第 3 章

Python常用机器学习库

本章概要

Python 的标准库包括 math、random、datetime、os 等。此外，Python 还拥有强大的第三方库资源，为开发者提供了大量的开发资源。这些库使 Python 保持活力和高效。丰富的开源生态系统也是 Python 成功和流行的原因之一。

借助常用的机器学习库，Python 可以解决多种问题，例如科学计算、数据分析、图像处理等。Python 及其生态系统使它成为全世界用户优先选择的开发工具。

本章会介绍常见的用于数据科学任务和机器学习处理的 Python 库，包括通用的 NumPy、Pandas、Scikit learn 和 Matplotlib 等，也包含 OpenCV、jieba、wordcloud 等专门库。这些库提供了机器学习任务中经常使用的基本功能。

学习目标

当完成本章的学习后，要求：

(1) 了解常用第三方库的调用方法。
(2) 熟悉机器学习库的使用步骤。
(3) 掌握 NumPy、Pandas 的数据处理功能。
(4) 掌握 Matplotlib 的绘图方法。
(5) 熟悉 Scikit learn 机器学习库的使用。
(6) 熟悉其他常用库的使用。

3.1 NumPy

NumPy 是 Numerical Python 的简称，是高性能计算和数据分析的基础包，是 Python 的一个重要扩充库。NumPy 支持高维度数组与矩阵运算，也针对数组运算提供了大量

的数学函数库。NumPy 运算效率极好，是大量机器学习框架的基础库。

NumPy 中主要包含一个强大的 N 维数组对象 ndarray、整合了 C/C++ 和 FORTRAN 代码的工具包，以及丰富的数学函数库，尤其是实用的线性代数、傅里叶变换和随机数生成函数。

使用 NumPy，开发人员可以很方便地执行数组运算、逻辑运算、傅里叶变换和图形图像操作。NumPy 数组的运算效率优于 Python 的标准 List 类型。而且在代码中使用 NumPy 可以省去很多烦琐的处理语句，代码更为简洁。

研究人员经常将 NumPy 和稀疏矩阵运算包 SciPy(Scientific Python)配合使用，来解决矩阵运算问题。将 NumPy 与 SciPy、Matplotlib 绘图库相组合是一个流行的计算框架，这个组合可以作为 MATLAB 的替代方案。


视频讲解

3.1.1 ndarray 对象

NumPy 的强大功能主要基于底层的一个 ndarray 结构，其可以生成 N 维数组对象。

ndarray 对象是一系列同类型数据的集合，下标索引从 0 开始，是一个用于存放同类型元素的多维数组。ndarray 中的每个元素在内存中都具有相同大小的存储区域。

与 Python 中的其他容器对象一样，ndarray 可以通过对数组建立索引或切片来访问数组内容，也可以使用 ndarray 的方法和属性来访问和修改 ndarray 的内容。

1. ndarray 的内部结构

相对标准的数组，ndarray 本质上是一个数据结构。如图 3.1 所示，ndarray 内部主要由以下内容构成。

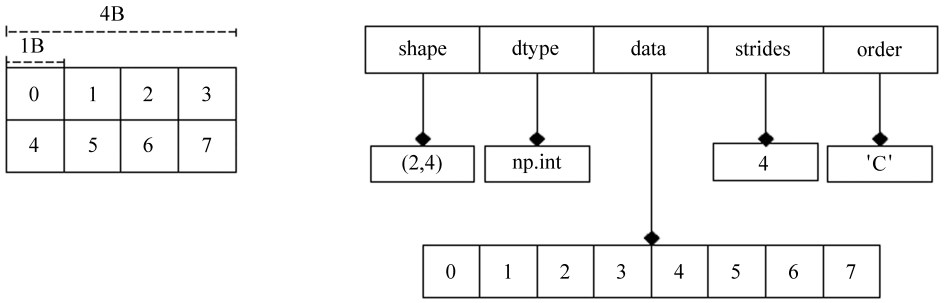

图 3.1 ndarray 的数据结构

(1) 数组形状 shape：是一个表示数组各维大小的整数元组。

(2) 数据类型 dtype：是一个描述数组的类型对象。对象类型为 NumPy 内置的 24 种数组标量类型中的一种。

(3) 数组数据 data：是一个指向内存中数据的指针。

(4) 跨度 strides：一个元组，是当前维度的宽，表示当前维度移动到下一个位置需要跨越的字节数。跨度可以是负数，这样会使数组在内存中后向移动。

(5) 数组顺序 order：访问数组元素的主顺序，如“C”为行主序，“F”为列主序等。

2. 创建 ndarray

在 NumPy 模块中，提供了 ndarray()和 array()两个函数，都可以用来建立一个 ndarray。其中 ndarray()函数属于底层的方法，一般情况下，建立数组使用的是更为便捷的 array()函数，其一般格式如下。

```
numpy.array(object, dtype = None, copy = True, order = None, subok = False,
ndmin = 0)
```

主要参数：

object：数组或嵌套的数列。

dtype：数组元素的数据类型，可选。

order：创建数组的样式，C 为行方向，F 为列方向，A 为任意方向(默认)。

ndmin：指定所生成数组应具有的最小维度。

【例 3.1】 建立一个一维 ndarray 数组。

```
import numpy as np
a = np.array([1,2,3])
print(a)
```

【例 3.2】 创建二维数组。

```
import numpy as np
a = np.array([[1,2], [3,4]])
print(a)
```

【例 3.3】 使用 ndmin 参数设置数组的最小维度。

```
import numpy as np
a = np.array([1,2,3,4,5], ndmin=2)
print(a)
```

【例 3.4】 使用 dtype 参数设置数组类型为复数。

```
import numpy as np
a = np.array([1,2,3], dtype = np.complex)
print(a)
```

在内存中，ndarray 对象的存放形式是连续的一维数列。在访问时，通过索引获取内存块中的对应元素位置。内存块以行顺序(C 样式)或列顺序(F 样式，即 FORTRAN 或 MATLAB 风格)来保存元素。

NumPy 还提供了 asarray()函数，可以将其他类型的结构数据转换为 ndarray。

3.1.2　NumPy 数据类型

NumPy 内置了 24 种数组标量(Array Scaler)类型，也支持 Python 的基本数据类型，某种程度上可以和 C 语言的数据类型相对应，如表 3.1 所示。

表 3.1 NumPy 的基本数据类型

名　称	描　述
bool_	布尔型，True 或 False
int8	有符号字节类型，范围为－128～127
int16	有符号 16 位整数，范围为－32 768～32 767
int32	有符号 32 位整数，范围为-2^{31}～$2^{31}-1$
int64	有符号 64 位整数，范围为-2^{63}～$2^{63}-1$
uint8	无符号字节类型，范围为 0～255
uint16	无符号 16 位整数，范围为 0～65 535
uint32	无符号 32 位整数，范围为 0～$2^{32}-1$
uint64	无符号 64 位整数，范围为 0～$2^{64}-1$
float_	64 位浮点数，同 float64
float16	16 位浮点数
float32	32 位浮点数
float64	64 位（双精度）浮点数，同 float_
complex_	128 位复数*，同 complex128
complex64	64 位复数
complex128	128 位复数，同 complex_

* 复数：形如 z=a+bi(a，b 均为实数)的数称为复数，其中，a 称为实部，b 称为虚部，i 称为虚数单位(i 表示－1 的平方根)。

当虚部 b=0 时，复数 z 是实数；

当虚部 b!=0 且实部 a=0 时，复数 z 是纯虚数。

对于每种数据类型，NumPy 还提供了同名的类型函数，例如 float16()、int32()等，可以用来创建该类型的数据对象，也可以用来转换数据对象的数据类型。

【例 3.5】 NumPy 数据类型的使用。

```
import numpy as np
x=np.float32(5)
print('x 为:',x)
print('x 对象的 data 属性: ',x.data)
print('x 对象的 size 属性:',x.size)
print('x 对象的维数:',x.ndim)
y=np.bool_(x)
print('转换为 bool 类型的 x 为:',y)
z=np.float16(y)
print('True 值转换为 float16 类型为:',z)
```

运行结果为：

```
x为: 5.0
x对象的data属性:  <memory at 0x000002D11F41FDC8>
x对象的size属性: 1
x对象的维数: 0
转换为bool类型的x为: True
True值转换为float16类型为: 1.0
```

上面的函数能够设置、修改对象数据类型。不过通常情况下，建议使用 NumPy 中的 dtype 对象指定数据类型。

1. 数据类型对象（dtype）

NumPy 中的 dtype(data type object)是由 numpy.dtype 类产生的数据类型对象，其作用是描述数组元素对应的内存区域的各部分的使用。其内部结构包括数据类型、数据的字节数、各组成部分的顺序、各字段的名称等。

构造 dtype 对象的语法如下。

```
numpy.dtype(object, align, copy)
```

主要参数如下。

object：要转换为 dtype 对象的数据对象。

align：如果为 True，则填充字段使其类似 C 的结构体。

copy：指明是否复制 dtype 对象。如果为 False，则是对内置数据类型对象的引用。

如果使用 dtype 对象设置数据类型，那么可以对例 3.5 做如下修改。

【例 3.6】 使用 dtype 对象设置数据类型。

```
import numpy as np
x=np.array(5,dtype="float32")
print('x为:',x)
print('x对象的data属性: ',x.data)
print('x对象的size属性:',x.size)
print('x对象的维数:',x.ndim)
y=np.array(x,dtype="bool_")
print('转换为bool类型的x为:',y)
z=np.array(y,dtype="float16")
print('True值转换为float16类型为:',z)
```

运行结果如下。

```
x为: 5.0
x对象的data属性:  <memory at 0x000002D11F552588>
x对象的size属性: 1
x对象的维数: 0
转换为bool类型的x为: True
```

```
True 值转换为 float16 类型为: 1.0
```

有些数据类型有简写，例如 int8，int16，int32，int64 四种数据类型可以使用字符串 'i1'，'i2','i4','i8'简写代替。

例如，使用“i4”字符串代替 int32 类型。

```
import numpy as np
dt = np.dtype('i4')
print(dt)
```

2. 使用 astype()修改数据类型

在数组建立之后，也可以使用 NumPy 中数组附带的 astype()方法修改其数据类型，格式如下。

```
array.astype(dtype,order='K',casting='unsafe',subok=True,copy=True)
```

例如，将 y 设置成 float32 类型，可以用：

```
y=y.astype("float32")
```

或者

```
y=y.astype(np.float32)
```

另外，使用 NumPy 数组的 astype()方法还可以把 Python 的数据类型映射给 dtype 类型，如语句 x＝x.astype(float) 与 x＝x.astype(np.float) 运行结果相同。

表 3.2 是常用 Python 对象与 NumPy 的 dtype 对象的对应表，其他数据类型没有与 Python 等效的数据类型。

表 3.2 Python 对象与 dtype 对象对应关系表

Python 对象	dtype 对象
int	numpy.int_
bool	numpy.bool_
float	numpy.float_
complex	numpy.complex_

【扩展】由于 Pandas 是基于 NumPy 数组的，所以也能用 astype()对 DataFrame 的字段进行类型转换。

【例 3.7】 使用 astype()转换 DataFrame。

```
import pandas as pd
df = pd.DataFrame([{'qty':'3', 'num':'50'}, {'qty':'7', 'num':'20'}])
print(df.dtypes)
print('---------------')
df['qty'] = df['qty'].astype('int')
```

```
df['num'] = df['num'].astype('float64')
print(df.dtypes)
```

可以看到，两个列的数据类型由初始的 object 变成了 float64 和 int32。

```
DataFrame 的 dtypes:
Num    object
Qty    object
dtype: object
---------------------------
astype()转换后的 dtypes:
Num    float64
Qty      int32
dtype: object
```

3.1.3　NumPy 数组属性

1. 常用术语

(1) 轴(Axis)：每个线性数组称为一个轴，轴即数组的维度(Dimensions)。例如，将二维数组看作一维数组，此一维数组中每个元素又是一个一维数组，则每个一维数组是 NumPy 中的一个轴。第一个轴相当于底层数组，第二个轴是底层数组中的数组。

(2) 秩(Rank)：秩描述 NumPy 数组的维数，即轴的数量。一维数组的秩为 1，二维数组的秩为 2，以此类推。

例如，[0,1,2]是一维数组，只有一个轴，其秩为 1，轴长度为 3；[[0,1,2],[3,4,5]]是一个二维数组，数组的秩为 2，具有两个轴，其中第一个轴(维度)的长度为 2，第二个轴(维度)的长度为 3。

在使用的时候可以声明 axis。如果 axis=0，表示按第 0 轴方向操作，即对每列进行操作；如果 axis=1，表示按第 1 轴方向操作，即对每行进行操作。

【例 3.8】 使用 axis 参数设置当前轴。

```
import numpy as np
arr=np.array([[0,1,2],[3,4,5]])
print(arr)
print(arr.sum(axis=0))
print(arr.sum(axis=1))
```

运行结果如下。

```
[[0 1 2]
 [3 4 5]]
[3 5 7]
[ 3 12]
```

在这个程序中，首先使用 arr.sum(axis=0)进行垂直(列)方向的求和计算，然后使用 arr.sum(axis=1)沿行方向计算。

2. 基本属性

NumPy 的 ndarray 数组具有属性，可以获得数组的信息，常见属性见表 3.3。

表 3.3 常见的 ndarray 数组属性

属　　性	说　　明
ndarray.ndim	秩，轴的数量
ndarray.shape	数组的维度
ndarray.size	数组元素的总个数
ndarray.dtype	数组元素类型
ndarray.itemsize	每个元素的大小(B)
ndarray.data	实际数组元素

1) ndarray.ndim

在 NumPy 中 ndarray.ndim 返回这个数组的维数，等于秩。reshape()函数可以将数组变形重构，调整数组各维度的大小。

reshape()的格式为：

```
numpy.reshape(a,newshape,order='C')
```

【例 3.9】 使用 reshape()函数调整数组形状。

```
import numpy as np
arr=np.array([0, 1, 2, 3, 4, 5, 6, 7])
#显示数组 arr 的 rank
print('秩为:',arr.ndim)
arr3D = arr.reshape(2,2,2)
print(arr3D)
print ('秩为:',arr3D.ndim)
```

显示结果如下。

```
秩为:  1
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
秩为:  3
```

思考：reshape()函数是否修改原 arr 数组？请使用 print()函数查看。

2) ndarray.shape

ndarray.shape 代表数组的维度，返回值为一个元组。这个元组的长度就是 ndim 属性(秩)。另外，ndarray.shape 也可以用于调整数组大小。

【例 3.10】　显示数组的维度。

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print (a.shape)
```

【例 3.11】　调整数组大小。

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
a.shape =  (3,2)
print (a)
```

3）数据类型 dtype

数据类型对象 dtype 是一个特殊的对象，包含 ndarray 将一块内存解析成特定数据类型所必需的信息。

【例 3.12】　创建 dtype 数据类型对象。

```
myArr=np.array([1,2,3],dtype=np.float64)
myArr.dtype
```

查看运行结果：

```
dtype('float64')
```

3.1.4　其他创建数组的方式

ndarray 数组可以使用 array()函数来构造。此外，还有其他几种方式可以用来创建特殊的数组。

1. numpy.empty()

NumPy 的 empty()函数能创建一个指定形状、数据类型的空数组。这个数组没有经过初始化，其内容为空。表 3.4 为创建空数组的参数。

格式：

```
numpy.empty(shape, dtype = float, order = 'C')
```

表 3.4　创建空数组的参数

参数	描　　述
shape	数组形状
dtype	数据类型，可选
order	有"C"和"F"两个选项，分别代表行优先和列优先，表示在计算机内存中存储元素的顺序

【例 3.13】　创建一个空数组。

```
import numpy as np
```

```
x = np.empty([3,2], dtype = int)
print(x)
```

运行后得到的数组元素值是不确定的,因为所用空间未初始化。

2. numpy.zeros()

有时需要创建全 0 填充的数组,此时可以使用 NumPy 的 zeros()函数,其参数见表 3.5。
格式:

```
numpy.zeros(shape, dtype = float, order = 'C')
```

表 3.5 创建 zeros 数组的参数

参数	描 述
shape	数组形状
dtype	数据类型,可选
order	'C' 用于 C 风格——行为主的数组,或者 'F' 用于 FORTRAN 风格——列为主的数组

【例 3.14】 创建一个全 0 数组。

```
import numpy as np
#默认为浮点数
x = np.zeros(5)
print(x)
#设置类型为整数
y = np.zeros((5,), dtype = np.int)
print(y)
#自定义类型
z = np.zeros((2,2), dtype = [('x', 'i4'), ('y', 'i4')])
print(z)
```

3. numpy.ones()

有时需要一个以 1 填充的数组,这时可以使用 NumPy 专门提供的 ones()函数来创建。
格式:

```
numpy.ones(shape, dtype=None, order='C')
```

【例 3.15】 建立一个全 1 数组。

```
import numpy as np
#默认为浮点数
x = np.ones(5)
print(x)
#自定义类型
```

```
x = np.ones([2,2], dtype = int)
print(x)
```

4. 产生数列的函数

在进行科学运算时，经常用到基本的简单数列，如1～50等。Python中提供了range()函数。NumPy中也有类似的函数，如arange()、linspace()函数等。

1) range()函数

Python内置的range()函数通过指定开始值、终值和步长可以创建一个一维数组。注意，生成的数组不包括终值。

格式：

```
range(start, stop [,step])
```

所生成的数组从start开始，到stop－1结束，间隔(步长)为step。默认情况下从0开始。step默认为1，需要是整数。

例如：

```
arr1=range(0,5,1)
```

2) arange()函数

NumPy的arange()函数功能与range()函数类似，在start开始到stop的范围内，生成一个ndarray数组。

格式：

```
arange([start,] stop [, step,], dtype=None)
```

【例3.16】 生成3～9的步长为0.2的数组。

```
import numpy as np
arr2=np.arange(3,9,0.2)
arr2
```

运行结果如下。

```
array([3. , 3.2, 3.4, 3.6, 3.8, 4. , 4.2, 4.4, 4.6, 4.8, 5. , 5.2, 5.4,
       5.6, 5.8, 6. , 6.2, 6.4, 6.6, 6.8, 7. , 7.2, 7.4, 7.6, 7.8, 8. ,
       8.2, 8.4, 8.6, 8.8])
```

3) linspace()函数

格式：

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

其中，start为序列的起始值，stop为结束值，num是生成的样本数。

【例3.17】 生成1～5中的10个数。

```
import numpy as np
```

```
arr3=np.linspace(1, 5, 10)
arr3
```

运行结果如下。

```
array([1.        , 1.44444444, 1.88888889, 2.33333333, 2.77777778,
       3.22222222, 3.66666667, 4.11111111, 4.55555556, 5.        ])
```

5. 使用随机函数创建数组

除了简单的顺序数列，NumPy 还在 random 子模块中提供了随机函数，常见的随机函数见表 3.6。

表 3.6 常用的 NumPy 随机函数

函　数	描　述
rand(d0,d1,…,dn)	随机产生指定维度的浮点数组
randint(low[,high,size,dtype])	随机产生[low,high]的整数
random([size])	随机产生[0.0, 1.0)的浮点数
uniform(start,end,size)	从[start,end)均匀分布的数据中随机抽取一组浮点数
normal(loc, scale, size)	基于给定的均值和方差，随机产生一组正态分布的浮点数

* 正态分布(Normal Distribution)：又称高斯分布，是许多统计方法的理论基础。分布图形左右对称。正态分布的参数包括均值和标准差。同样的均值情况下，标准差越大，曲线越平阔；标准差越小，曲线越狭窄。使用 np.random.normal()函数可以生成服从正态分布的数据。

【例 3.18】 创建随机数组。

```
#生成 2 行 3 列的随机浮点数组
np.random.rand(2,3)
#生成 2 行 2 列的 10 以内的随机整数数组
np.random.randint(0,10,(2,2))
#生成 2 行 3 列的[1,2)的随机浮点数组
np.random.uniform(1,2,(2,3))
```

6. 其他数据结构转换成 ndarray

NumPy 中可以通过 array()函数，将 Python 中常见的数值序列，如 List(列表)和 Tuple(元组)等，转换为 ndarray 数组。

【例 3.19】 将 list 类型转换成 ndarray。

```
import numpy as np
#List
```

```
data = [[2000, 'Ohino', 1.5],
        [2002, 'Ohino', 3.6],
        [2002, 'Nevada', 2.9]]
print(type(data))
#List to array
ndarr = np.array(data)
print(type(ndarr))
```

运行结果如下。

```
<class 'list'>
<class 'numpy.ndarray'>
```

3.1.5　切片、迭代和索引


视频讲解

切片是指取数据序列对象的一部分的操作，前面介绍过字符串、列表、元组都支持切片语法。ndarray 数组与其他数据序列类似，也可以进行索引、切片和迭代。

1. 切片

对 ndarray 进行切片操作与一维数组相同，用索引标记切片的起始和终止位置即可。因为 ndarray 可以是多维数组，在进行切片时，通常需要设定每个维度上的切片位置。

NumPy 还提供了一个 copy()方法，可以根据现有的 ndarray 数组创建新的 ndarray 数组。使用 copy()方法与切片，可以用原数组的一部分生成新数组。

【例 3.20】 创建二维 ndarray 的切片。

```
import numpy as np
#创建一个 4 行 6 列的二维数组
arr = np.arange(24).reshape(4,6)
print('arr =\n',arr)
#截取第 2 行到最后一行、第 1 列到第 3 列构成的 ndarray
arr1 = arr[1:, :3]
print('B = \n',arr1)
```

运行结果如下。

```
arr =
 [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
B =
 [[ 6  7  8]
 [12 13 14]
 [18 19 20]]
```

【例 3.21】 使用 numpy.copy()函数对 ndarray 数组进行切片复制。

```
import numpy as np
#创建一个 4 行 6 列的二维数组
arr = np.arange(24).reshape(4, 6)
print('arr =\n',arr)
#切片复制 arr 的第 2 行到第 4 行、第 1 列到第 3 列
arr2 = np.copy(arr[1:4, 0:3])
print('A = \n',arr2)
#复制 arr2 到 arr3
arr3 = arr2.copy()
print('B = \n',arr3)
```

2. 迭代

与其他数据序列类似,ndarray 也可以通过 for 循环实现迭代。当维数多于一维时,迭代操作使用嵌套的 for 循环。

迭代时,通常按照第一条轴(默认为行)对二维数组进行扫描。如果需要按其他维度迭代,可以使用 apply_along_axis(func,axis,arr)函数指定当前处理的轴。

此外,NumPy 还包含一个循环迭代器类 numpy.nditer,所生成的迭代器(Iterator)对象是一个根据位置进行遍历的对象。这是一个有效的多维迭代器对象,与 Python 内置的 iter()函数类似,每个数组元素可使用迭代器对象来访问,从而很方便地对数组进行遍历。

【例 3.22】 使用嵌套 for 循环对 ndarray 数组进行迭代遍历。

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
for xline in a:
    for yitem in xline:
        print(yitem,end=' ')
```

运行结果如下。

```
0 5 10 15 20 25 30 35 40 45 50 55
```

【例 3.23】 使用 nditer 对象对 ndarray 数组进行迭代。

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print(a)
print(np.nditer(a))
for x in np.nditer(a):
    print(x,end=' ')
```

运行结果如下。

```
[[ 0  5 10 15]
 [20 25 30 35]
 [40 45 50 55]]
<numpy.nditer object at 0x000002D121467CB0>
0 5 10 15 20 25 30 35 40 45 50 55
```

迭代的顺序与数组的内容布局相匹配，不受数据排序的影响。例如，对上述数组的转置进行迭代，可以发现，虽然数据的显示顺序发生了变化，但不影响迭代的顺序。

【例 3.24】 转置数组的迭代。

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print(a)
b = a.T
print(b)
print('Iterator in a:')
for x in np.nditer(a):
    print(x,end='|')
print('\nIterator in a.T:')
for y in np.nditer(b):
    print(y,end='|')
```

运行结果如下。

```
[[ 0  5 10 15]
 [20 25 30 35]
 [40 45 50 55]]
[[ 0 20 40]
 [ 5 25 45]
 [10 30 50]
 [15 35 55]]
Iterator in a:
0|5|10|15|20|25|30|35|40|45|50|55|
Iterator in a.T:
0|5|10|15|20|25|30|35|40|45|50|55|
```

如果需要特定的顺序，可以设置显式参数来强制 nditer 对象使用某种顺序，如例 3.25。

【例 3.25】 数组的访问顺序。

```
import numpy as np
a = np.arange(0,60,5)
a = a.reshape(3,4)
print(a)
print('C 风格的顺序:')
for x in np.nditer(a, order =  'C'):
    print(x,end='|')
```

```
print( '\n'  )
print( 'F 风格的顺序:')
for y in np.nditer(a, order =  'F'):
    print(y,end='|')
```

运行结果如下。

```
[[ 0  5 10 15]
 [20 25 30 35]
 [40 45 50 55]]
C风格的顺序:
0|5|10|15|20|25|30|35|40|45|50|55|

F风格的顺序:
0|20|40|5|25|45|10|30|50|15|35|55|
```

3.1.6 NumPy 计算

NumPy 中的 ndarray 可以直接进行基本运算,包括条件运算、统计运算,以及基本数组运算等。

1. 条件运算

NumPy 中的条件运算除了常见的比较大小运算,还可以使用 where()函数实现查找操作。where()函数格式如下。

```
where(condition, x if true, y if false)
```

该函数根据条件表达式 condition 的值返回特定的数组。当条件为真时返回 x 数组,条件为假时返回 y 数组。

【例 3.26】 简单条件运算。

```
import numpy as np
stus_score = np.array([[80, 88], [82, 81], [84, 75], [86, 83], [75, 81]])
result=[stus_score> 80]
print(result)
```

运行结果如下。

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

【例 3.27】 用 np.where()函数实现数据筛选。

```
import numpy as np
num = np.random.normal(0, 1, (3,4))
print(num)
```

```
num[num<0.5]=0
print(num)
print(np.where(num>0.5,1,0))
```

运行结果如下。

```
[[-1.76760946  1.37716782 -0.93033474  0.89155541]
 [-0.91615883 -1.00495783 -0.66251008  1.64800667]
 [-0.59892913  0.49531236 -0.85283977  0.35239407]]
[[0.          1.37716782 0.          0.89155541]
 [0.          0.         0.          1.64800667]
 [0.          0.         0.          0.        ]]
[[0 1 0 1]
 [0 0 0 1]
 [0 0 0 0]]
```

2. 统计计算

NumPy 提供了丰富的统计函数，常用统计函数如表 3.7 所示。

表 3.7　NumPy 的常用统计函数

函　数	描　述
argmax()	求最大值的索引
argmin()	求最小值的索引
cumsum()	从第一个元素开始累加各元素
max()	求最大值
mean()	求算术平均值
min()	求最小值
std()	求数组元素沿给定轴的标准偏差
sum()	求和

【例 3.28】 ndarray 的统计计算。

```
import numpy as np
stus_score = np.array([[80, 88], [82, 81], [84, 75], [86, 83], [75, 81]])
#求每列的最大值(0 表示列)
result = np.max(stus_score, axis=0)
print(result)
#求每行的最大值(1 表示行)
result = np.max(stus_score, axis=1)
print(result)
#求每行的最小值(1 表示行)
result = np.min(stus_score, axis=1)
print(result)
#求每列的平均值(0 表示列)
```

```
result = np.mean(stus_score, axis=0)
print(result)
```

运行结果如下。

```
[86 88]
[88 82 84 86 81]
[80 81 75 83 75]
[81.4 81.6]
```

3.2 Pandas

Pandas(Python Data Analysis Library)是 Python 的一个数据分析包,是基于 NumPy 的一种工具,是为了解决数据分析任务而创建的。

Pandas 使用强大的数据结构提供高性能的数据操作和分析工具。模块提供了大量的能便捷处理数据的函数、方法和模型,还包括操作大型数据集的工具,从而能够高效分析数据。

Pandas 主要处理以下三种数据结构。

(1) Series:一维数组,与 NumPy 中一维的 ndarray 类似。数据结构接近 Python 中的 List 列表,数据元素可以是不同的数据类型。

(2) DataFrame:二维数据结构。DataFrame 可以理解成 Series 的容器,其内部的每项元素都可以看作一个 Series。DataFrame 是重要的数据结构,在机器学习中经常使用。

(3) Panel:三维数组,可以理解为 DataFrame 的容器,其内部的每项元素都可以看作一个 DataFrame。

这些数据结构都构建在 NumPy 数组的基础之上,运算速度很快。

3.2.1 Series 数据结构

Series 是一种类似于一维数组的对象,它由一组数据以及一组与之相关的数据标签(即索引)组成,数据可以是任何 NumPy 数据类型(整数、字符串、浮点数、Python 对象等)。

1. 创建 Series 对象

创建 Series 对象可以使用函数 pd.Series(data, index),其中,data 表示数据值,index 是索引,默认情况下会自动创建一个 0～N－1(N 为数据的长度)的整数型索引。访问 Series 对象的成员可以使用类似 ndarray 数组的切片访问方法,也可以按索引名访问。

【例 3.29】 创建一个 Series 对象。

```
import pandas as pd
s = pd.Series([1,3,5,9,6,8])
print(s)
```

【例 3.30】　为一个地理位置数据创建 Series 对象。

```
import pandas as pd
#使用列表创建,索引值为默认值
print('--------   列表创建 Series   ----------')
s1=pd.Series([1,1,1,1,1])
print(s1)
print('--------   字典创建 Series   ----------')
#使用字典创建,索引值为字典的 key 值
s2=pd.Series({'Longitude':39,'Latitude':116,'Temperature':23})
print('First value in s2:',s2['Longitude'])
print('--------- 用序列作 Series 索引 ----------')
#使用由 range()函数生成的迭代序列设置索引值
s3=pd.Series([3.4,0.8,2.1,0.3,1.5],range(5,10))
print('First value in s3:',s3[5])
```

运行结果如下。

```
--------   列表创建 Series   ----------
0    1
1    1
2    1
3    1
4    1
dtype: int64
--------   字典创建 Series   ----------
First value in s2: 39
--------- 用序列作 Series 索引 ----------
First value in s3: 3.4
```

2. 访问 Series 数据对象

1）修改数据

可以通过赋值操作直接修改 Series 对象成员的值,还可以为多个对象成员批量修改数据。

【例 3.31】　对例 3.30 创建的 s2,将温度增加 2℃,设置城市为 Beijing。

```
#温度增加 2℃,设置城市为 Beijing
s2["City"]="Beijing"
s2['Temperature']+=2
s2
```

运行结果如下。

```
Longitude          39
Latitude          116
```

```
Temperature        25
City          Beijing
dtype: object
```

2）按条件表达式筛选数据

【例 3.32】 找出 s3 中大于 2 的数据。

```
s3[s3>2]
```

输出结果如下。

```
5    3.4
7    2.1
dtype: float64
```

3）增加对象成员

两个 Series 对象可以通过 append()函数进行拼接，从而产生一个新的 Series 对象。进行拼接操作时，原来的 Series 对象内容保持不变。

【例 3.33】 为 s2 添加一项湿度数据。

```
stiny=pd.Series({'humidity':84})
s4=s2.append(stiny)
print('--------原 Series:--------\n',s2)
print('--------新 Series:--------\n',s4)
```

输出结果如下。

```
--------原 Series:--------
Longitude          39
Latitude          116
Temperature        25
City          Beijing
dtype: object
--------新 Series:--------
Longitude          39
Latitude          116
Temperature        25
City          Beijing
humidity           84
dtype: object
```

可以看到，合并操作不影响原 Series。结果中原 s2 数据没有变化，新创建的 s4 对象接收了合并后的新数据。

4）删除对象成员

可以通过 drop()函数删除对象成员，可以删除一个或多个对象成员。与 append()函数一样，drop()函数也不改变原对象的内容，而会返回一个新的 Series 对象。