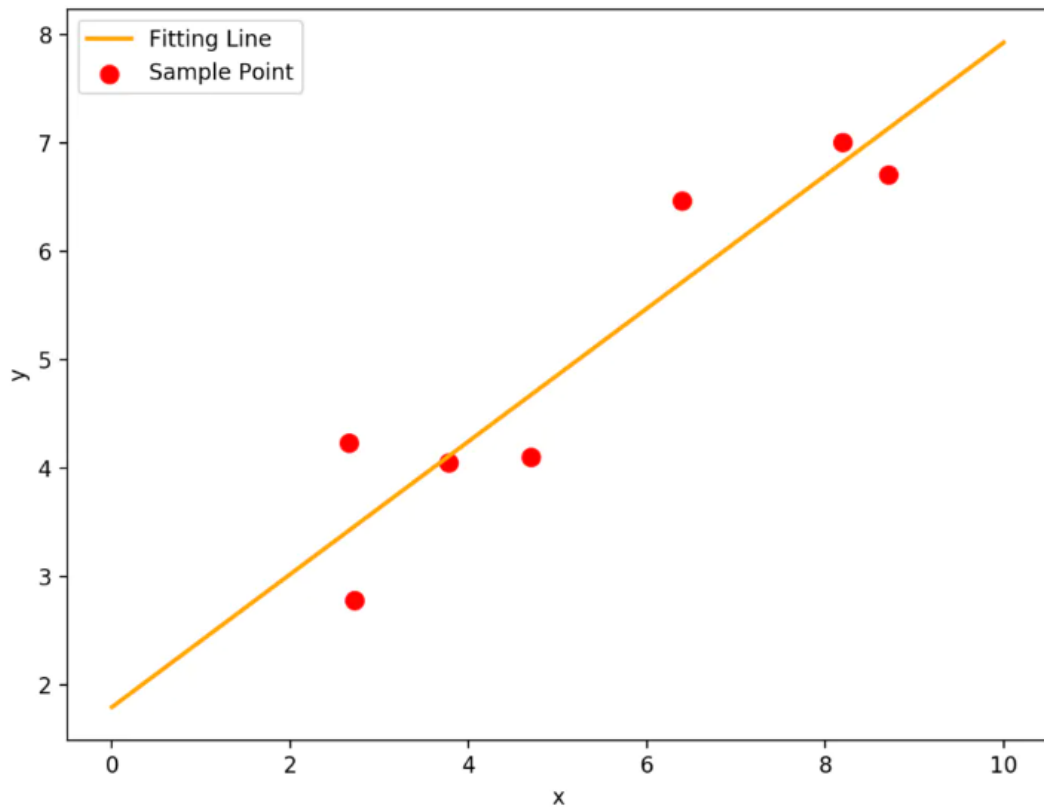
#随机给出参数的初始值
p = [10,2]

#使用leastsq()函数进行参数估计
s = '参数估计次数'
Para = leastsq(error,p,args=(xi,yi,s))
k,b = Para[0]
print('k=',k,'\n','b=',b)

#图形可视化
plt.figure(figsize = (8,6))
#绘制训练数据的散点图
plt.scatter(xi,yi,color='r',label='Sample Point',linewidths = 3)
plt.xlabel('x')
plt.ylabel('y')
x = np.linspace(0,10,1000)
y = k*x+b
plt.plot(x,y,color= 'orange',label = 'Fitting Line',linewidth = 2)
plt.legend()
plt.show()

```

拟合效果如下：



最小二乘法拟合直线

```
'''
使用最小二乘法拟合正弦函数
'''

import numpy as np
from scipy.optimize import leastsq
import matplotlib.pyplot as plt

#定义拟合函数图形
def func(x,p):
    A,k,theta = p
    return A*np.sin(2*np.pi*k*x+theta)

#定义误差函数
def error(p,x,y):
    return y-func(x,p)

#生成训练数据
#随机给出参数的初始值
p0 = [10,0.34,np.pi/6]
A,k,theta = p0
x = np.linspace(0,2*np.pi,1000)
#随机指定参数

y0 = func(x,[A,k,theta])
#randn(m)从标准正态分布中返回m个值，在本例作为噪声
y1 = y0 + 2*np.random.randn(len(x))
```

```

#进行参数估计
Para = leastsq(error,p0,args=(x,y1))
A,k,theta = Para[0]
print('A=',A,'k=',k,'theta=',theta)

'''
图形可视化
'''

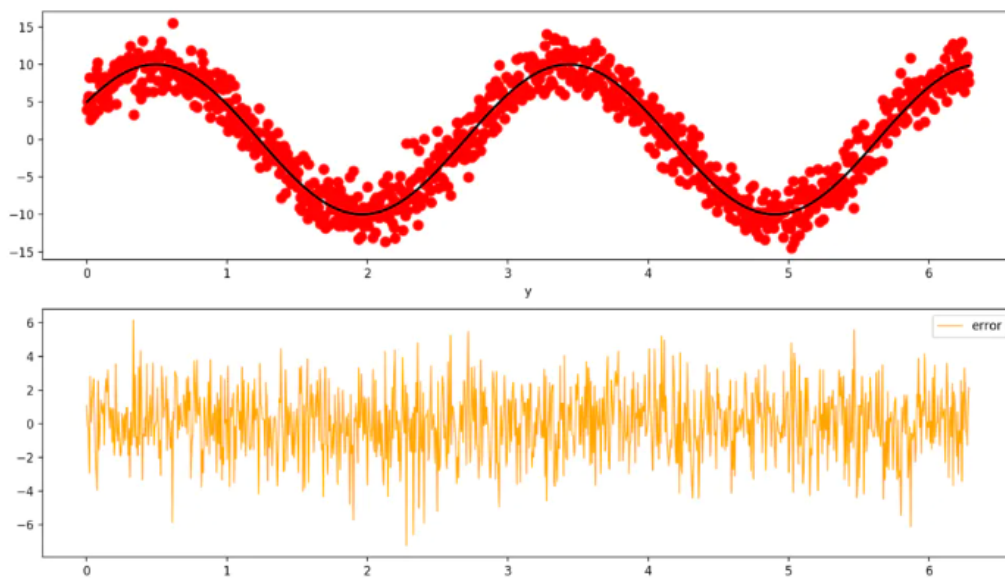
plt.figure(figsize=(20,8))
ax1 = plt.subplot(2,1,1)
ax2 = plt.subplot(2,1,2)

#在ax1区域绘图
plt.sca(ax1)
#绘制散点图
plt.scatter(x,y1,color='red',label='Sample Point',linewidth = 3)
plt.xlabel('x')
plt.ylabel('y')
y = func(x,p0)
plt.plot(x,y0,color='black',label='sine',linewidth=2)

#在ax2区域绘图
plt.sca(ax2)
e = y-y1
plt.plot(x,e,color='orange',label='error',linewidth=1)

#显示图例和图形
plt.legend()
plt.show()

```



最小二乘法拟合曲线

关于Scipy更多内容后续会慢慢进行更新，如有需要请参考：

1.图像模糊

图像的高斯模糊是非常经典的图像卷积例子。本质上，图像模糊就是将（灰度）图像 I 和一个高斯核进行卷积操作： $I_{\sigma} = I * G_{\sigma}$ ，其中 $G_{\sigma} = \frac{1}{2\pi\sigma} e^{-(x^2+y^2)/2\sigma^2}$ 是标准差为 σ 的二维高斯核。高斯模糊通常是其他图像处理操作的一部分，比如图像插值操作、兴趣点计算以及很多其他应用。SciPy 有用来做滤波操作的scipy.ndimage.filters 模块。该模块使用快速一维分离的方式来计算卷积。eg：

```
from PIL import Image
from numpy import *
from scipy.ndimage import filters
im = array(Image.open('empire.jpg').convert('L'))
im2 = filters.gaussian_filter(im,5)    %第二个参数表示标准差
```

随着 σ 的增加，一幅图像被模糊的程度。 σ 越大，处理后的图像细节丢失越多。如果打算模糊一幅彩色图像，只需简单地对每一个颜色通道进行高斯模糊：

```
im = array(Image.open('empire.jpg'))
im2 = zeros(im.shape)
for i in range(3):
    im2[:, :, i] = filters.gaussian_filter(im[:, :, i], 5)
im2 = uint8(im2)
```



原始图像

使用 $\sigma=5$ 的高斯滤波器

2.图像导数

在很多应用中图像强度的变化情况是非常重要的信息。强度的变化可以用灰度图像 I （对于彩色图像，通常对每个颜色通道分别计算导数）的 x 和 y 方向导数 I_x 和 I_y 进行描述。图像的梯度向量为 $\nabla I = [I_x, I_y]^T$ 。梯度有两个重要的属性，一是梯度的大小： $|\nabla I| = \sqrt{I_x^2 + I_y^2}$ ，它描述了图像强度变化的强弱，二是梯度的角度： $\alpha = \arctan2(I_y, I_x)$ ，描述了图像中在每个点（像素）上强度变化最大的方向。NumPy 中的arctan2() 函数返回弧度表示的有符号角度，角度的变化区间为 $-\pi \dots \pi$ 。可以用离散近似的方式来计算图像的导数。图像导数大多数可以通过卷积简单地实现： $I_x = I * D_x$ 和 $I_y = I * D_y$ 。

对于 D_x 和 D_y ，通常选择Prewitt 滤波器：

$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ 和 } D_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

或者Sobel 滤波器:

$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ 和 } D_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

这些导数滤波器可以使用scipy.ndimage.filters 模块的标准卷积操作来简单地实现:

```
from PIL import Image
from numpy import *
from scipy.ndimage import filters
im = array(Image.open('empire.jpg').convert('L')) # 转化为灰度图像
# Sobel 导数滤波器
imx = zeros(im.shape)
filters.sobel(im,1,imx)
imy = zeros(im.shape)
filters.sobel(im,0,imy)
magnitude = sqrt(imx**2+imy**2)
```

上面的脚本使用Sobel 滤波器来计算x 和y 的方向导数, 以及梯度大小。sobel() 函数的第二个参数表示选择x 或者y 方向导数, 第三个参数保存输出的变量。在两个导数图像中, 正导数显示为亮的像素, 负导数显示为暗的像素。灰色区域表示导数的值接近于零。

上述计算图像导数的方法有一些缺陷: 在该方法中, 滤波器的尺度需要随着图像分辨率的变化而变化。为了在图像噪声方面更稳健, 以及在任意尺度上计算导数, 可以使用高斯导数滤波器:

$$I_x = I * G_{ox} \text{ 和 } I_y = I * G_{oy} .$$

G_{ox} 和 G_{oy} 表示 G_σ 在 x 和 y 方向上的导数, G_σ 为标准差为 σ 的高斯函数。

之前用于模糊的filters.gaussian_filter() 函数可以接受额外的参数, 用来计算高斯导数。可以简单地按照下面的方式来处理:

```
sigma = 5 # 标准差
imx = zeros(im.shape)
filters.gaussian_filter(im, (sigma,sigma), (0,1), imx)
imy = zeros(im.shape)
filters.gaussian_filter(im, (sigma,sigma), (1,0), imy)
```

该函数的第三个参数指定对每个方向计算哪种类型的导数, 第二个参数为使用的标准差。