第 3 章

CHAPTER 3

常用的 Python 工具

本章介绍 Python 数据类编程中常用的工具包，有相关基础的读者可以跳过本章。

3.1 NumPy

NumPy 是一款在 Python 中被广泛使用的开源科学计算库，它的特点是支持高维大型数组的运算，其图标如图 3-1 所示。

图 3-1 NumPy 的图标

在金融领域应用中使用 NumPy 库进行运算，尤其是当涉及大型矩阵运算等时使用 NumPy 计算十分方便。除此之外，在图像相关的应用中也可以使用 NumPy，输入的图像实际上是一个形状为 (H,W,C) 的高维数组，其中 H、W、C 分别为图像的高度、宽度与通道数（也有其他的图像表示形式，例如将通道数放在最前）。

3.1.1 NumPy 中的数据类型

NumPy 中的数据类型众多，与 C 语言的数据类型较为相近。例如，其中的整型就分为 int8、int16、int32、int64 及它们对应的无符号形式，而 Python 中的整型则只有 int 进行表示，同样对于 float 也是类似的。

数据类型之间的转换使用 np.[类型](待转换数组)或.as_type([类型])。如希望将 int64 类型的数组 a 转换为 float32，则可以使用 np.float32(a)或 a.as_type(np.float32)完成。

3.1.2 NumPy 中数组的使用

1. 创建数组

在 NumPy 中多维数组被称作 ndarray，使用 NumPy 创建多维数组十分方便，可以通过转换 Python 中的列表或元组得到，也能直接通过 NumPy 中的函数创建。NumPy 中对数

组进行操作的函数所返回的数据都是 ndarray。例如要创建一个形状为(2,3,4)一共 24 个 1 的 ndarray,可以通过以下的两种方式进行创建,代码如下:

```
//ch3/test_numpy.py
import numpy as np

#方法 1: 通过列表创建数组
a_list = [
    [[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]],
    [[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
]

a_np1 = np.array(a_list)
print(a_np1)
#<class 'numpy.ndarray'>
print(type(a_np1))
#<class 'numpy.int32'>
print(type(a_np1[0][0][0]))

#方法 2: 通过 NumPy 函数创建数组
a_np2 = np.ones(shape=[2, 3, 4])
print(a_np2)
#<class 'numpy.ndarray'>
print(type(a_np2))
#<class 'numpy.float64'>
print(type(a_np2[0][0][0]))
```

使用 python test_numpy.py 运行程序后,发现通过两种方式创建的数组都可以得到形状为(2, 3, 4)的数组,并且它们的类型都是<class 'numpy.ndarray'>。不同的是,a_np1 与 a_np2 中的元素数据类型分别为<class 'numpy.int32'>与<class 'numpy.float64'>,这是因为 a_list 中的元素原本是 Python 中的 int 型,所以在转换为 ndarray 时是 numpy.int32 型。如果将 a_list 中的元素改为 1.或 1.0,此时 a_np1 与 a_np2 中的元素类型则都是<class 'numpy.float64'>。使用 ndarray 的 astype 方法进行类型转换,代码如下:

```
a_np2_int = a_np2.astype(np.int32)
#<class 'numpy.int32'>
print(type(a_np2_int[0][0][0]))
```

NumPy 除了提供了创建所有元素为 1 数组的方法 np.ones,相似地,也可以使用 np.zeros 创建所有元素为 0 的数组,代码如下:

```
b_np = np.zeros([2, 3, 4])
print(b_np)
```

除了可以用 np.ones 和 np.zeros 创建指定形状的数组,也可以使用 np.ones_like 与 np.zeros_like 创建和已知数组形状相同的全 1 或全 0 数组,其过程相当于先获得目标数组的形状,再使用 np.ones 或 np.zeros 进行创建,代码如下:

```
#创建和 b_np 数组形状相同的数组,其中值全为 1
one_like_b_np = np.ones_like(b_np)
print(one_like_b_np)
#(2, 3, 4)
print(one_like_b_np.shape)
```

2. 创建占位符

当不知道数组中的每个元素的具体值时,还可以使用 np.empty 来创建一个“空”数组作为占位符,这个“空”只是语义上而言的,其实数组中存在随机的数值。NumPy 对使用 empty 方法创建的数组元素随机进行初始化,而后期需要做的是为数组中的元素进行赋值,代码如下:

```
#empty1 和 empty2 中的值都是随机初始化的,empty 方法实际上是创建了占位符,运行效率高
empty1 = np.empty([2, 3])
print(empty1)
empty2 = np.empty([2, 3, 4])
print(empty2)
```

3. 数组的属性

所有的 ndarray 都有 ndim、shape、size、dtype 等属性,其中 ndim 用来查看数组的维度个数,如 a_np1 的形状为(2,3,4),那么它就是一个三维的数组,ndim 的值为 3,而 shape 是用来查看数组的形状的,即 a_np1.shape 是(2, 3, 4);size 的意义则说明数组中总的元素个数,即 a_np1.size=2×3×4=24,代码如下:

```
#3
print(a_np1.ndim)
#(2, 3, 4)
print(a_np1.shape)
#24
print(a_np1.size)
```

4. 数组的转置

NumPy 可以对高维数组进行转置(transpose),转置是指改变数组中元素的排列关系而不改变元素的数量。转置时需要指定 axes 参数,它指输出的数组的轴顺序与输入数组的轴顺序之间的对应关系。如新建一个形状为(2, 3, 4)的数组 a,其在 0、1、2 轴上的长度分别为 2、3、4,使用 np.transpose(a, axes=[0, 2, 1])表示将 a 数组的 2 轴和 1 轴进行交换,而原数组的 0 轴保持不变,得到的新数组的形状则为(2,4,3)。可以这样理解:原数组 a 是由 2(0 轴长度)个 3×4(1 轴和 2 轴)的矩阵组成的,0 轴保持不变,而只有 1 轴与 2 轴进行转置,即两个 3×4 的矩阵分别进行矩阵转置即为最后的结果,故最终的形状为(2,4,3),具体转置结果的代码如下:

```
//ch3/test_numpy.py
b_list = [
```

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

b_np = np.array(b_list)
print(b_np)
#(2, 3, 4)
print(b_np.shape)

b_np_t1 = np.transpose(b_np, axes=[0, 2, 1])
print(b_np_t1)
#(2, 4, 3)
print(b_np_t1.shape)

b_np_t2 = np.transpose(b_np, axes=[1, 0, 2])
print(b_np_t2)
#(3, 2, 4)
print(b_np_t2.shape)

b_np_t3 = np.transpose(b_np, axes=[1, 2, 0])
print(b_np_t3)
#(3, 4, 2)
print(b_np_t3.shape)

b_np_t4 = np.transpose(b_np, axes=[2, 0, 1])
print(b_np_t4)
#(4, 2, 3)
print(b_np_t4.shape)

b_np_t5 = np.transpose(b_np, axes=[2, 1, 0])
print(b_np_t5)
#(4, 3, 2)
print(b_np_t5.shape)
```

5. 数组的变形

NumPy 支持对数组进行变形(reshape),变形和 3.1.2 节第 4 部分的转置一样,都能改变数组的形状,但是转置改变的是数组元素的排列,而变形改变的是数组元素的分组。换言之,转置前后数组元素的顺序会发生改变,而变形操作不会改变元素之间的顺序关系。比较转置操作和变形操作的异同的代码如下:

```
//ch3/test_numpy.py
b_np_transpose = np.transpose(b_np, axes=[0, 2, 1])
print(b_np_transpose)
#(2, 4, 3)
print(b_np_transpose.shape)

b_np_reshape = np.reshape(b_np, newshape=[2, 4, 3])
print(b_np_reshape)
```

```
#(2, 4, 3)
print(b_np_reshape.shape)
```

从结果可以看出，b_np_transpose 和 b_np_reshape 的形状都是(2, 3, 4)，而数组内部元素的排列不同，b_np_transpose 与原数组 b_np 中元素的排列顺序不同，而 b_np_reshape 的元素排列与原数组相同。

在实际操作中，常需要将数组变形为只有一行或者一列，此时将 newshape 指定为[1，－1]或[－1，1]即可，－1 表示让程序自动求解该维度的长度。np.squeeze 也是一个常用的函数，它可以对数组中长度为 1 的维度进行压缩，其本质也是 reshape。

6. 数组的切分与合并

NumPy 可以将大数组拆分为若干小数组，同时也能将若干小数组合并为一个大数组，切分通常使用 split 方法，而根据不同的需求，通常会使用 stack 或者 concatenate 方法进行数组的合并，下面分别介绍这几种方法的应用。

当使用 split 将大数组切分为小数组时，需要指定切分点的下标或切分的数量(indices_or_sections)及在哪个维度上切分(axis)。当指定切分下标时，需要为 indices_or_sections 参数传入一个切分下标的列表(list)；当指定切分数量时，为 indices_or_sections 参数传入一个整数 k 即可，表示需要将待切分数组沿指定轴平均切分为 k 部分，若指定的 k 无法完成均分，则此时 split 方法会抛出 ValueError，分割的结果为含有若干分割结果 ndarray 的列表。如果需要非均等切分，读者则可以参考 array_split 方法，该方法在此不进行介绍。演示 split 方法的不同使用场景与方法的代码如下：

```
//ch3/test_numpy.py
to_split_arr = np.arange(12).reshape(3, 4)

'''
[[ 0 1 2 3]
 [ 4 5 6 7]
 [ 8 9 10 11]]
'''
print(to_split_arr)
#形状为(3, 4)
print(to_split_arr.shape)

#[array([[0, 1, 2, 3]]), array([[4, 5, 6, 7]]), array([[ 8, 9, 10, 11]])]
axis_0_split_3_equal_parts = np.split(to_split_arr, 3)
print(axis_0_split_3_equal_parts)

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

```
       [10, 11]])]
'''
axis_1_split_2_equal_parts = np.split(to_split_arr, 2, axis=1)
print(axis_1_split_2_equal_parts)

#ValueError,因为轴0的长度为3,所以无法被均分为2份
axis_0_split_2_equal_parts = np.split(to_split_arr, 2)

'''
[array([[0, 1, 2, 3],
        [4, 5, 6, 7]]),
 array([[8, 9, 10, 11]])]
'''
axis_0_split_indices = np.split(to_split_arr, [2, ])
print(axis_0_split_indices)

'''
[array([[ 0, 1, 2],
        [ 4, 5, 6],
        [ 8, 9, 10]]),
 array([[ 3],
        [ 7],
        [11]])]
'''
axis_1_split_indices = np.split(to_split_arr, [3, ], axis=1)
print(axis_1_split_indices)
```

运行以上程序，从控制台打印的结果可以看出 axis_0_split_3_equal_parts 与 axis_1_split_2_equal_parts 分别将原数组在轴 0(长度为 3)和轴 1(长度为 4)平均切分为 3 份和 2 份，此时为 split 的 indices_or_sections 传入的值分别为 3 和 2，代表需要切分的数量，而当尝试在 0 轴上切分为两部分时，程序会报错。当为 split 的 indices_or_sections 传入的值为[2,]和[3,]时会分别得到 axis_0_split_indices 和 axis_1_split_indices，前者表示将原数组在 0 轴上分为两部分，第一部分是 0 轴下标小于 2 的部分，第二部分是下标大于或等于 2 的部分，即分为 to_split_arr[:2,:]和 to_split_arr[2:,:]；类似地，axis_1_split_indices 表示将原数组在 1 轴上分为两部分，分别为 to_split_arr[:,:3]和 to_split_arr[:,3:]。

在 NumPy 中合并数组通常有两种方式：stack 和 concatenate，两者有很多相似之处，也有明显的区别。这两个函数都需要传入待合并的数组列表及指定在哪个轴上进行合并；区别是 stack 会为合并的数组新建一个轴，而 concatenate 直接在原始数组的轴上进行合并。假设现在需要对两个形状都为(3,4)的数组进行合并，使用 stack 函数在 2 轴进行合并时，由于原始数组只有 0 轴和 1 轴，并没有 2 轴，因此 stack 函数会为新数组新建一个 2 轴，得到的数组形状为(3,4,2)，而如果使用 concatenate 在 1 轴上合并，则得到的新数组的形状为(3,4+4)，即(3,8)。这两个函数在合并数组时的异同的代码如下：

```
//ch3/test_numpy.py
#新建两个形状为(3, 4)的待合并数组
```

```
merge_arr1 = np.arange(12).reshape(3, 4)
merge_arr2 = np.arange(12, 24). reshape(3, 4)

print(merge_arr1)
print(merge_arr2)

#stack 为新数组新建一个轴 2
stack_arr1 = np.stack([merge_arr1, merge_arr2], axis = 2)
print(stack_arr1)
#(3, 4, 2)
print(stack_arr1.shape)

#stack 为新数组新建一个轴 1,原始的轴 1 变为轴 2
stack_arr2 = np.stack([merge_arr1, merge_arr2], axis = 1)
print(stack_arr2)
#(3, 2, 4)
print(stack_arr2.shape)

#新数组在原始轴 1 上进行连接
concat_arr1 = np.concatenate([merge_arr1, merge_arr2], axis = 1)
print(concat_arr1)
#(3, 8)
print(concat_arr1.shape)

#新数组在原始轴 0 上进行连接
concat_arr2 = np.concatenate([merge_arr1, merge_arr2], axis = 0)
print(concat_arr2)
#(6, 4)
print(concat_arr2.shape)
```

运行以上程序可以得知,stack 会在 axis 参数指定的轴上新建一个轴,改变合并后数组的维度,而 concatenate 函数仅会在原始数组的某一 axis 上进行合并,不会产生新的轴。

本书仅对 NumPy 基本的用法进行介绍,有兴趣了解其更多强大功能与用法的读者可以到 NumPy 的官方网站进一步学习。

3.2 Matplotlib

Matplotlib 是一个强大的 Python 开源图形库,其图标如图 3-2 所示。可以使用它轻松地绘制各种图形或者对数据进行可视化,如函数图、直方图、饼图等。Matplotlib 常常和 NumPy 配合使用,NumPy 提供绘图中的数据,Matplotlib 对数据进行可视化。

图 3-2 Matplotlib 的图标

3.2.1　Matplotlib 中的相关概念

在 Matplotlib 中，绘图主要通过两种方式，一种方式是使用 pyplot 直接绘图，使用方法较为简便，但是功能比较受限；另一种方式则是通过 pyplot.subplot 返回的 fig 和 axes 对象进行绘图，这种方式的灵活性较强，本节主要对后者进行讲解。

首先需要了解 Matplotlib 中的一些概念，如图 3-3 所示。整个承载图像的画布称作 Figure，Figure 上可以有若干 Axes，每个 Axes（可以将 Axes 认为是子图）有自己独立的属性，如 Title（标题）、Legend（图例）、图形（各种 plot）等。

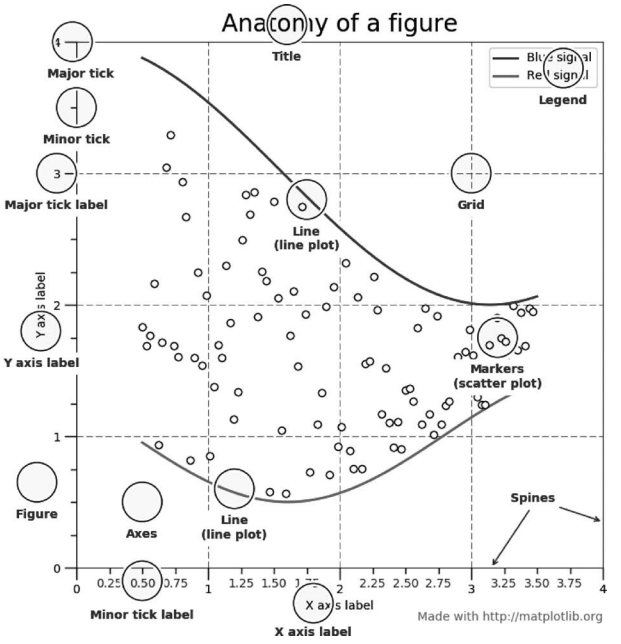


图 3-3　Matplotlib 中的各种概念

在实际使用时，首先使用 plt.subplot 方法创建若干 Axes，再依次对每个 Axes 进行绘图并设置它的 Title 与 Legend 等属性，最后使用 plt.show 或 plt.savefig 方法对图像进行显示或者保存。

3.2.2　使用 Matplotlib 绘图

1. 绘制函数图像

本节将展示如何使用 Matplotlib 绘制函数图像，以使用正弦函数为例。首先，定义数据产生接口正弦函数并得到画图数据，接着创建 figure 与 axes 对象并使用 axes.plot 进行图像的绘制，代码如下：

```
//ch3/test_matplotlib.py
import matplotlib.pyplot as plt
```

```
import numpy as np

#定义数据产生函数
def sin(start, end):
    #使用 np.linspace 产生 1000 个等间隔的数据
    x = np.linspace(start, end, num = 1000)
    return x, np.sin(x)

start = -10
end = 10

data_x, data_y = sin(start, end)

#得到 figure 与 axes 对象,使用 subplots 默认只生成一个 axes
figure, axes = plt.subplots()
axes.plot(data_x, data_y, label = 'Sin(x)')
#显示 plot 中定义的 label
axes.legend()
#在图中显示网格
axes.grid()
#设置图题
axes.set_title('Plot of Sin(x)')
#显示图像
plt.show()
```

运行程序可以得到如图 3-4 所示的函数图像。

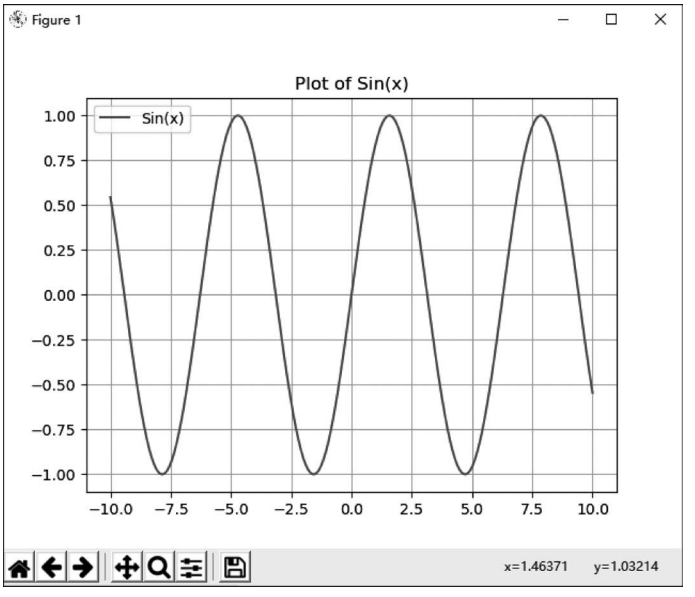


图 3-4　sin(x)在[−10,10]上的图像

下面使用 plt.subplots 绘制多张子图,给 plt.subplots 方法以行列的形式(如给函数传

入(m,n),则表示要画 m 行 n 列总共 $m \times n$ 张子图)传入需要绘制的子图数量。绘制 2 行 3 列一共 6 张正弦函数的图像,代码如下:

```
//ch3/test_matplotlib.py
row = 2; col = 3
fig, axes = plt.subplots(row, col)
for i in range(row):
    for j in range(col):
        #以索引的形式取出每个 axes
        axes[i][j].plot(data_x, data_y, label = 'Sin(x)')
        axes[i][j].set_title('Plot of Sin(x) at [{}, {}]'.format(i, j))
        axes[i][j].legend()
#设置总图标题
plt.suptitle('All 2 * 3 plots')
plt.show()
```

运行以上程序可以得到如图 3-5 所示的图像。

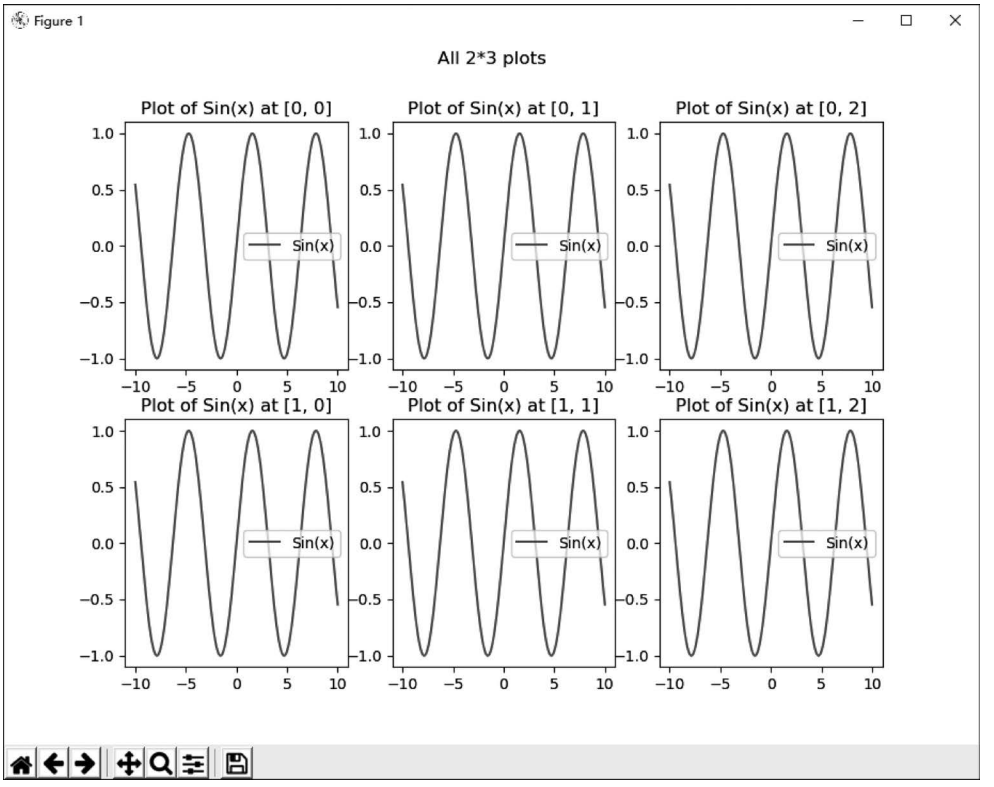


图 3-5　2×3 张 sin(x)在[−10,10]上的图像

2. 绘制散点图

当数据是杂乱无章的点时,常常需要绘制散点图以观察其在空间内的分布情况,此时可以使用 scatter 函数直接进行绘制,其用法与 3.2.2 节第 1 部分的 plot 函数基本类似。在正

弦函数值上引入了随机噪声，并使用散点图呈现出来，代码如下：

```
//ch3/test_matplotlib.py
#从均值为0、标准差为1的正态分布中引入小的噪声
noise_y = np.random.randn( * data_y.shape) / 2
noise_data_y = data_y + noise_y

figure, axes = plt.subplots()
#使用散点图进行绘制
axes.scatter(data_x, noise_data_y, label = 'sin(x) with noise scatter')
axes.grid()
axes.legend()
plt.show()
```

运行程序可以得到如图 3-6 所示的结果，从图中能看出引入小噪声后图像整体仍然维持正弦函数的基本形态，以散点图的形式绘制的结果十分直观。

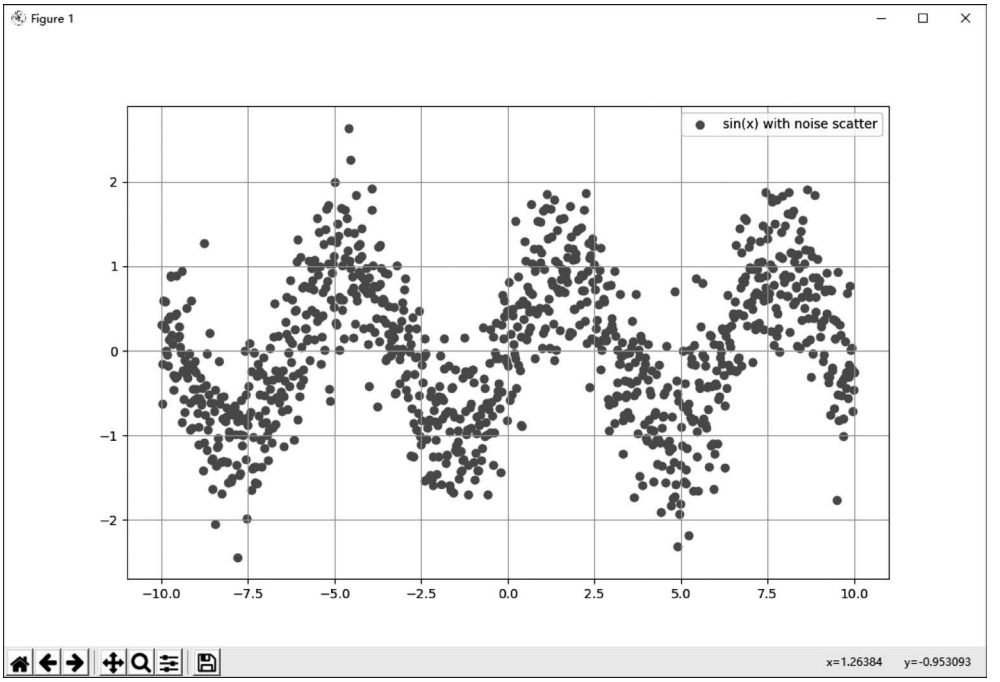


图 3-6　引入小噪声后的正弦函数散点图

3. 绘制直方图

当需要查看数据的整体分布情况时，可以绘制直方图进行可视化，其用法与 3.2.2 节第 1 部分的 plot 函数基本类似，仅仅是可视化的图形表现上有所区别。绘制直方图的代码如下：

```
//ch3/test_matplotlib.py
#生成10 000个正态分布中的数组
```

```
norm_data = np.random.normal(size = [10000])
figure, axes = plt.subplots(1, 2)
#将数据分置于10个桶中
axes[0].hist(norm_data, bins = 10, label = 'hist')
axes[0].set_title('Histogram with 10 bins')
axes[0].legend()
#将数据分置于1000个桶中
axes[1].hist(norm_data, bins = 1000, label = 'hist')
axes[1].set_title('Histogram with 1000 bins')
axes[1].legend()
plt.show()
```

运行程序可以得到如图 3-7 所示的结果，可以看到桶的数量越多结果越细腻，越接近正态分布的结果。

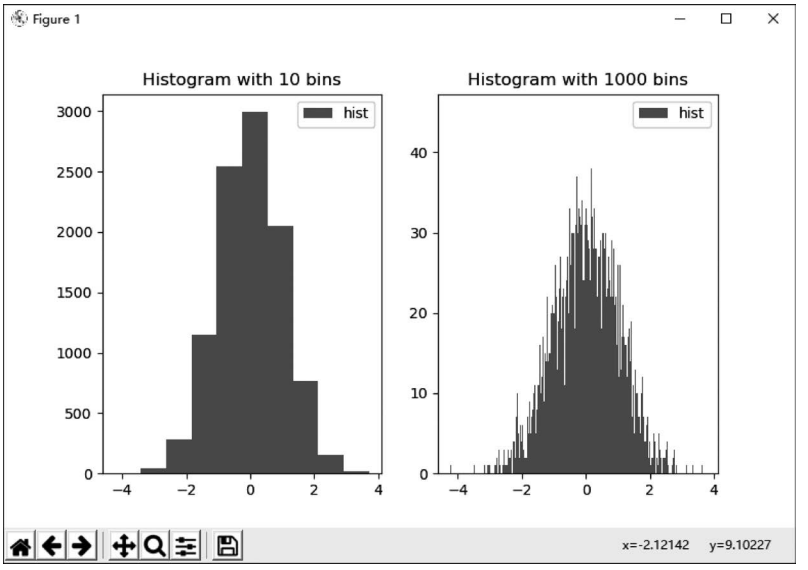


图 3-7　以不同的区间间隔绘制直方图

4. 绘制条形图

使用 Matplotlib 绘制条形图十分方便，条形图常常也被称为柱状图，它的图形表现与直方图十分类似，但是条形图常被用于分类数据的可视化，而直方图则主要用于数值型数据的可视化，这就意味着在横轴上条形图的分隔不需要连续并且区间大小可以不相等，而直方图则需要区间连续并且间隔相等。使用 bar 进行绘图时，需要传入对应的 x 与 y 值，使用条形图绘制正弦函数的图像的代码如下：

```
figure, axes = plt.subplots()
axes.bar(data_x, data_y, label = 'bar')
axes.legend()
axes.grid()
plt.show()
```

运行程序后可以得到如图 3-8 所示的结果，可以看出 Matplotlib 中的条形图可以绘制因变量为负值的图像，此时图像在 x 轴下方，使用 bar 绘制的条形图可以认为是散点图中所有的点向 x 轴作垂线而形成的图形。

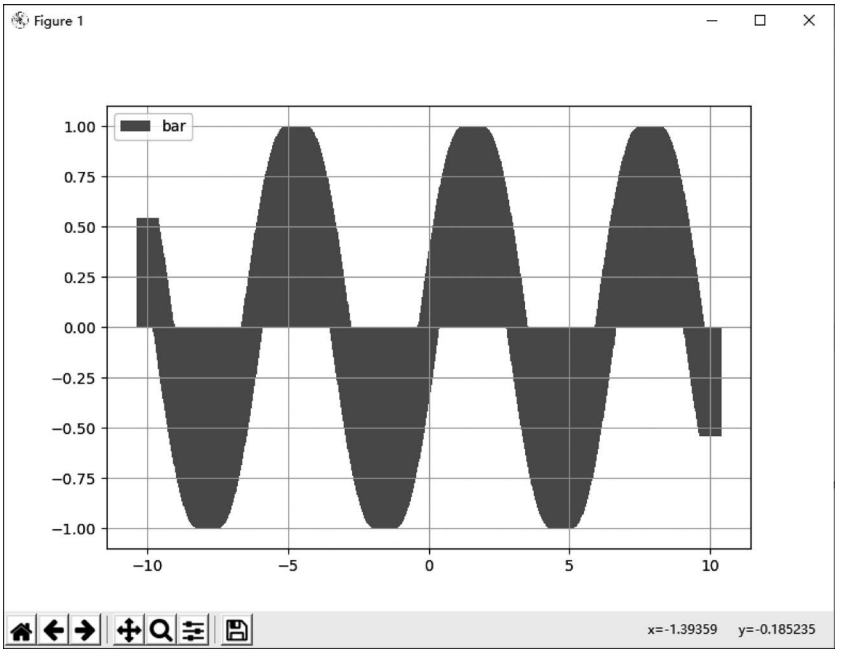


图 3-8 以条形图的形式绘制正弦函数图像

5. 在同一张图中绘制多个图像

3.2.2 节第 1 部分至第 4 部分都仅绘制了单个图像，本节将说明如何在同一张图中绘制多个图像。Matplotlib 会维护一个当前处于活动状态的画布，此时直接在画布上使用绘图函数进行绘制即可，直到程序运行到 plt.show 显示图像时才会将整个画布清除，在一张图中同时绘制正弦函数曲线图与散点图，代码如下：

```
//ch3/test_matplotlib.py
figure, axes = plt.subplots()
#绘制曲线图
axes.plot(data_x, data_y, label = 'Sin(x)', linewidth = 5)
#绘制散点图，此时 axes 对象仍处于活动状态，直接绘制即可
axes.scatter(data_x, noise_data_y, label = 'scatter noise data', color = 'yellow')
axes.legend()
axes.grid()
plt.show()
```

程序的运行结果如图 3-9 所示，由于绘图函数默认使用蓝色，所以在绘制曲线图与散点图时本书额外使用 linewidth 与 color 参数指定线条和点的颜色与宽度，以便读者能看清图像。

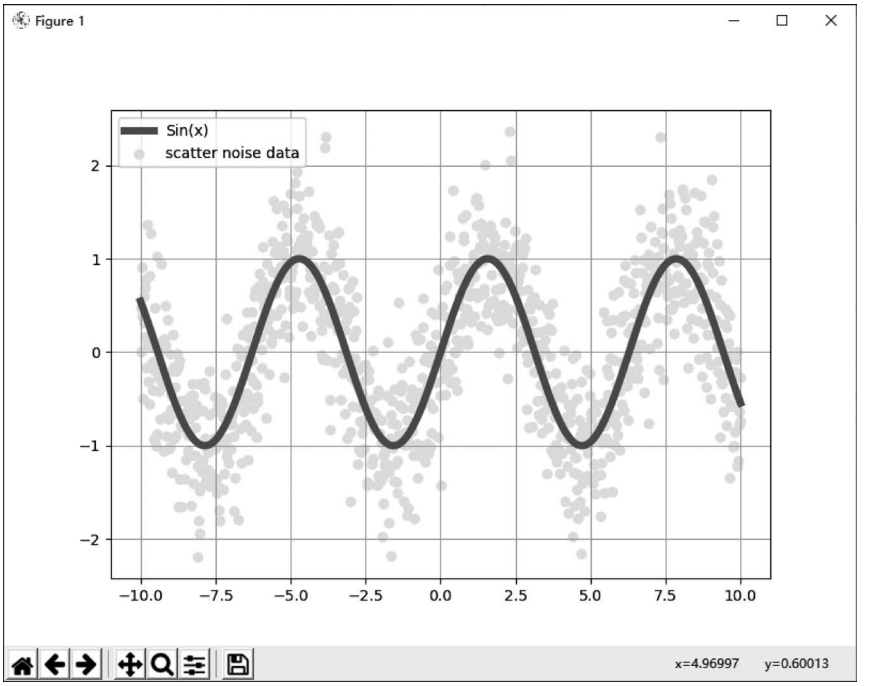


图 3-9　在同一张图中同时绘制曲线图与散点图

6. 动态绘制图像

前面所绘制的图像都是静态图像，当数据随时间变化时，静态图像则不能表现出数据的变化规律，因此本部分将说明如何使用 Matplotlib 绘制实时的动态图像。首先需要对用到的函数进行一些说明。

(1) plt.ion()：用于开启 Matplotlib 中的交互模式(interactive)，开启交互模式后，只要程序遇到绘图指令，如 plot、scatter 等，就会直接显示绘图结果，而不需要再调用 plt.show 进行显示。

(2) plt.cla()：表示清除当前活动的 axes 对象，清除后需要重新绘图以得到结果。相似的指令还有 plt.clf()，这个函数表示清除当前 figure 对象。

(3) plt.pause(time)：延迟函数，由于交互模式下显示的图像会立即关闭，无法看清，所以需要使用 plt.pause 函数使绘制的图像暂停，以便观察。传入的参数 time 用于延迟时间，单位为秒。

(4) plt.ioff()：表示退出交互模式。在绘图完成之后调用。

下面的代码定义了一个带系数的正弦函数，以传入不同的系数来模拟产生和时间相关的数据，并在交互模式下实时显示不同系数的正弦函数图像的变化情况，代码如下：

```
//ch3/test_matplotlib.py
figure, axes = plt.subplots()
#定义时间的长度
```

```
num = 100

#定义带系数的正弦函数,以模拟不同时刻的数据
def sin_with_effi(start, end, effi):
    x = np.linspace(start, end, num = 1000)
    return x, np.sin(effi * x)

#打开 Matplotlib 的交互绘图模式
plt.ion()

#对时间长度进行循环
for i in range(num):
    #清除上一次绘图结果
    plt.cla()
    #取出当前时刻的数据
    data_x, data_y = sin_with_effi(start, end, effi = i / 10)
    axes.plot(data_x, data_y)
    #暂停图像以显示最新结果
    plt.pause(0.001)

#关闭交互模式
plt.ioff()
#显示最终结果
plt.show()
```

运行以上程序可以得到如图 3-10 所示的结果,可以看到随着时间的变化,由于正弦函数的系数越来越大,因此函数图像越来越紧密,能以十分直观的形式观察函数图像的变化情况。

7. 显示图像

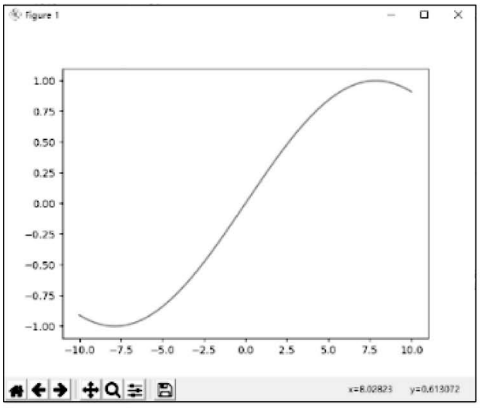
Matplotlib 也可以用于显示图像,代码如下:

```
img_path = 'matplotlib_logo.png'
#读取图像
img = plt.imread(img_path)
#显示图像
plt.imshow(img)
```

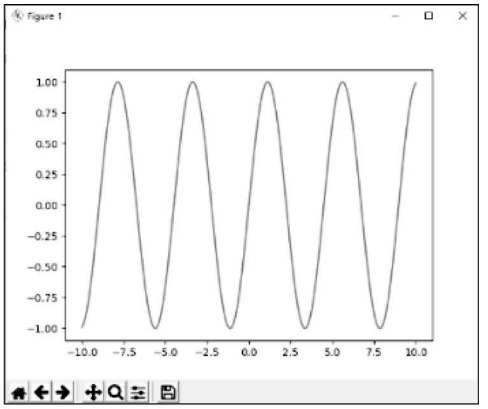
运行程序后,能看到如图 3-11 所示的结果(前提是当前文件夹下有 matplotlib_logo.png 这张图像)。如果需要显示非 PNG 格式的图像,则需要使用 pip install pillow 命令额外安装 pillow 库以获得对更多图像的支持。

plt.imshow 也能以热力图的形式显示矩阵,随机初始化形状为(256, 256)矩阵并使用 Matplotlib 进行显示的代码如下:

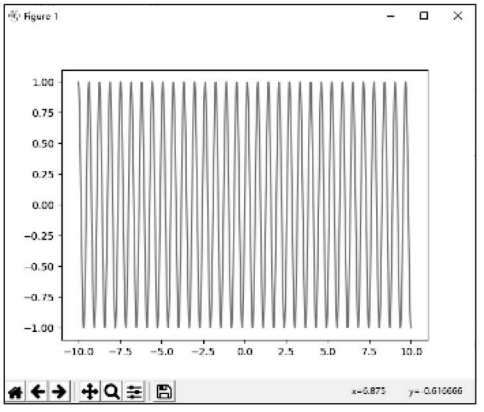
```
//ch3/test_matplotlib.py
row = col = 256
#定义一个空的占位符
heatmap = np.empty(shape = [row, col])
```



(a) 程序运行初期图像



(b) 程序运行中期图像



(c) 程序运行后期图像

图 3-10　在同一张图中同时绘制曲线图与散点图

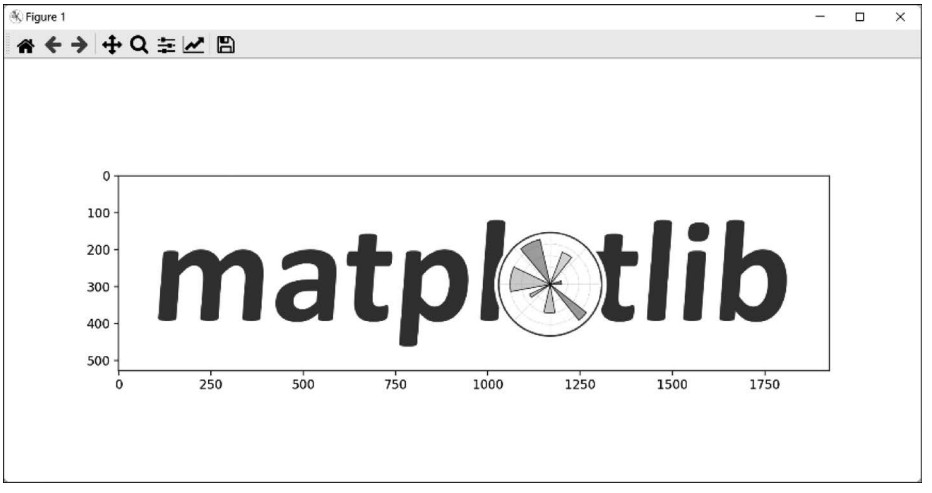


图 3-11　使用 Matplotlib 显示图像

```
#初始化占位符中的每个像素
for i in range(row):
    for j in range(col):
        heatmap[i][j] = np.random.rand() * i + j
#imshow将输入的图像进行归一化并映射至0～255,较小值使用深色表示,较大值使用浅色表示
plt.imshow(heatmap)
plt.show()
```

运行程序后,得到如图 3-12 所示的结果,能看出图像从左上角到右下角的颜色逐渐变亮,说明其值从左上角到右下角逐渐增大,这符合代码中所写的矩阵初始化逻辑。

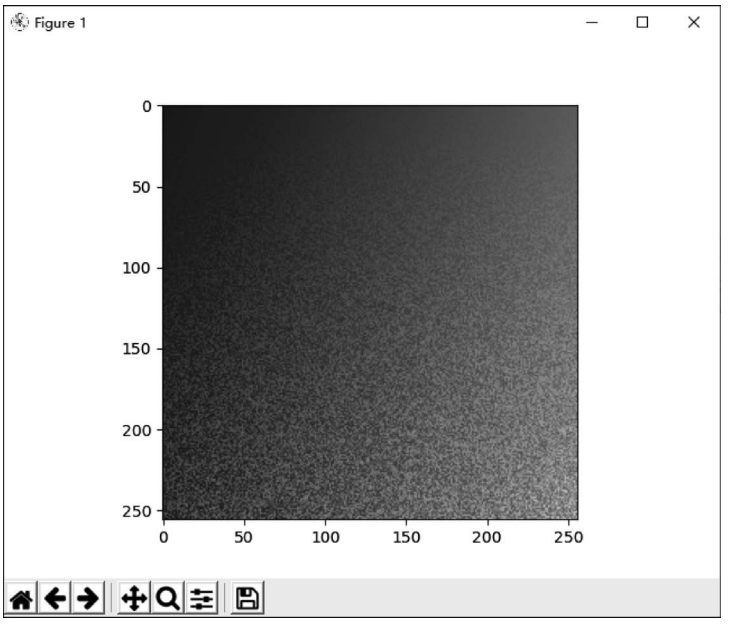


图 3-12　使用 Matplotlib 显示热力图

除了本节所展示的绘图方式与绘图类型,Matplotlib 还有许多更广泛的应用。关于其更多用法可以查看 Matplotlib 官网。

3.3　Pandas

Pandas 是一个用于数据分析的开源 Python 库,其图标如图 3-13 所示。使用 Pandas 能高效地读取和处理类表格格式的数据,例如 CSV、Excel、SQL 数据等。

图 3-13　Pandas 的图标

3.3.1　Pandas 中的数据结构

在 Pandas 中，有两种核心的数据结构，其中一维的数据称为 Series，本书在此不对 Series 进行过多讲解，二维的数据结构被称作 DataFrame，其结构如图 3-14 所示。从图中可以看出 DataFrame 是一种类表格的数据结构，其包含 row 和 column，DataFrame 中的每个 row 或者 column 都是一个 Series。

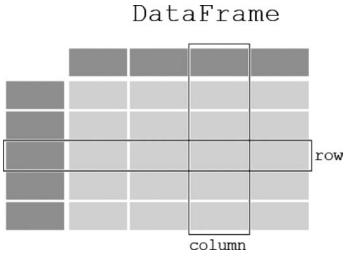


图 3-14　DataFrame 示意图

3.3.2　使用 Pandas 读取数据

1. 读取 CSV 文件

CSV 是逗号分隔值文件，使用纯文本来存储表格数据，前面讲过 Pandas 适用于读取类表格格式的数据，因此本节将展示如何使用 Pandas 便捷地读取 CSV 文件。

首先，打开记事本，在其中输入如图 3-15 所示的数据，并将其保存为 num_csv.csv（也可以不更改文件扩展名，文件内的数据使用逗号分隔即可）。

接下来，使用 pandas.read_csv 方法进行读取即可，代码如下：

```
import pandas as pd

file_name = 'num_csv.csv'
csv_file = pd.read_csv(file_name)
print(csv_file)
```

读取后的结果如图 3-16 所示，可以看到结果中最左边一列的 0 和 1 代表行号，这说明函数认为 CSV 文件中只有两行数据，而第 1 行的 first～fifth 不算作数据，仅算作表头。

```
first, second, third, fourth, fifth
1,2,3,4,5
6,7,8,9,10
```

图 3-15　用于测试的 CSV 数据

	first	second	third	fourth	fifth
0	1	2	3	4	5
1	6	7	8	9	10

图 3-16　读取 CSV 数据的结果

如果需要将第 1 行作为数据考虑，或者需要读取的 CSV 文件没有表头，则需要在 read_csv 方法中指定 header＝None，即数据中不存在表头行，读取 CSV 文件的结果如图 3-17 所示，代码如下：

```
csv_file_wo_header = pd.read_csv(file_name, header = None)
print(csv_file_wo_header)
```

```
      0       1       2       3       4
0  first   second   third   fourth   fifth
1      1       2       3       4       5
2      6       7       8       9      10
```

图 3-17　读取不带 header 的 CSV 数据

从结果可以看出，指定 header＝None 后，左侧行号变为从 0～2，一共 3 列，原本的表头被认为是第 1 行数据。除此之外程序自动给数据加上了表头，以数字进行标识。

如果只想取出数据部分，而不需要表头与行号信息，则可以使用 DataFrame 的 values 属性进行获取，代码如下：

```
csv_file_values = pd.read_csv(file_name).values
print(csv_file_values)
```

运行结果的 csv_file_values 类型是 NumPy 的 ndarray，取出数据后可以进一步使用 NumPy 中的方法对其进行处理。

2. 读取 Excel 文件

Excel 是人们日常生活中最常用的软件之一，Pandas 对于 Excel 文件的读取也提供了十分方便的接口。和 CSV 文件不同，由于 Excel 文件中可以存在多张表(Sheet)，在读取 Excel 文件时需要指定读取的表，使用 Pandas 读取 Excel 文件的代码如下：

```
file_name = 'num_excel.xlsx'
#可以通过 Sheet 名或者 Sheet 的索引进行访问('Sheet1' == 0,'Sheet2' == 1)
excel_file = pd.read_excel(file_name, 0)
print(excel_file)
```

3. 读取 JSON 文件

Pandas 同样可以读取 JSON 数据，与 3.8 节中将要提到的 JSON 模块不同，Pandas 读取的 JSON 数据仍然是 DataFrame 的形式，所读取的 JSON 文件可以分为 4 种格式(orient)，第 1 种是 split，其表示将 DataFrame 中的行号、列号及内容分开存储。JSON 中 index 的数据为 DataFrame 中的最左一列的行号索引，以 columns 的数据为 DataFrame 中的表头名，以 data 的数据为 DataFrame 中的内容；剩余 3 种分别是 index、records 及 table，这 3 种格式在此不进行讲解，详细内容可以参考 Pandas 官网。

新建一个 num_json.json 文件，其内容如图 3-18 所示。

```
{
    "index": [1, 2],
    "columns": ["first", "second", "third", "fourth", "fifth", "sixth"],
    "data": [
        [1, 2, 3, 4, 5, 6],
        [7, 8, 9, 10, 11, 12]
    ]
}
```

图 3-18　新建 JSON 数据示例

读取 num_json.json 文件中的数据的代码如下：

```
file_name = 'num_json.json'
#index ->[index], columns ->[columns], data ->[values]
json_file = pd.read_json(file_name, orient='split')
print(json_file)
```

运行程序后，可以得到如图 3-19 所示的结果。

```
   first  second  third  fourth  fifth  sixth
1      1       2      3       4      5      6
2      7       8      9      10     11     12
```

图 3-19　读取 JSON 文件

3.3.3　使用 Pandas 处理数据

使用 Pandas 读取数据后，可以使用 NumPy 中的方法处理 DataFrame 中的 values，也可以直接使用 Pandas 对 DataFrame 进行处理，本节将展示几种常见的数据处理方式。

1. 取行数据

本节将说明如何取出 DataFrame 中的一行。使用 DataFrame 的 loc 属性并传入行名称取出对应行，或者使用 DataFrame 的 iloc 属性，此时需要传入的是行索引以取出对应行。取出 csv_file 中的第 1 行(索引为 0)的代码如下：

```
print(csv_file.iloc[0])
```

运行以上程序后，可以得到如图 3-20 所示的结果。可以看出其确实取出了 DataFrame 中的第 1 行，并且其结果同时输出了列名、行名和数据类型等信息。

2. 取列数据

本节说明如何取出 DataFrame 中的一列，同样可以使用 loc 属性和 iloc 属性，此时对 loc 的索引格式为 loc[:, <column_name>]，需要给 loc 传入两个索引值，前一个表示行索引，后一个则表示列索引。除此之外，还可以直接使用 DataFrame[<column_name>]的形式取出列数据，这两种方法的代码如下：

```
#方法 1
print(csv_file['first'])
#方法 2
print(csv_file.loc[:, 'first'])
```

运行程序后，可以得到如图 3-21 所示的结果。

```
first     1
second    2
third     3
fourth    4
fifth     5
Name: 0, dtype: int64
```

图 3-20　取出 DataFrame 中的行数据

```
0    1
1    6
Name: first, dtype: int64
```

图 3-21　取出 DataFrame 中的列数据

3. 求数据的统计信息

使用 Pandas 可以方便地得到 DataFrame 的统计信息，如最大值、最小值、平均值等，下面的程序分别展示了如何取出 DataFrame 中行的最大值、DataFrame 中列的最小值及整个 DataFrame 中的平均值，代码如下：

```
#axis=1 表示对行进行操作
print(csv_file.max(axis=1))
#axis=0 表示对列进行操作，默认 axis 为 0
print(csv_file.min(axis=0))
#先取出列的平均值，接着求一次列均值的均值即为整个 DataFrame 的均值
print(csv_file.mean().mean())
```

```
0     5
1    10
dtype: int64
first      1
 second    2
 third     3
 fourth    4
 fifth     5
dtype: int64
5.5
```

图 3-22 DataFrame 中的统计信息

运行程序后可以得到如图 3-22 所示的结果，能看到 0 行和 1 行的最大值分别为 5 和 10，first 到 fifth 这 5 列的最小值分别为 1、2、3、4、5，而整个 DataFrame 的均值为 5.5。

4. 处理缺失值

在数据集中，常常存在缺失数据，对于缺失数据的处理通常有两种方法，一种是直接将含有缺失数据的记录删除；另一种是将特征值填入缺失位置，通常会在该位置填入该列的平均值或 0，或者直接填入字段以示此处空缺/无效。找到 DataFrame 中的缺失数据并进一步进行处理的代码如下：

```
//ch3/test_pandas.py
#插入一条所有数据为 NaN 的记录
csv_file_with_na = csv_file.reindex([0, 1, 2])
print(csv_file_with_na)
#查看 NaN 在 DataFrame 中的位置
print(csv_file_with_na.isna())
#使用每列的平均值填入该列所有 NaN 的位置
print(csv_file_with_na.fillna(csv_file_with_na.mean(axis=0)))
#在所有 NaN 的位置填入 0
print(csv_file_with_na.fillna(0))
#在所有 NaN 的位置填入"Missing"字段
print(csv_file_with_na.fillna('Missing'))
#丢弃 DataFrame 中含有 NaN 的行
print(csv_file_with_na.dropna(axis=0))
#丢弃 DataFrame 中含有 NaN 的列
print(csv_file_with_na.dropna(axis=1))
```

运行程序后，可以得到如图 3-23 所示的结果。从结果可以看出，使用 reindex 方法后，由于原 DataFrame 中没有索引为 2 的行，所以 Pandas 自动新建了一条所有字段都为 NaN 的记录，使用了 isna 方法查看 DataFrame 中所有 NaN 的位置，其返回一个 bool 类型的 DataFrame，可以看到除了索引为 2 的行都为 True 外，其他记录都为 False，说明只有刚才插入的记录是 NaN。接下来代码分别使用了 fillna 方法将各列均值、0 或 Missing 字段对缺

失值进行了填充，从图 3-23 中可以看出不同填充方案得到的结果。最后代码还展示了如何直接舍弃含有 NaN 的数据，一共有两种舍弃方式，即舍弃行或者舍弃列。从舍弃行的结果可以看出，索引为 2 的记录被直接删除，而舍弃列的结果返回了一个空的 DataFrame，因为每列都含有 NaN，因此所有的数据都被舍弃了。

```
   first   second   third   fourth   fifth
0    1.0      2.0     3.0      4.0     5.0
1    6.0      7.0     8.0      9.0    10.0
2    NaN      NaN     NaN      NaN     NaN
   first   second   third   fourth   fifth
0  False    False   False    False   False
1  False    False   False    False   False
2   True     True    True     True    True
   first   second   third   fourth   fifth
0    1.0      2.0     3.0      4.0     5.0
1    6.0      7.0     8.0      9.0    10.0
2    3.5      4.5     5.5      6.5     7.5
   first   second   third   fourth   fifth
0    1.0      2.0     3.0      4.0     5.0
1    6.0      7.0     8.0      9.0    10.0
2    0.0      0.0     0.0      0.0     0.0
     first   second    third   fourth    fifth
0        1        2        3        4        5
1        6        7        8        9       10
2  Missing  Missing  Missing  Missing  Missing
   first   second   third   fourth   fifth
0    1.0      2.0     3.0      4.0     5.0
1    6.0      7.0     8.0      9.0    10.0
Empty DataFrame
Columns: []
Index: [0, 1, 2]
```

图 3-23　处理 DataFrame 中的缺失值

本节说明了 Pandas 的基本用法，从各种类型数据的读取到数据的处理，相比本书介绍的部分，Pandas 还有许多数据处理与分析方法，更多信息可以查看 Pandas 的官网。

3.4　scikit-learn

scikit-learn 是一个开源的 Python 机器学习库，在开始说明具体任务之前，先向读者阐述 scikit-learn 中模型的建立与使用过程。在使用 scikit-learn 做机器学习的过程中，第 1 步是准备数据，准备好训练集数据与测试集数据；接着实例化一个模型的对象（模型即后文中将要介绍的 SVM、随机森林等），需要根据具体任务选择适合的模型；在实例化模型完成之后，直接调用 fit 函数即可，此函数需要传入训练数据，函数内部自动拟合所传入的数据。在 fit 完成后，若需要测试，则调用 predict 函数即可，接着可以使用 score 函数对预测值与真实值之间的差异进行评估，该函数会返回一个得分以表示差异的大小。

从上述使用框架的过程中可以看出，scikit-learn 是一个封装性很强的包，这对于新手而言十分友好，无须自己定义过多函数或写过多代码，直接调用其封装好的函数即可，而整个过程对用户形成了一个黑盒，使用户难以理解算法内部的具体实现，这也是封装过强的弊端。

下面本书就以回归及分类任务为例具体讲解框架中每步的做法。

3.4.1 使用 scikit-learn 进行回归

本节以简单的回归问题说明 scikit-learn 的用法，以 3.4 节中所讲的 5 个步骤依次进行介绍。

1. 准备数据

本节选取了一个图像较为复杂的函数 $y=x\sin(x)+0.1x^2\cos(x)+x$ ，使用以下程序生成 y 在 $x\in[-10, 10]$ 上的数据，并且 x 以间隔为 0.01 取值。为了验证 scikit-learn 中模型的学习能力，直接使用训练数据进行测试，代码如下：

```
//ch3/test_scikit_learn.py
from sklearn.svm import SVR, SVC
import numpy as np
import matplotlib.pyplot as plt

#生成回归任务的数据
def get_regression_data():
    start = -10
    end = 10
    space = 0.01

    #自变量从[start, end]中以 space 为等间距获取
    x = np.linspace(start, end, int((end - start) / space))
    #根据自变量计算因变量，并给其加上噪声干扰
    y = x * np.sin(x) + 0.1 * x ** 2 * np.cos(x) + x + 5 * np.random.randn(*x.shape)

    #返回训练数据
    return np.reshape(x, [-1, 1]), y

#得到回归数据
x, y = get_regression_data()
#打印数据形状以进行验证
print(x.shape, y.shape)
```

运行以上程序可以得到命令行的输出：(2000, 1) (2000,)，说明训练数据与测试数据已经被正确获取，其中训练数据 x 的形状中 2000 表示有 2000 个训练样本，1 表示每个训练样本由 1 个数构成，而训练标签 y 的形状表明其是由 2000 个数组成的标签。

接下来对训练数据进行可视化，让读者对数据有一个直观上的认识，同时也测试生成的数据是否符合要求。使用 Matplotlib 对数据进行可视化，以蓝色的点进行标识，实现可视化过程的代码如下：

```
//ch3/test_scikit_learn.py
#可视化数据
figure, axes = plt.subplots()

#以散点图绘制数据
```

```
axes.scatter(x, y, s = 1, label = 'training data')
#以 LaTeX 风格设置标题
axes.set_title('$y = x sin(x) + 0.1x^2 cos(x) + x$')
axes.legend()
axes.grid()
plt.show()
```

运行以上程序可以得到如图 3-24 所示的函数图像，从图中可以看出数据基本处于一条曲线，该曲线即上面设置的函数，说明训练与测试数据都被正确地生成了。

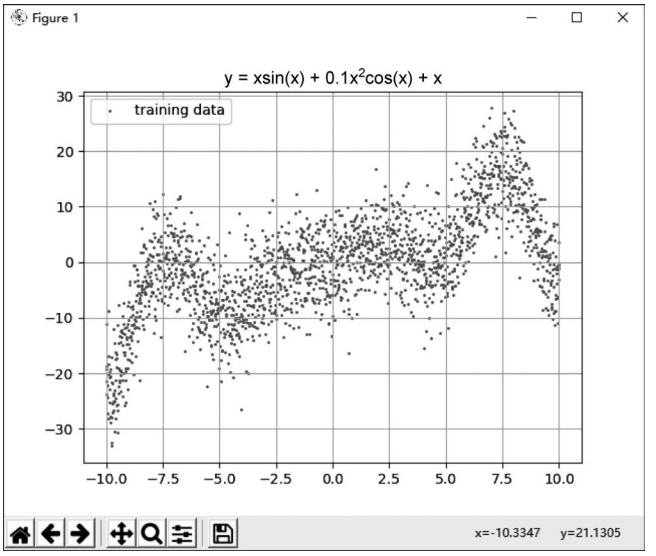


图 3-24　训练数据的图像

2. 实例化模型

本书在这一节选用 SVR(Support Vector Regression，支持向量回归)，将 SVM 运用到回归问题上。本节对于原理不做阐述，仅说明 scikit-learn 的用法。初始化 SVR 模型的代码如下：

```
#初始化分类模型
svr = SVR(kernel = 'rbf', C = 10)
```

上面的代码表示初始化了一个 SVR 模型，其 kernel 参数表示模型选用的核函数，scikit-learn 对 SVR 的核函数有以下常见 3 种选择：rbf、linear 和 poly，在此选用 rbf 作为核函数，C 表示误差的惩罚系数，惩罚系数越大，则对训练数据拟合越好，但有可能造成过拟合，其默认值为 1，由于训练数据较难拟合，因此本书将 C 值设置为 10，以加强模型的拟合能力，读者可以自行尝试其他值。

3. 使用模型进行拟合

定义好模型后，接下来使用模型拟合训练数据，代码如下：

```
#用模型对数据进行拟合
svr_fit = svr.fit(x, y)
```

使用 fit 函数对训练数据进行拟合，需要传入数据及其对应的标签。

4. 使用模型进行测试

在模型拟合完数据后，为了测试模型的性能，可以使用 predict 函数查看其对不同的输入值的预测，测试的代码如下：

```
#使用模型进行测试
svr_predict = svr_fit.predict(x)
```

在 predict 函数中只用传入训练数据，函数会将预测值返回，此时使用训练数据进行预测，这样方便在之后的可视化过程中查看模型预测与真实值之间的差异。绘制真实值与预测值的代码如下：

```
#可视化模型学到的曲线
fig, axes = plt.subplots()
axes.scatter(x, y, s = 1, label = 'training data')
axes.plot(x, svr_predict, lw = 2, label = 'rbf model', color = 'red')
axes.legend()
axes.grid()
plt.show()
```

可视化的结果如图 3-25 所示，其中深色的曲线是模型预测结果，从图中可以看出，该曲线和原离散数据呈现的图形很相似，说明模型对数据的拟合较好。

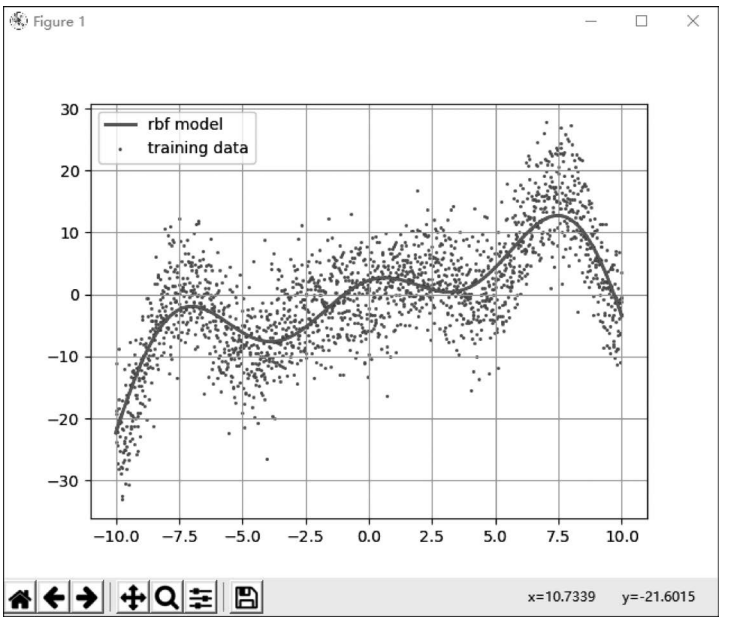


图 3-25　训练数据的图像

5. 评估模型性能

除了可以使用可视化的方式查看模型拟合情况，还可以使用 score 方法定量评估模型的好坏，score 函数定量地刻画了预测值与真实值之间的差异，其用法的代码如下：

```
#评估模型性能
score = svr_fit.score(x, y)
print(score)
```

score 方法需要传入训练数据及其标签，最终在命令行输出的 score 为 0.6428078808326549(读者的 score 和本书所展示的值可能不同，因为数据的初始化是随机的)，说明模型对于 64.28% 的数据预测正确。由于原数据在空间中较为离散化，过高的 score 可能会带来过拟合的问题，无论从可视化的结果还是 score 上来看，该模型的表现可以接受。

3.4.2　使用 scikit-learn 进行分类

本节同样以使用 scikit-learn 的 5 个步骤分别说明如何解决分类问题。

1. 准备数据

与回归的数据不同，分类需要给特定的数据指定类标签，为了简便起见，本节使用二维坐标作为分类特征，落在椭圆 $\frac{x^2}{1.5^2}+y^2=1$ 内的点为一类，类标签以 0 表示，而落在椭圆 $\frac{x^2}{1.5^2}+y^2=1$ 与圆 $x^2+y^2=4$ 之间的点为另一类，其类标签为 1。下面的程序用于生成分类的训练数据，代码如下：

```
//ch3/test_scikit_learn.py
#生成分类任务的数据
def get_classification_data():
    #数据量
    cnt_num = 1000
    #计数器
    num = 0

    #初始化数据与标签的占位符,其中训练数据为平面上的坐标,标签为类别号
    x = np.empty(shape=[cnt_num, 2])
    y = np.empty(shape=[cnt_num])

    while num < cnt_num:
        #生成随机的坐标值
        rand_x = np.random.rand() * 4 - 2
        rand_y = np.random.rand() * 4 - 2

        #非法数据,如果超出了圆 x^2 + y^2 = 4 的范围,则重新生成合法坐标
```

```
        while rand_x ** 2 + rand_y ** 2 > 4:
            rand_x = np.random.rand() * 4 - 2
            rand_y = np.random.rand() * 4 - 2

        #如果生成的坐标在椭圆 x^2 / 1.5^2 + y^2 = 1 的范围内,则类标号为 0,否则为 1
        if rand_x ** 2 / 1.5 ** 2 + rand_y ** 2 <= 1:
            label = 0
        else:
            label = 1

        #将坐标存入占位符
        x[num][0] = rand_x
        x[num][1] = rand_y

        #将标签存入占位符
        y[num] = label

        num += 1

    #给训练数据添加随机扰动以模拟真实数据
    x += 0.3 * np.random.randn( * x.shape)

    return x, y

#得到训练数据与标签
x, y = get_classification_data()
#查看数据和标签的形状
print(x.shape, y.shape)
```

运行上面的程序后,能看到命令行输出(1000, 2) (1000,),表示有 1000 个训练数据及其对应的标签,其中每个训练数据由两个数(坐标)构成,而标签由 1 个数字构成。

除此之外,使用 Matplotlib 以散点图的形式可视化训练数据,由于同时存在不同类别的数据,所以需要先将类标为 0 的数据和将类标为 1 的点分开,并以不同的标识绘制,这样会增强图像的直观性,代码如下:

```
//ch3/test_scikit_learn.py
#获取标签为 0 的数据下标
zero_cord = np.where(y == 0)
#获取标签为 1 的数据下标
one_cord = np.where(y == 1)

#以下标取出标签为 0 的训练数据
zero_class_cord = x[zero_cord]
#以下标取出标签为 1 的训练数据
one_class_cord = x[one_cord]

figure, axes = plt.subplots()
#以圆点画出标签为 0 的训练数据
```

```
axes.scatter(zero_class_cord[:, 0], zero_class_cord[:, 1], s=15, marker='o', label='class 0')
#以十字画出标签为1的训练数据
axes.scatter(one_class_cord[:, 0], one_class_cord[:, 1], s=15, marker='+', label='class 1')
axes.grid()
axes.legend()

#分别打印标签为0和1的训练数据的形状
print(zero_class_cord.shape, one_class_cord.shape)
plt.show()
```

运行以上程序，命令行会输出(388, 2) (612, 2)(读者输出的数据可能不同，因为数据的初始化是随机的)，表示在1000个训练样本中，有388个属于类0，有612个属于类1；同时能看到类似图3-26所示的结果。

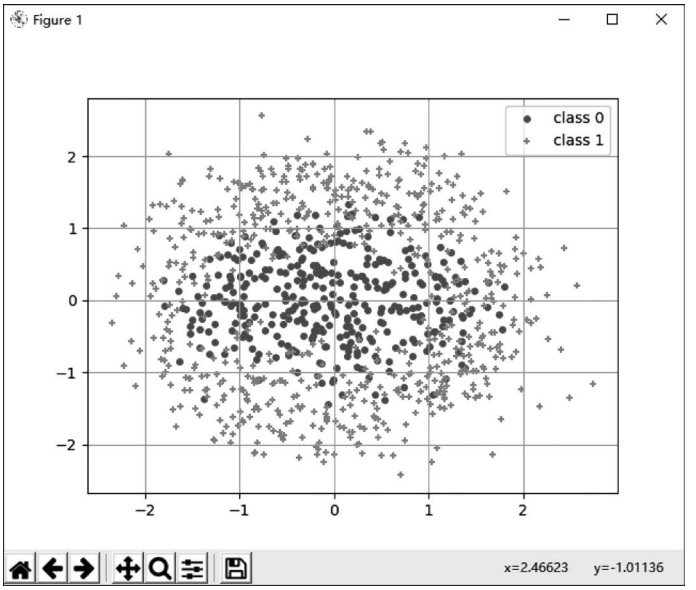


图 3-26　分类数据的散点图

2. 实例化模型

本节使用SVM完成对训练数据的分类，代码如下：

```
#创建SVM模型
clf = SVC(C=100)
```

3. 使用模型进行拟合

同样，类似3.4.1节中的第3部分，使用fit函数即能使模型拟合训练数据，代码如下：

```
clf.fit(x, y)
```

4. 使用模型进行测试

本节对分类器的分类边界进行可视化，由于变量(训练数据)可以充斥整个二维空间，因

此可以从二维空间中取出足够多的点以覆盖所关心的区域(由于需要将分类数据与分类边界相比较,所以可以将关心的区域设置为训练数据所覆盖的区域),使用得到的模型对关心的区域中的每个点进行分类,以得到其类别号,最终将不同预测的类别号以不同的颜色画出,即可得到模型的分类边界,代码如下:

```
//ch3/test_scikit_learn.py
def border_of_classifier(sklearn_cl, x):
    #求出所关心范围的最边界值: 最小的 x、最小的 y、最大的 x、最大的 y
    x_min, y_min = x.min(axis = 0) - 1
    x_max, y_max = x.max(axis = 0) + 1

    #将[x_min, x_max]和[y_min, y_max]这两个区间分成足够多的点(以 0.01 为间隔)
    x_values, y_values = np.meshgrid(np.arange(x_min, x_max, 0.01),
                    np.arange(y_min, y_max, 0.01))

    #将上一步分隔的 x 与 y 值使用 np.stack 两两组成一个坐标点,覆盖整个关心的区域
    mesh_grid = np.stack((x_values.ravel(), y_values.ravel()), axis = -1)

    #使用训练好的模型对于上一步得到的每个点进行分类,得到对应的分类结果
    mesh_output = sklearn_cl.predict(mesh_grid)

    #改变分类输出的形状,使其与坐标点的形状相同(颜色与坐标一一对应)
    mesh_output = mesh_output.reshape(x_values.shape)

    fig, axes = plt.subplots()

    #根据分类结果从 cmap 中选择颜色进行填充(为了使图像清晰,此处选用 binary 配色)
    axes.pcolormesh(x_values, y_values, mesh_output, cmap = 'binary')

    #将原始训练数据绘制出来
    axes.scatter(zero_class_cord[:, 0],
       zero_class_cord[:, 1], s = 15, marker = 'o', label = 'class 0')
    axes.scatter(one_class_cord[:, 0],
       one_class_cord[:, 1], s = 15, marker = '+', label = 'class 1')
    axes.legend()
    axes.grid()

    plt.show()

#绘制分类器的边界,传入已训练好的分类器,以及训练数据(为了得到关心的区域范围)
border_of_classifier(clf, x)
```

运行上面的程序可以得到类似图 3-27 所示的结果(由于训练数据的随机性,因此读者的模型的分类边界与图 3-27 可能会不完全一致)。

5. 评估模型性能

与 3.4.1 节第 5 部分类似,使用 score 函数评估模型即可,代码如下:

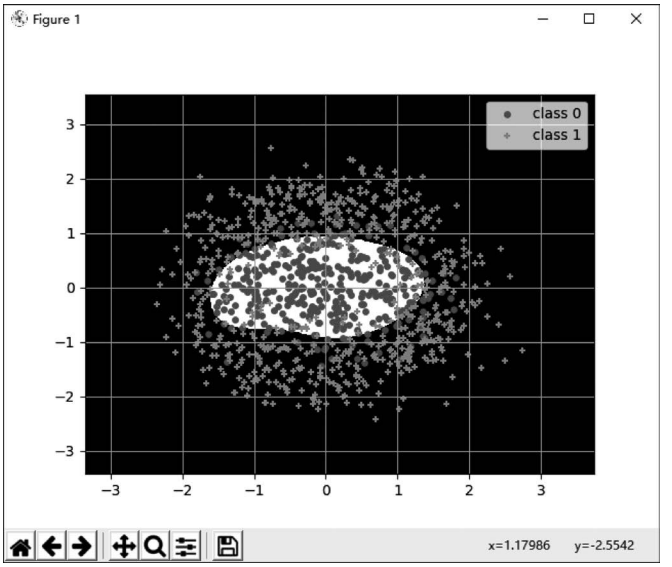


图 3-27　训练数据与模型分类边界

```
#评估模型性能
score = clf.score(x, y)
print(score)
```

运行以上代码，可以得到输出 0.859（读者得到的结果可能与此不同），说明分类器对85.9%的数据进行了正确分类，而从可视化结果可以看出来，剩余 14.1%未被正确分类的数据很有可能是噪声数据（那些存在于两类数据交叉部分的点）。基于这个准确率，可以接受此分类器。

本节以简单的回归与分类问题作为实例，讲解了 scikit-learn 的基本用法，还有许多别的模型和应用值得读者进一步探究，由于本书不涉及过多的机器学习知识，在此不进行讲解，更多信息可以参考 scikit-learn 官网。

3.5　collections

collections 模块在 Python 标准数据类型的基础上极大地扩展了一些特殊用途的数据类型，包括 namedtuple、deque、ChainMap、Counter、OrderedDict、defaultdict、UserDict、UserList 和 UserString，本节将对其中几种常用的数据类型进行介绍。

3.5.1　namedtuple

namedtuple 是 collections 中的一个重要的数据结构，从名称可以看出，它其实是一个元组类型的数据结构，但同时元组内的每个元素还有其对应的名称，这使它也具有类似字典的特性，如下代码展示了使用 namedtuple 表示四通道图像中每个通道值：

```
//ch3/test_collections.py
from collections import namedtuple

#创建一个表示四通道图像的 namedtuple
Channel = namedtuple('ImageChannels', field_names=['R', 'G', 'B', 'A'])
ch = Channel(R=127, G=200, B=255, A=100)
#获取四通道中的 R 通道值
print(ch[0], ch.R)
```

创建 namedtuple 时，第 1 个参数为类型名(typename)，类似于 class 的名称，表示所创建的 namedtuple 名称，第 2 个参数表示 namedtuple 中的属性名，可以传入一个 str 的列表或者用空格分隔的字符串，为了提高代码的可读性，在如上代码中采用了传入字符串列表的形式对 namedtuple 中的变量进行定义。从 namedtuple 中获取属性值则可以采取元组或者类似字典的形式，如上代码所示。

从以上示例可以看出，使用 namedtuple 相比使用元组而言更加灵活。获取元组中的元素只能使用下标的形式，这对于代码阅读而言十分不直观，同时由于字典类型无法 hash，无法被添加进入集合，因此使用 namedtuple 能很好地解决这些问题。

3.5.2 Counter

顾名思义，Counter 是一个计数器的数据类型，其能方便地辅助计数相关应用的实现，本质是 dict 的一个子类，使用 Counter 进行计数的时候，其会返回一个 dict，key 为元素，value 为该元素出现的次数，如下代码以不同的方式对字符串中的不同字符进行了计数：

```
//ch3/test_collections.py
from collections import Counter
#待计数的字符串
s = 'abbcdd'
#直接传入字符串进行计数
counter1 = Counter(s)
#传入列表进行计数
counter2 = Counter(list(s))
#传入元组进行计数
counter3 = Counter(tuple(s))
#传入字典进行计数
counter4 = Counter({'a': 1, 'b': 2, 'c': 1, 'd': 2})
print(counter1, counter2, counter3, counter4)
```

以上代码分别以直接传入字符串、列表/元组、字典计数器的方式创建了具有相同结果的 Counter。

3.5.3 OrderedDict

OrderedDict 可以使读取字典中数据的顺序与写入数据的顺序相同，使用方法的代码如下：

```
//ch3/test_collections.py
from collections import OrderedDict
#创建 OrderedDict 对象
od = OrderedDict()
#向 OrderedDict 中存放值
od['A'] = 'a'
od['B'] = 'b'
od['C'] = 'c'
#读取 OrderedDict 中的值
for k, v in od.items():
   print(k, v)
```

运行如上代码可以看出，打印出来的元素顺序与插入数据时的顺序一致。在 Python 3.5 及之前，标准数据类型 dict 的存储和数据插入顺序并不一致，因此在 Python 3.5 及其之前需要使用 OrderedDict 保证读取数据的有序性。从 Python 3.6 开始，标准数据类型 dict 的读取顺序和写入顺序已经保持一致，读者可以根据自身的应用场景使用字典的顺序特性。

3.5.4　defaultdict

使用 Python 标准数据类型 dict 时，如果访问的 key 不在字典中，则程序会抛出异常 KeyError，此时需要使用逻辑判断该 key 是否存在于字典或者使用字典的 get 方法为不存在的键赋予默认值，代码如下：

```
//ch3/test_collections.py
#创建只包含一个元素的字典
d = {'ip': '127.0.0.1'}
#当尝试访问一个字典中不存在的元素时会抛出 KeyError
#port = d['port']
#在使用之前需要判断字典中是否有键
port = d['port'] if 'port' in d else None
#使用 get 方法赋予默认值
port = d.get('port')
```

虽然通过以上两种方法能够避免程序出现异常，但是无疑都增加了编程的成本，此时可以使用 collections 模块中的 defaultdict 来创建带有默认值的字典，如果此时字典中不存在某键，则会直接返回默认值，如下代码为字典中不存在的键返回默认值 80：

```
//ch3/test_collections.py
from collections import defaultdict
#创建一个默认值为 80 的 defaultdict
dd = defaultdict(lambda: 80)
dd['ip'] = '127.0.0.1'
#返回 80
port = dd['port']
```

使用 defaultdict 一方面简化了开发者对于异常的处理，另一方面由于程序会默认返回

值而不抛出异常，因此使用不慎会为程序引入额外的调试成本，这种行为可能是开发人员不希望的，读者需要根据自身的需要酌情使用。

3.6 typing

由于 Python 是动态语言，其对于类型定义没有严格的检查，编程方式十分灵活，然而便捷的同时也带来了另一个问题，这便是在编程过程中代码提示较少，在阅读他人的代码或后期进行代码维护时会增加成本，因此，在 Python 3.5 及其以后的版本中，原生 Python 内置了 typing 模块，用于辅助 Python 编程时的类型检查，可以很方便地用于集成开发环境。

在 Python 中，对于参数或者变量的类型，使用“：[类型]”的形式编写，而函数返回值的类型则在函数签名后使用“-> [类型]”即可，代码如下：

```
//ch3/test_typing.py
a: str = 2
#期望接受一种类型为 str 的参数，没有返回值
def print_var(v: str) -> None:
    print(v)
```

从代码中可以看出，虽然变量 a 的值为整型值，但是在进行类型标注时可以任意进行标注，并且在编程过程中以开发人员标注的类型为准，除此之外以上代码还定义了一个期望接受一种类型为字符串的参数，并且由于函数体内只进行了打印操作，没有返回值，因此在函数头使用“-> None”进行表示。

在程序运行时，Python 并不强制开发人员标注函数和变量的类型，使用 typing 的主要作用：①类型检查，防止程序运行时出现参数或返回值类型不一致的情形；②作为开发文档的附加说明，方便调用。本节将分不同的数据类型（Python 标准数据类型、扩展类型等）为读者介绍使用 typing 的代码写法。

3.6.1 标准数据类型标识

Python 3 中有 6 种基本的数据类型，分别是 Number、String、List、Tuple、Set 和 Dictionary，在 Number 中又包含 int、float、bool 和 complex，下面就分别介绍这几种数据类型在 typing 中的使用方法。

对于 Number 和 String 类型的标识比较简单，直接使用类型即可，代码如下：

```
//ch3/test_typing.py
#整型变量
int_var: int = 1
#浮点型变量
float_var: float = 1.0
#布尔型变量
bool_var: bool = True
```

```
#复数型变量
complex_var: complex = 1 + 2j
#字符串变量
str_var: str = '1'
#一个整型变量和一个浮点型变量相加并返回
def func_with_type(i: int, f: float, b: bool, c: complex, s: str) -> float:
    return i + f
```

对于 Number 和 String 类型变量的标识只需读者理解类型标识的基本写法，并未用到 typing 模块，而在对 List、Tuple、Set 和 Dictionary 类型的变量进行标识时，则需要进一步标识出这些组合类型中数据的类型，例如对于 List 类型变量，需要使用如下代码进行标识：

```
//ch3/test_typing.py
from typing import List
#将标识元素为整型值的列表
list_var: List[int] = [1, 2, 3, 4]
```

可以看出，在进行类型定义时，需要在“[]”中表明列表中元素的类型。类似地，在对 Tuple、Set 及 Dictionary 进行类型定义时，可以使用的代码如下：

```
//ch3/test_typing.py
from typing import Tuple, Set, Dict
#含有 4 个不同类型元素的元组
tuple_var: Tuple[int, str, float, bool] = [1, '2', '3.0', False]
#元素为整型变量的集合
set_var: Set[int] = {1, 2, 3, 4}
#键为字符串且值为整型值的字典
dict_var: Dict[str, int] = {'1': 1, '2': 2, '3': 3}
```

在 Python 3.9 及其以后的版本中，Python 中标准的组合类型也开始支持“[]”进行类型标识，因此更加推荐使用以下方式进行类型标识：

```
//ch3/test_typing.py
#将标识元素为整型值的列表
list_var2: list[int] = [1, 2, 3, 4]
#含有 4 个不同类型元素的元组
tuple_var2: tuple[int, str, float, bool] = [1, '2', '3.0', False]
#元素为整型变量的集合
set_var2: set[int] = {1, 2, 3, 4}
#键为字符串且值为整型值的字典
dict_var2: dict[str, int] = {'1': 1, '2': 2, '3': 3}
```

当然，不同类型标识也可以进行组合使用，例如下面的代码标识了变量 combined_var 是一个内部元素同时包含 List、Tuple 和 Dictionary 的元组：

```
//ch3/test_typing.py
combined_var: tuple[list[int], tuple[int, str, float, bool], dict[str, int]]
```

不难看出，每次进行类型标识的时候都需要重新写一次类型的定义，因此 Python 提供

了类型别名的写法，如下代码所示，分别使用了 A、B、C 3 个变量代表不同的复合数据类型，这样做的好处是能增强代码的可读性，其次也能很方便地复用已经定义过的类型。

```
//ch3/test_typing.py
#自定义类型别名
A = list[int]
B = tuple[int, str, float, bool]
C = dict[str, int]
combined_var2: tuple[A, B, C] = ([1, ], (1, '2', 3., True), {'1': 1})
```

对于 List、Set 等可以存储不同类型变量的复合数据结构而言，则需要使用 Union 进行标识，从 Python 3.10 开始，Union 可以使用“|”进行标识，下面的代码表示变量为包含整型值或字符串的列表：

```
//ch3/test_typing.py
from typing import Union
int_str_var: list[Union[int, str]] = ['a', 2, 'b', 4]
#Python 3.10 及以后
int_str_var2: list[int | str] = ['a', 2, 'b', 4]
```

3.6.2 collections 中的数据类型标识

类似地，typing 模块也支持对 collections 中的数据类型进行标识，下面将分别进行介绍。在 3.5 节中已经向读者介绍了 collections 模块中常用的几种数据结构，本节就以 3.5 节中的数据结构为例说明 collections 中的数据类型标识。

使用 typing 中的 NamedTuple 为 collections 中的 namedtuple 进行类型标识的时候并不用在注解内，而是使用如下代码进行声明：

```
//ch3/test_typing.py
from typing import NamedTuple
#相当于 collections.namedtuple('Address', ['ip', 'port])
class Address(NamedTuple):
    ip: str
    port: int
address = Address(ip = '127.0.0.1', port = 80)
```

可以看到，在 typing 中对于 namedtuple 是以类继承的形式进行实现的，在创建 namedtuple 实例时使用类似创建类对象的方法即可。

对于 Counter、OrderedDict 和 defaultdict 则还是以注解的形式进行类型标识即可，代码如下：

```
//ch3/test_typing.py
from typing import Counter as TCnt, OrderedDict as TOrdD, DefaultDict as TDD
from collections import Counter, OrderedDict, defaultdict
#由于 Counter 的 value 必定为 int, 因此只需标识 key 的类型
```

```
counter: TCnt[str] = Counter('aabbccddefg')
#OrderedDict需要标识key和value的类型
od: TOrdD[str, str] = OrderedDict()
#defaultdict需要标识key和value的类型
dd: TDD[str, str] = defaultdict(str)
```

对于Counter、OrderedDict和defaultdict的类型标识与Python中的标准数据类型标识方法类似，因此在此不再说明。

在Python 3.9及其之后的版本中，collections中的Counter、OrderedDict和defaultdict已原生支持，代码如下：

```
//ch3/test_typing.py
#Python 3.9之后
counter2: Counter[str] = Counter('aabbccddefg')
od2: OrderedDict[str, str] = OrderedDict()
dd2: defaultdict[str, str] = defaultdict()
```

3.6.3　其他常用标识

本节将为读者介绍一些在typing模块中常用的类型标识。

Callable表示当前的变量或参数是一个可调用对象，例如在以下的代码中函数期望传入一个函数对象：

```
//ch3/test_typing.py
from typing import Callable
#传入的函数参数接收一个整型值作为参数并且返回值为str
def wrapper1(func: Callable[[int], str]):
    return func(0)
#传入的函数参数接收任意的可变参数，无返回值
def wrapper2(func: Callable[..., None]):
    func()
```

Callable使用列表的形式接收函数的入参类型，并在其后跟随返回值类型。

当传入的参数类型可以为任意时，可以使用Any进行标识，在Python中无法进行类型推断的变量同样也被默认认为是Any类型，在此不使用代码进行说明。

当一个函数从不终止或总会抛出异常时，可以使用NoReturn进行标识，代码如下：

```
//ch3/test_typing.py
from typing import NoReturn
#包含死循环的函数
def func_while() -> NoReturn:
    from time import sleep
    while True:
        sleep(1)
#必定会抛出异常的函数
def func_exc(num: int) -> NoReturn:
    raise ValueError(f'Bad Value: {num}')
```

读者需要区分返回值为 NoReturn 和 None 的区别，NoReturn 表示函数不会终止或者不会返回，而 None 表示函数无返回值。

Optional 表示当前的类型标识是可选的，代码如下：

```
//ch3/test_typing.py
from typing import Optional
#期望传入一种类型为整型或浮点型的参数，允许传入 None
def func_with_optional_param(num: Optional[Union[int, float]]):
    if num is None:
        raise ValueError(f'Unexpected value: {num}')
    print(num)
```

Literal 表示变量或参数只能为指定的字面值，如下代码指定了打开文件的模式只能采用['r', 'rb', 'w', 'wb']中的值：

```
//ch3/test_typing.py
from typing import Literal
MODE = Literal['r', 'rb', 'w', 'wb']
#打开文件
def open_helper(file: str, mode: MODE) -> str:
    with open(file, mode, encoding = 'utf8'):
        pass
    return ''
```

3.7 argparse

在 Python 中，可以使用 argparse 模块方便地对命令行参数进行处理。编程时，通过命令行传入的不同参数改变程序的执行逻辑，极大地增加了程序的灵活性，因此本节将对 argparse 模块进行简要介绍。

3.7.1 argparse 的使用框架

argparse 模块能处理指定的命令行参数与位置命令行参数(根据传入参数的位置进行识别)，其整体使用流程如下：

首先使用 ArgumentParser 方法创建一个解析器 parser(此时 parser 中的参数列表为空)，在创建过程中可以为 ArgumentParser 方法传入定制化的属性，如对该 parser 的描述等。

创建完 parser 后，接下来需要为 parser 添加参数，一般使用 add_argument 方法，该方法需要指定参数名，同时 add_argument 方法含有许多可选的属性，如参数目标数据类型、默认值，目标参数指定动作 action 等。读者可以将一个添加完参数的 parser 理解成一个参数的集合，该集合包含了所有即将从命令行接受的参数。对于 add_argument 方法的探究是本节的重点，将在 3.7.2 节详细说明。

为 parser 添加完目标参数后，使用 parse_args 方法将命令行中传入的参数转换为 argparse 中的 Namespace 对象(表示参数读取完毕)，此时可以使用 args.[变量名]的形式访问由命令行传入的参数。

通过以上 3 个步骤，即可方便地解析来自命令行的参数。值得注意的是，使用 add_argument 方法为 parser 添加参数时，如何正确地设计参数的类型、属性及动作是至关重要的，有时错误的设计会给编码带来不小的麻烦。

3.7.2　使用 argparse 解析命令行参数

本节只对 add_argument 方法进行探讨，若读者想进一步学习 ArgumentParser 方法的运用，则可以参考 argparse 的帮助文档。

1. 解析字符串类型的参数

parser 从命令行接受的参数的默认类型是字符串，因此直接按照 3.7.1 节所讲的使用流程接受参数即可，下面的程序说明了这一过程，为创建的 parser 添加了一个名为 vvv(--vvv)的参数，并且此参数的简写形式为 v(-v)，默认值为 string，代码如下：

```
//ch3/test_argparse.py
import argparse

def parse_str():
    #创建 parser
    parser = argparse.ArgumentParser()
    #为 parser 添加一个名为 vvv(简称 v)的参数,其默认值为 string
    parser.add_argument('-v', '--vvv', default='string')
    #解析参数
    args = parser.parse_args()
    return args

args = parse_str()
#打印 Namespace
print(args)
#打印接受的参数及其类型
print(args.vvv, type(args.vvv))
```

在控制台使用命令 python test_argparse.py -v argparse_is_good(或 python test_argparse.py --vvv argparse_is_good)可以得到如图 3-28(a)所示的结果，能看到 args 中只有一个名为 vvv 的参数，它的值恰好是从命令行传入的"argparse_is_good"字符串。同时该参数类型为 str。如果直接使用命令 python test_argparse.py(不传入任何参数)，则会得到如图 3-28(b)所示的结果，可以看到此时 vvv 参数的值为默认值 string。

```
Namespace(vvv='argparse_is_good')
argparse_is_good <class 'str'>
```

(a) 从命令行传入参数

```
Namespace(vvv='string')
string <class 'str'>
```

(b) 使用默认值

图 3-28　使用 argparse 解析字符串参数

2. 解析 int 类型的参数

与 3.7.2 节的第 1 部分类似，在使用 add_argument 时，为 type 参数传入目标类型即可，下面的程序说明了如何将 type 指定为 int，以解析整型参数，代码如下：

```
#为 parser 添加一个名为 iii(简称 i)的参数,其默认值为 0,将传入的类型限制为整型
parser.add_argument('-i', '--iii', default=0, type=int)
```

将 type 指定为 int 后，程序会尝试将从命令行传入的参数转换为 int 型，如果转换失败(如使用命令 python test_argparse.py -i argparse_is_good)，则程序会报错终止。有意思的是，default 值的类型可以与指定的类型无关，因为只有当命令行未传入参数时，才会使用 default 值(尝试将 default 值转换为 type 类型)，因此，如果将上述程序的 default 改为 string，而使用正确传入整型数的命令时，则程序仍能正常执行。除了可以使用 int 作为 type 外，对于 float 也采用类似的处理方式，而 type 为 bool 则采用其他的处理方法，有兴趣的读者可以自行尝试将 type 指定为 bool 的解析结果(因为本质是将字符转换为 bool 值，因此会发现无论传入什么值其结果都为 True，除非将 default 置为 False 并不传入任何参数)。

3. 解析 bool 类型的参数

由于 bool 仅有两个值，即 True 或 False，因此 argparse 处理 bool 的过程中不需要传入任何值，仅以是否写出该参数进行判别。例如定义了一个名为 b 的参数，仅当命令中写出了参数 b 时，该值才为 True(或 False)，当没写时为 False(或 True)，而无须显式地传入 True 或 False。这一点和使用 if 语句判别 bool 值十分相似，如下面的程序，使用 b==True 进行判断是多此一举的，直接使用 b 本身即可，代码如下：

```
b = True
if b == True:
    …
if b:
    …
```

对于 bool 型参数的处理，需要用到 add_argument 中的 action 参数，将其指定为 store_true(或 store_false)表示命令中写了该参数就将其置为 True(False)，解析 bool 型的参数的代码如下：

```
#添加一个名为 bbb(简称 b)的参数,其默认值为 False,若命令写出 --bbb(-b),则值为 True
parser.add_argument('-b', '--bbb', default=False, action='store_true')
#添加一个名为 ppp(简称 p)的参数,其默认值为 True,若命令写出 --ppp(-p),则值为 False
parser.add_argument('-p', '--ppp', default=True, action='store_false')
```

4. 解析 list 类型参数

将命令行传入的参数返回为一个 list 有多种方法，本节介绍其中常用的两种。下面就分别对这两种方法进行介绍。

第 1 种是将 add_argument 方法的 action 指定为 append(列表的追加)，这种用法适合

命令中多次重复使用相同参数传值的情况。假设现已为 parser 添加了名为 eee 的参数并将 action 指定为 append，此时使用命令 python test_argparse.py --eee 1 --eee 2 则会得到参数 eee 为['1', '2']，这种方法的缺点是需要多次传入同名参数，不方便使用。

第 2 种更为便捷的方法是指定 add_argument 方法中的 nargs 参数，将这个参数指定为“+”“?”或“*”，分别表示传入 1 个或多个参数、0 个或 1 个参数及 0 个或多个参数（同正则表达式的规则一致），并将传入的参数转换为 list。例如将 nargs 指定为“+”并且变量名为 eee 的整型变量时，使用 python test_argparse.py --eee 1 2 后，直接可以得到名为 eee 值为[1, 2]的参数。

下面的程序分别说明了以上两种解析列表参数的方法，第 1 种方法得到的结果为['1', '2']，而第 2 种方法将 type 指定为 int，结果为[1, 2]，代码如下：

```
#添加一个名为 eee(简称 e)的参数,若多次使用 -- eee( - e),则结果以列表的 append 形式连接
parser.add_argument('-e', '--eee', action='append')
#添加一个名为 lll(简称 l)的参数,将传入的参数返回为一个 list
parser.add_argument('-l', '--lll', nargs='+', type=int)
```

argparse 还有更多高级用法，如打开指定文件等。

3.8　JSON

JSON 的全称为 JavaScript Object Notation，是一种轻量级的数据交换格式，其使用键-值对的形式存储与交换数据（与 Python 中的字典相同，不过 Python 中的字符串可以使用单引号或双引号表示，而 JSON 中仅能使用双引号），其键是无序的，仅支持由键访问数据，而其值是可以有序的，使用有序列表（数组）进行存储。

在 Python 中，使用 JSON 模块可以轻松地完成 JSON 数据的存储与读取。在 Python 中，JSON 支持直接以 JSON 格式处理 Python 字典，也支持处理类 JSON 格式的字符串。值得注意的一点是，使用 JSON 持久化字典数据时，仅支持 Python 中的内置数据类型，除此以外的类型需要进行转换，如键-值对中存在 NumPy 中的数据类型（常常会持久化 NumPy 数组），需要先将其转换为 Python 中的基本类型才能继续持久化。下面以 JSON 数据的写入与读取来分别介绍这两种处理方式。

3.8.1　使用 JSON 模块写入数据

在 JSON 中，写入数据使用 dump 方法，需要为其传入待存储的字典数据及对应的文件指针。除此之外，可以使用 dumps（dump＋string）方法将 Python 字典数据转换为字符串，dump 与 dumps 用法的代码如下：

```
//ch3/test_json.py
import json
```

```
#初始化 Python 字典
py_dict = {'message': 'json is brilliant!', 'version': 1.14, 'info': 'python dict'}

#使用 dump 方法向文件写入 Python 字典
with open('py_dict.json', 'w', encoding = 'utf8') as f:
    json.dump(py_dict, f)

#使用 dumps(dump + string)将字典值转换为对应字符串
dict2str = json.dumps(py_dict)
print(dict2str)
```

运行以上程序后，能发现代码目录下多了一个 py_dict.json 文件，其内容即定义的 py_dict 字典中的值，不同的是，在持久化为 JSON 文件时会将原字典中的格式自动重整为 JSON 的标准格式。与此同时，控制台打印的 dict2str 结果也正是 py_dict 转换为 JSON 格式字符串的结果。

3.8.2 使用 JSON 模块读取数据

本节将说明如何读取 JSON 文件。与持久化数据时所用的 dump 与 dumps 这一对“孪生兄弟”类似，读取 JSON 文件时也有对应的 load 与 loads(load+string)方法：load 方法从 JSON 文件中将持久化的内容读取到 Python 字典中，而 loads 则直接从类 JSON 字符串中获取数据，这两种数据读取的方法的代码如下：

```
//ch3/test_json.py
#打开并读取 JSON 文件
with open('py_dict.json', 'r', encoding = 'utf8') as f:
    load_json_file = json.load(f)

#初始化一个 JSON 格式的字符串
json_like_str = r'{"message": "json is brilliant!", "version": 1.14, "info": "json - like
string"}'
#从字符串中读取数据
load_json_str = json.loads(json_like_str)

#打印从文件中读取的数据
print(load_json_file)
#打印从字符串读取的数据
print(load_json_str)
```

运行程序后，能看到控制台分别打印出来自文件与字符串的内容，并且它们都是 Python 中的字典类型，说明读取的内容已经从字符串正确加载并转换为字典类型。

3.9 TA-Lib

TA-Lib 的全称为 Technical Analysis Library，从其名称可以看出这是一个“技术分析”库，其中包含了像是 ADX、MACD、RSI 这种技术指标，还包括 K 线信号的模式识别，下面将

分别对技术指标和模式识别进行说明。

3.9.1　技术指标

本节将为读者简单地介绍几种 TA-Lib 中常用的技术指标及其使用方法。首先在路径 code/ch3/下准备 CSV 数据 202001.csv，其数据内容如图 3-29 所示，可以看出数据包含以日为单位的基金数据（单位净值、日增长率、复权净值、复权净值增长率、累计净值、累计收益率、同类型排名、总排名、同类型排名百分比）。

```
datetime,unit_val,unit_val_growth_rate,adjust_val,adjust_val_growth_rate,cum_val,cum_val_growth_rate,same_type_rank,total_rank,same_type_rank_ratio
2001-09-28T00:00:00.000Z,1,NaN,1,NaN,0,-1,9999999999,9999999999,1.01
2001-09-29T00:00:00.000Z,1,0,1,0,0,-1,9999999999,9999999999,1.01
2001-10-19T00:00:00.000Z,0.9997,-0.0003,0.9997,-0.0003,0,-1,9999999999,9999999999,1.01
2001-10-26T00:00:00.000Z,1.0013,0.0016,1.0013,0.00160048,0,-1,9999999999,9999999999,1.01
2001-11-02T00:00:00.000Z,1.0013,NaN,1.0013,0,0,-1,9999999999,9999999999,1.01
2001-11-09T00:00:00.000Z,0.9989,-0.002397,0.9989,-0.00239688,0,-1,9999999999,9999999999,1.01
2001-11-16T00:00:00.000Z,1.0079,0.00901,1.0079,0.00900991,0,-1,9999999999,9999999999,1.01
2001-11-22T00:00:00.000Z,1.0146,0.006647,1.0146,0.00664748,0,-1,9999999999,9999999999,1.01
2001-11-29T00:00:00.000Z,1.0145,-0.000099,1.0145,-0.00009856,0,-1,9999999999,9999999999,1.01
2001-12-07T00:00:00.000Z,1.0191,0.004534,1.0191,0.00453425,0,-1,9999999999,9999999999,1.01
2001-12-10T00:00:00.000Z,1.0174,-0.001668,1.0174,-0.00166814,0,-1,9999999999,9999999999,1.01
2001-12-11T00:00:00.000Z,1.0092,-0.00806,1.0092,-0.00805976,0,-1,9999999999,9999999999,1.01
2001-12-12T00:00:00.000Z,1.0068,-0.002378,1.0068,-0.00237812,0,-1,9999999999,9999999999,1.01
2001-12-13T00:00:00.000Z,1.0023,-0.00447,1.0023,-0.00446961,0,-1,9999999999,9999999999,1.01
2001-12-14T00:00:00.000Z,1.0004,-0.001896,1.0004,-0.00189564,0,-1,9999999999,9999999999,1.01
2001-12-17T00:00:00.000Z,0.9981,-0.002299,0.9981,-0.00229908,0,-1,9999999999,9999999999,1.01
2001-12-18T00:00:00.000Z,1.0027,0.004609,1.0027,0.00460876,0,-1,9999999999,9999999999,1.01
2001-12-19T00:00:00.000Z,1.0001,-0.002593,1.0001,-0.002593,0,-1,9999999999,9999999999,1.01
2001-12-20T00:00:00.000Z,0.9928,-0.007299,0.9928,-0.00729927,0,-1,9999999999,9999999999,1.01
2001-12-21T00:00:00.000Z,0.9942,0.00141,0.9942,0.00141015,0,-1,9999999999,9999999999,1.01
2001-12-24T00:00:00.000Z,0.9888,-0.005432,0.9888,-0.0054315,0,-1,9999999999,9999999999,1.01
2001-12-25T00:00:00.000Z,0.9939,0.005158,0.9939,0.00515777,0,-1,9999999999,9999999999,1.01
2001-12-26T00:00:00.000Z,0.9984,0.004528,0.9984,0.00452762,0,-1,9999999999,9999999999,1.01
2001-12-27T00:00:00.000Z,0.998,-0.000401,0.998,-0.00040064,0,-1,9999999999,9999999999,1.01
2001-12-28T00:00:00.000Z,1.0011,0.003106,1.0011,0.00310621,0,-1,9999999999,9999999999,1.01
2001-12-31T00:00:00.000Z,1.0037,0.002597,1.0037,0.00259714,0,-1,9999999999,9999999999,1.01
2002-01-04T00:00:00.000Z,0.9961,-0.007572,0.9961,-0.00757198,0,-1,9999999999,9999999999,1.01
2002-01-07T00:00:00.000Z,0.9933,-0.002811,0.9933,-0.00281096,0,-1,9999999999,9999999999,1.01
2002-01-08T00:00:00.000Z,0.9926,-0.000705,0.9926,-0.00070472,0,-1,9999999999,9999999999,1.01
```

图 3-29　202001.csv 中的数据示例

使用 Pandas 模块读取 CSV 数据，代码如下：

```
//ch3/test_talib.py
import pandas as pd

pd.set_option('display.max_rows', None)

#读取 CSV 数据
data = pd.read_csv('202001.csv')
print(data)
#获取复权净值
 adjust_val = data.loc[:, 'adjust_val']
```

接下来使用最常用的指标 SMA（Simple Moving Average，简单移动平均），其计算如式（3-1）所示。

$$\mathrm{SMA}_i^k = \frac{1}{k}\sum_{j=i-k+1}^{i} x_j \tag{3-1}$$

其中，下标 i 表示第 i 个简单移动平均值，上标 k 表示周期为 k，因此 SMA 用于简单计算原数据中包含第 i 个元素在内的前 k 个元素的平均值，使用 TA-Lib 计算 SMA 的方法，代码如下：

```
//ch3/test_talib.py
import talib
#计算周期为5天的复权净值的移动平均值
sma = talib.SMA(adjust_val, timeperiod=5)
print(sma)
```

```
0            NaN
1            NaN
2            NaN
3            NaN
4       1.000460
          ...
5326    7.232026
5327    7.230002
5328    7.223385
5329    7.222296
5330    7.226344
Length: 5331, dtype: float64
```

图 3-30 计算复权净值 SMA 值的结果

运行以上代码，可以得到收盘价的 SMA 值的计算结果，如图 3-30 所示。

从图 3-30 中可以看出，前 4 个 SMA 值为 NaN，这是因为代码中计算 SMA 的周期值为 5，因此在计算前 4 个复权净值的 SMA 时数据不足，因此返回值为 NaN，而第 5 个值的计算方式为$\frac{1}{5}(1+1+0.9997+1.0013+1.0013)=1.00046$，其他的 SMA 值的计算以此类推。

接下来再介绍一个常用的指标 MACD(Moving Average Convergence/Divergence，异同移动平均线)，其需要计算一个快速平均线(指数平均线，典型周期为 12)和一个慢速平均线(指数平均线，典型周期为 26)，并计算两者之间的差值作为信号的依据，平均线的计算方法如式(3-2)所示。

$$\begin{aligned} \mathrm{EMA}_m^{\mathrm{fast}} &= \mathrm{EMA}_{m-1}^{\mathrm{fast}} \times \frac{M-1}{M} + x_m \times \frac{1}{M} \\ \mathrm{EMA}_n^{\mathrm{slow}} &= \mathrm{EMA}_{n-1}^{\mathrm{slow}} \times \frac{N-1}{N} + x_n \times \frac{1}{N} \end{aligned} \tag{3-2}$$

式中的 M、N 表示计算 EMA 的周期，m、n 表示序列中第 m、第 n 个元素的 EMA 值。在 TA-Lib 中，如果计算 EMA 的所需元素不够，则其直接使用算术平均值代替计算。得到快速和慢速平均线后，使用式(3-3)计算 DIF 值：

$$\mathrm{DIF}_m = \mathrm{EMA}_m^{\mathrm{fast}} - \mathrm{EMA}_m^{\mathrm{slow}} \tag{3-3}$$

对 DIF 同样使用 EMA 计算指数移动平均值即可得到 DEA(MACD 值)，如式(3-4)所示。

$$\mathrm{DEA}_n = \mathrm{DEA}_{n-1} \times \frac{N-1}{N} + \mathrm{DIF}_m \times \frac{1}{N} \tag{3-4}$$

分别得到 DIF 和 DEA 的值之后，再使用式(3-5)计算 MACD 值：

$$\mathrm{MACD} = \mathrm{DIF}_m - \mathrm{DEA}_n \tag{3-5}$$

得到的 MACD 值即为行情软件中 MACD 指标的红/绿柱所表示的值。在 TA-Lib 中计算 MACD 值的方法，代码如下：

```
//ch3/test_talib.py
#使用定义计算MACD
ema_fast = talib.EMA(adjust_val, timeperiod=3)
ema_slow = talib.EMA(adjust_val, timeperiod=5)
dif = ema_fast - ema_slow
dea = talib.EMA(dif, timeperiod=2)
```

```
macd_hist = (dif - dea)
print(macd_hist)

#直接使用 talib 计算 MACD
macd, macd_signal, macd_hist = \
    talib.MACD(adjust_val, fastperiod=3, slowperiod=5, signalperiod=2)
print(macd_hist)
```

如上代码中展示了如何使用定义与 TA-Lib 计算 MACD 值，运行代码可以发现两者的计算结果一致。

3.9.2　模式识别

TA-Lib 中除了能计算各指标值以外，还能针对 K 线的排列进行模式识别，例如乌云盖顶、三只乌鸦等 K 线形态。由于基金数据不符合 OHLC 的格式，因此 TA-Lib 中的模式识别不适用于基金数据。例如需要识别 OHLC 数据中的“乌云盖顶”模式，可以使用以下代码实现：

```
res = talib.CDLDARKCLOUDCOVER(opens, highs, lows, closes, penetration=0.5)
```

更多有关 TA-Lib 模式识别的内容可以参见其文档。

3.10　AKShare

AKShare 是一个功能十分强大的财经数据获取开源 Python 包，它能够提供股票、期货、债券、期权、外汇、货币、现货、利率、基金、指数等数据，股票包括 A 股、港股和美股等数据，提供实时与历史行情数据，同时针对股票还提供市场的评价信息和年报等基本面数据。对于期货、期权和外汇等品种，AKShare 也提供了类似的数据。

AKShare 同时提供了许多有趣的数据，包括中国宏观杠杆率、CPI 和 PPI 报告等宏观数据、不同国家的宏观数据、奥运奖牌、空气质量等。使用者可以将 AKShare 作为一个大的数据集市，其中不仅提供了金融数据，也能为其他领域提供数据分析应用的数据。

AKShare 从相对权威的财经数据网站获取原始数据并进行加工并返回，使用 AKShare 时的 Python 版本最好在 3.8.5 以上。由于原始的财经网站数据格式与接口可能经常发生变化，因此推荐将 AKShare 升级到最新版本进行使用。

下面以公募基金相关数据获取为例说明 AKShare 的使用方法。

3.10.1　获取基金基础信息

在 AKShare 中，获取基金的基础信息使用 fund_name_em 方法，这种方法从天天基金网获取基金的基础信息并返回一个 5 列的 DataFrame，每列信息分别为基金代码、拼音缩写、基金简称、基金类型和拼音全称，其中基金代码和基金类型是最重要的信息，基金的简称

与拼音等对实际的分析意义不大。

使用如下的代码完成基金基础信息的获取：

```
//ch3/test_akshare.py
import akshare as ak

#获取当前时刻所有基金的基础数据
fund_infos = ak.fund_name_em()
print(fund_infos)
```

执行代码后，可以得到如图 3-31 所示的数据。

	基金代码	拼音缩写	基金简称	基金类型	拼音全称
0	000001	HXCZHH	华夏成长混合	混合型-灵活	HUAXIACHENGZHANGHUNHE
1	000002	HXCZHH	华夏成长混合(后端)	混合型-灵活	HUAXIACHENGZHANGHUNHE
2	000003	ZHKZZZQA	中海可转债债券A	债券型-可转债	ZHONGHAIKEZHUANZHAIZHAIQUANA
3	000004	ZHKZZZQC	中海可转债债券C	债券型-可转债	ZHONGHAIKEZHUANZHAIZHAIQUANC
4	000005	JSZQXYDQZQ	嘉实增强信用定期债券	债券型-长债	JIASHIZENGQIANGXINYONGDINGQIZHAIQUAN
...	...	...	...	...	...
19738	970204	XZZGJQLXXZLLGYCYQZQA	兴证资管金麒麟兴享增利六个月持有期债券A	债券型-混合债	XINGZHENGZIGUANJINQILINXINGXIANGZENGLILIUGEYUE...
19739	970205	XZZGJQLXXZLLGYCYQZQC	兴证资管金麒麟兴享增利六个月持有期债券C	债券型-混合债	XINGZHENGZIGUANJINQILINXINGXIANGZENGLILIUGEYUE...
19740	970206	ZJYSLHYNCYHHC	中金优势领航一年持有混合C	混合型-偏股	ZHONGJINYOUSHILINGHANGYINIANCHIYOUHUNHEC
19741	970207	GXRFZQC	国信睿丰债券C	债券型-混合债	GUOXINRUIFENGZHAIQUANC
19742	970208	GXJDZHSGYCYHHFOF	国信经典组合三个月持有混合(FOF)	FOF	GUOXINJINGDIANZUHESANGEYUECHIYOUHUNHEFOF

[19743 rows x 5 columns]

图 3-31　使用 AKShare 获取基金基本信息的结果

从图 3-31 可以看出本书执行代码的时候一共获取了 19 743 只不同类型的基金。

3.10.2　获取基金历史行情

不同类型的基金在数据组织结构上存在不同，本节以开放式基金为例讲解如何获取基金历史行情。在 AKShare 中，使用 fund_open_fund_info_em 方法获取开放式基金的历史行情，这种方法接收两个参数，分别是基金代码 fund 与指标名称 indicator，其中 indicator 的可选值为下列值之一：单位净值走势、累计净值走势、累计收益率走势、同类排名走势、同类排名百分比、分红送配详情、拆分详情，不同的指标值的返回值字段有所不同，下面将以代码为 202001 的基金为例进行讲解。

1. 获取单位净值走势

使用 fund_open_fund_info_em(fund, indicator='单位净值走势')获取开放式基金的单位净值走势，可以得到如图 3-32 所示的结果。可以看出返回的 DataFrame 数据不仅包括每日的净值，还包括日增长率的数据。

日增长率的计算方式如式(3-6)所示，读者可以自行进行验算。

$$\text{growth\_rate}_i = \frac{\text{unit\_val}_i - \text{unit\_val}_{i-1}}{\text{unit\_val}_{i-1}} \times 100 \tag{3-6}$$

通过 fund_open_fund_info_em 方法得到的数据大多是以包含日期的及其相应指标数据的 DataFrame。

2. 获取累计净值走势

使用 fund_open_fund_info_em(fund, indicator='累计净值走势')获取开放式基金的累

计净值走势，返回的数据如图 3-33 所示，累计净值走势的数据只有两列，分别为日期及当日该基金的累计净值。

	净值日期	单位净值	日增长率
0	2001-09-28	1.0000	0.00
1	2001-10-12	1.0005	0.05
2	2001-10-19	0.9997	-0.08
3	2001-10-26	1.0013	0.16
4	2001-11-02	1.0013	0.00
...	...	...	...
5338	2023-10-11	1.8347	0.24
5339	2023-10-12	1.8352	0.03
5340	2023-10-13	1.8202	-0.82
5341	2023-10-16	1.8132	-0.38
5342	2023-10-17	1.8158	0.14

[5343 rows x 3 columns]

图 3-32　使用 AKShare 获取基金的历史单位净值

	净值日期	累计净值
0	2001-09-28	1.0000
1	2001-10-12	1.0005
2	2001-10-19	0.9997
3	2001-10-26	1.0013
4	2001-11-02	1.0013
...	...	...
5338	2023-10-11	3.8297
5339	2023-10-12	3.8302
5340	2023-10-13	3.8152
5341	2023-10-16	3.8082
5342	2023-10-17	3.8108

[5343 rows x 2 columns]

图 3-33　使用 AKShare 获取基金的历史累计净值走势

由于返回的数据是 DataFrame，因此此时可以直接使用 plot 方法进行绘图，代码如下：

```
//ch3/test_akshare.py
#获取累计净值走势
import matplotlib.pyplot as plt

plt.rcParams['font.sans-serif'] = ['SimHei']
fund_cum_val = ak.fund_open_fund_info_em(fund_symbol, indicator='累计净值走势')
fund_cum_val.plot()
plt.show()
```

运行以上代码可以得到如图 3-34 所示的绘图结果。

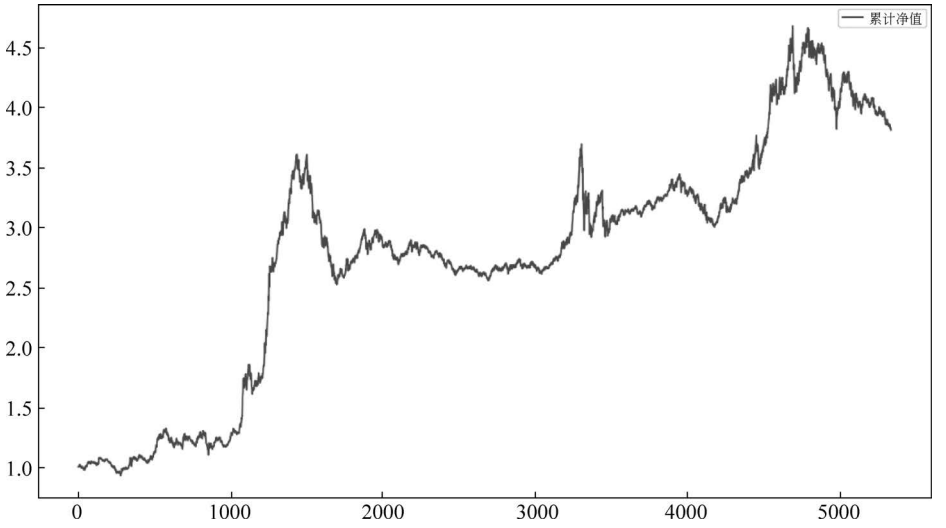


图 3-34　绘制历史累计净值走势图

3. 获取累计收益率走势

使用 fund_open_fund_info_em(fund, indicator='累计收益率走势')获取开放式基金的

累计收益率走势，得到如图 3-35 所示的结果。

```
          净值日期  累计收益率
0    2023-04-17   0.00
1    2023-04-18   0.00
2    2023-04-19  -0.39
3    2023-04-20  -0.24
4    2023-04-21  -2.19
..          ...    ...
116  2023-10-11 -11.93
117  2023-10-12 -11.91
118  2023-10-13 -12.63
119  2023-10-16 -12.97
120  2023-10-17 -12.84

[121 rows x 2 columns]
```

图 3-35　使用 AKShare 获取基金的历史累计收益率走势

需要注意的是，返回的累计收益率数据是近半年的，在进行数据拼接的时候需要进行平滑。

4. 获取同类排名走势

使用 fund_open_fund_info_em(fund, indicator='同类排名走势')获取开放式基金的同类排名走势，得到如图 3-36 所示的结果。

```
          报告日期  同类型排名-每日近三月排名  总排名-每日近三月排名
0     2013-01-04          230          389
1     2013-01-07          248          389
2     2013-01-08          250          389
3     2013-01-09          240          389
4     2013-01-10          233          389
...          ...          ...          ...
2617  2023-10-11         1597         3830
2618  2023-10-12         1816         3828
2619  2023-10-13         1629         3889
2620  2023-10-16         1576         3827
2621  2023-10-17         1630         3825

[2622 rows x 3 columns]
```

图 3-36　使用 AKShare 获取基金的历史同类排名走势

在返回的数据中，“同类型排名-每日近三月排名”表示该基金在同类型基金中的具体名次，“总排名-每日近三月排名”表示该基金所在的同类型总基金数量。

5. 获取同类排名百分比

使用 fund_open_fund_info_em(fund, indicator='同类排名百分比')获取开放式基金的同类排名百分比，得到如图 3-37 所示的结果。

```
          报告日期  同类型排名-每日近3月收益排名百分比
0     2013-01-04              40.87
1     2013-01-07              36.25
2     2013-01-08              35.73
3     2013-01-09              38.30
4     2013-01-10              40.10
...          ...                ...
2617  2023-10-11              58.30
2618  2023-10-12              52.56
2619  2023-10-13              58.11
2620  2023-10-16              58.82
2621  2023-10-17              57.39

[2622 rows x 2 columns]
```

图 3-37　使用 AKShare 获取基金的历史同类排名百分比

在返回的数据中，“同类型排名-每日近 3 月收益排名百分比”的值表示在同类型的基金中，当前基金近三月收益超越的百分比数，例如对于 2013-01-04 的数据而言，表示基金 202001 的近三月收益优于 40.87%的同类型基金。百分比数可以由 3.10.2 节中第 4 部分的同类排名走势计算得到，使用 1－(同类型排名/总排名)即可得到，读者可以自行验证。

6. 获取分红送配详情

使用 fund_open_fund_info_em(fund, indicator='分红送配详情')获取开放式基金的分红送配详情，得到如图 3-38 所示的结果。

	年份	权益登记日	除息日	每份分红	分红发放日
0	2022年	2022-01-18	2022-01-18	每份派现金0.0200元	2022-01-19
1	2021年	2021-01-15	2021-01-15	每份派现金0.0200元	2021-01-18
2	2020年	2020-01-16	2020-01-16	每份派现金0.0200元	2020-01-17
3	2019年	2019-01-17	2019-01-17	每份派现金0.0200元	2019-01-18
4	2018年	2018-01-16	2018-01-16	每份派现金0.0200元	2018-01-17
5	2017年	2017-01-18	2017-01-18	每份派现金0.0200元	2017-01-19
6	2016年	2016-01-19	2016-01-19	每份派现金0.0200元	2016-01-20
7	2015年	2015-01-19	2015-01-19	每份派现金0.0200元	2015-01-20
8	2011年	2011-01-19	2011-01-19	每份派现金0.0200元	2011-01-20
9	2010年	2010-03-17	2010-03-17	每份派现金0.0200元	2010-03-18
10	2008年	2008-04-25	2008-04-25	每份派现金0.0500元	2008-04-28
11	2007年	2007-04-26	2007-04-26	每份派现金0.1300元	2007-04-27
12	2007年	2007-01-29	2007-01-30	每份派现金1.3900元	2007-01-31
13	2006年	2006-02-23	2006-02-24	每份派现金0.0500元	2006-02-27
14	2004年	2004-12-17	2004-12-20	每份派现金0.0500元	2004-12-21
15	2004年	2004-04-05	2004-04-06	每份派现金0.0600元	2004-04-09
16	2003年	2003-12-22	2003-12-23	每份派现金0.0250元	2003-12-26
17	2002年	2002-09-19	2002-09-20	每份派现金0.0150元	2002-09-30
18	2002年	2002-04-19	2002-04-22	每份派现金0.0250元	2002-04-30

图 3-38　使用 AKShare 获取基金的历史分红送配详情

返回的结果一共分为 5 列，分别为年份、权益登记日、除息日、每份分红和分红发放日，其中最关键的信息是每份分红与分红发放日，对于返回结果中的每份分红是以文字的形式展现的，需要进一步地进行解析处理。

7. 获取拆分详情

使用 fund_open_fund_info_em(fund, indicator='拆分详情')获取开放式基金的拆分详情，由于 202001 没有历史的拆分数据，因此使用 000277 可以得到如图 3-39 所示的结果。

	年份	拆分折算日	拆分类型	拆分折算比例
0	2023年	2023-10-11	份额折算	1:1.0090
1	2023年	2023-08-03	份额折算	1:1.0090
2	2023年	2023-06-05	份额折算	1:1.0089
3	2023年	2023-04-06	份额折算	1:1.0089
4	2022年	2022-12-12	份额折算	1:0.9775
5	2022年	2022-11-03	份额折算	1:1.0081
6	2022年	2022-09-05	份额折算	1:1.0100
7	2022年	2022-07-05	份额折算	1:1.0081
8	2022年	2022-05-09	份额折算	1:1.0101
9	2022年	2022-03-03	份额折算	1:1.0101
10	2022年	2022-01-06	份额折算	1:1.0101
11	2021年	2021-11-03	份额折算	1:1.0101
12	2021年	2021-09-03	份额折算	1:1.0100
13	2021年	2021-07-05	份额折算	1:1.0101
14	2021年	2021-05-10	份额折算	1:1.0101
15	2021年	2021-03-03	份额折算	1:1.0101

图 3-39　使用 AKShare 获取基金的历史拆分详情

3.11 Tushare

Tushare是一个免费提供各类数据助力行业和量化研究的大数据开放社区，其拥有股票、基金、期货、数字货币等市场行情数据，同时也包括公司财务、基金经理等基本面数据，相较于需要收费的Wind、RQData等服务，Tushare是一个对于新手而言较为友好的量化数据获取方式，API的调用使用积分制，其官网首页如图3-40所示。

图3-40 Tushare官网首页

首先需要在官网注册一个Tushare的账号，登录之后在个人主页能够查看接口TOKEN，如图3-41所示。

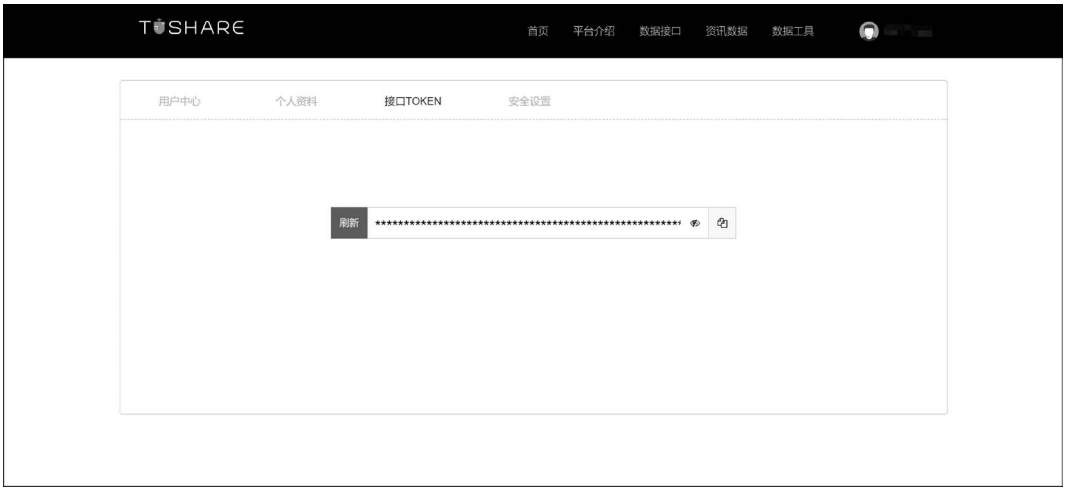


图3-41 获取Tushare的接口TOKEN

获取 TOKEN 后，可以使用如下代码测试是否能够正常使用接口：

```
//ch3/test_tushare.py
import tushare as ts

#读者的 Tushare 接口 TOKEN
TOKEN = '*********************************************************'
#需要读取行情的标的代码
TS_CODE = '165509.SZ'

#使用 TOKEN 初始化 API
pro = ts.pro_api(TOKEN)
#读取标的的日线数据
fund_info = pro.fund_basic()
print(fund_info)
```

运行以上代码，可以得到如图 3-42 所示的结果，可以看到能够正常读取到基金的基础数据，其中包含的字段有基金代码、基金名称、基金公司、基金托管人、基金类型、成立日期。

	ts_code	name	management	custodian	fund_type	found_date	\
1	512850.SH	中信建投北京50ETF	中信建投基金	招商银行	股票型	20180927	
2	168601.SZ	汇安裕阳三年定期开放	汇安基金	中国光大银行	混合型	20180927	
3	512860.SH	华安中国A股ETF	华安基金	中国农业银行	股票型	20180927	
4	159960.SZ	恒生国企	平安大华基金	中国银行	股票型	20180921	
5	501062.SH	南方瑞合三年	南方基金	中国建设银行	混合型	20180906	
6	510600.SH	沪50ETF	申万菱信基金	中国工商银行	股票型	20180903	
7	501061.SH	金选300C	中金基金	中国建设银行	股票型	20180830	
8	501060.SH	金选300A	中金基金	中国建设银行	股票型	20180830	
9	166802.SZ	浙商300	浙商基金	华夏银行	股票型	20180820	

图 3-42　使用 Tushare 读取基金的基础数据

同样，使用 Tushare 也可以获取基金的净值信息，代码如下：

```
//ch3/test_tushare.py
#读取基金的净值数据
fund_val = pro.fund_nav(ts_code = TS_CODE)
print(fund_val)
```

运行代码可以得到如图 3-43 所示的结果，可以看到 Tushare 返回的结果中同时包含单位净值与累计净值数据，相较于 AKShare 已经将数据对齐返回。

	ts_code	ann_date	nav_date	unit_nav	accum_nav	accum_div	\
0	165509.SZ	20181019	20181018	1.104	1.587	None	
1	165509.SZ	20181018	20181017	1.110	1.587	None	
2	165509.SZ	20181017	20181016	1.110	1.587	None	
3	165509.SZ	20181016	20181015	1.110	1.587	None	
4	165509.SZ	20181013	20181012	1.110	1.587	None	
5	165509.SZ	20181012	20181011	1.110	1.587	None	
6	165509.SZ	20181011	20181010	1.110	1.587	None	
7	165509.SZ	20181010	20181009	1.110	1.587	None	
8	165509.SZ	20181009	20181008	1.109	1.586	None	
9	165509.SZ	20180929	20180928	1.109	1.586	None	
10	165509.SZ	20180928	20180927	1.109	1.586	None	

图 3-43　使用 Tushare 读取基金的净值数据

Tushare 的 API 功能众多，更多的用法参见 Tushare 数据接口说明。

3.12 PyPortfolioOpt

PyPortfolioOpt 是一个开源的投资组合优化 Python 库，它可以方便地完成有效前沿的求解、Black-Litterman 资产配置模型等，同时可以使用 PyPortfolioOpt 完成各种期望回报的计算。下面以 3.9 节中使用的 202001.csv 文件为例，介绍 PyPortfolioOpt 的使用。

使用 PyPortfolioOpt 计算基金复权净值的日收益率，可以使用如下代码实现：

```
//ch3/test_pyportfolioopt.py
import pandas as pd
from pypfopt import expected_returns

#读取 CSV 数据
data = pd.read_csv('202001.csv')
#从复权净值计算收益率
returns = expected_returns.returns_from_prices(data['adjust_val'])
print(returns)
```

运行代码的结果如图 3-44 所示。

```
1       0.000000
2      -0.000300
3       0.001600
4       0.000000
5      -0.002397
          ...
5326    0.010521
5327   -0.002522
5328   -0.001883
5329    0.001078
5330   -0.004308
Name: adjust_val, Length: 5330, dtype: float64
```

图 3-44 使用 PyPortfolioOpt 计算基金复权净值的收益率

也可以使用 PyPortfolioOpt 计算基于复权净值的年化收益率，代码如下：

```
//ch3/test_pyportfolioopt.py
#计算年化收益率
mu = expected_returns.mean_historical_return(data['cum_val'])
print(mu, (1 + returns).prod() ** (252 / returns.count()) - 1)
```

PyPortfolioOpt 的 mean_historical_return 方法首先将传入的净值转换为收益率序列，将收益率序列进行复利计算后进行年化转换，从而得到最终的结果，在如上代码的最后 print 函数中展示了手动计算的过程。mean_historical_return 函数有许多可选的入参，上面的代码只是展示了默认参数的计算方法，更多用法可以参考官方文档。

由于 PyPortfolioOpt 模型的使用需要较强的理论知识，因此更多的原理与使用方法将在第 6 章中介绍。

3.13　empyrical

empyrical是一个金融风险指标开源Python库，通过empyrical读者可以方便地计算常见的金融风险指标，例如最大回撤、夏普比率等。使用empyrical计算最大回撤的代码如下：

```
//ch3/test_empyrical.py
import empyrical
import pandas as pd
from pypfopt import expected_returns

#读取CSV数据
data = pd.read_csv('202001.csv')
#由净值计算得到收益率
returns = expected_returns.returns_from_prices(data['cum_val'])
#计算收益率的最大回撤
md = empyrical.max_drawdown(returns)
print(md)
```

结合3.12节中通过PyPortfolioOpt计算收益率序列，将该序列传入empyrical的max_drawdown方法可以得到序列中的最大回撤，输出值为－0.301 469 365 123 372 6，说明在传入的收益率序列值中发生的最大亏损约为30%。表3-1中列出了empyrical提供的主要金融风险指标。

表3-1　empyrical提供的主要金融风险指标

API	指标名称	API	指标名称
alpha	Alpha系数	max_drawdown	最大回撤
annual_return	年收益率	omega_ratio	Omega比率
annual_volatility	年波动率	sharpe_ratio	夏普比率
beta	Beta系数	sortino_ratio	索提诺比率
calmar_ratio	卡玛比率	stability_of_timeseries	序列稳定性
capture	捕获比率	tail_ratio	尾部比率
conditional_value_at_risk	条件风险价值	up_alpha_beta	上行的Alpha和Beta系数
down_alpha_beta	下行的Alpha和Beta系数	up_capture	上行捕获比率
down_capture	下行捕获比率	up_down_capture	上下行捕获比率比值
downside_risk	下行风险	value_at_risk	风险价值

表3-1中列出的指标在本节不进行详细解释，更多指标相关原理与使用可以参考4.5节。

3.14　Orange

Orange是一个基于Python的数据挖掘和机器学习平台，由于其不依赖过多的代码就能完成数据分析的数据流，所以可以使用Orange快速验证模型和想法。Orange的官网如

图 3-45 所示。

图 3-45　Orange 的官网首页

从 Orange 的官网介绍不难看出，其是一个方便的数据挖掘的可视化工具，安装 Orange 有多种方式，其下载页面如图 3-46 所示，可以通过直接下载安装包进行本地安装，也可以通过 conda、pip 或者源码安装。为了方便起见，建议读者直接下载安装包进行安装。

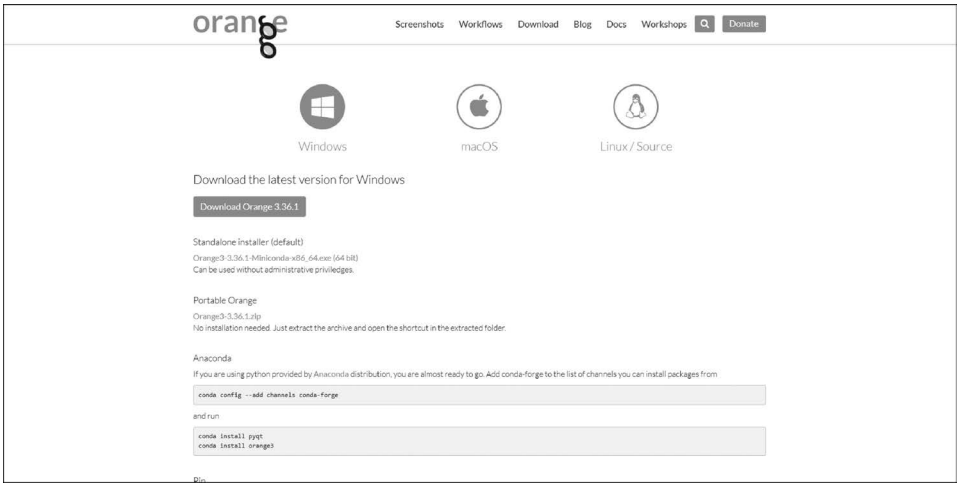


图 3-46　Orange 的下载页面

3.14.1　Orange 中的示例

安装完 Orange 后，启动后可以看到如图 3-47 所示的界面，可以通过单击 Help→Example Workflows 来查看 Orange 内置的部分工作流的示例，如图 3-47 所示，创建一个 Orange 中主成分分析（Principal Components Analysis，PCA）的工作流。

图 3-47　Orange 的主界面

对于工作流中的每个构件都可以双击打开、查看并调整其属性，在如图 3-48 所示的主成分分析工作流中，可以看到整个流的起始节点为 File 节点，说明数据输入是以文件的形式进行输入的，在示例中采用的是 brown-selected 数据集，其包含 186 个数据，其中每个数据由 79 个特征组成，数据总共可以分为 3 类。

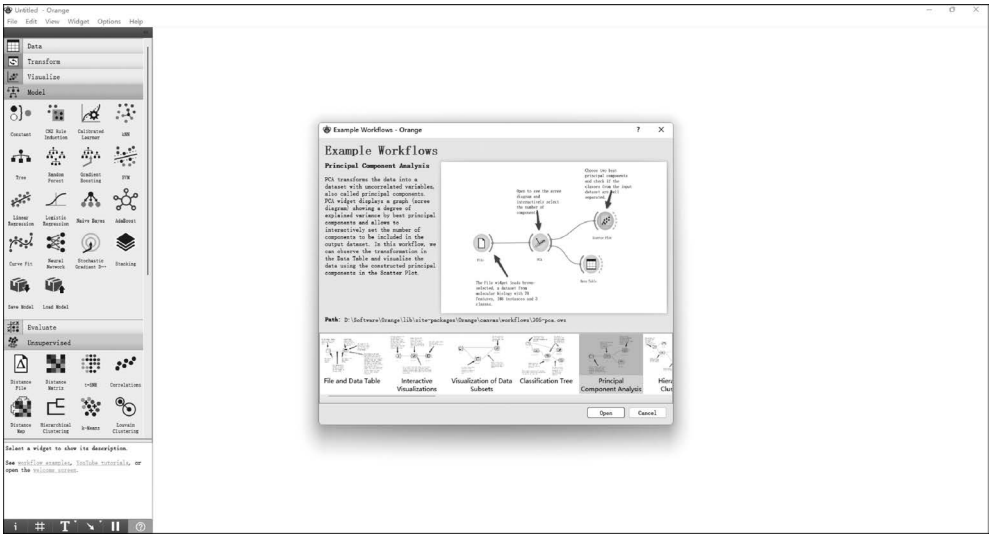


图 3-48　Orange 中的主成分分析示例

双击 File 节点，可以看到数据集的相关信息，例如各数据列的数据类型及其属性（特征、标签等），File 组件支持多种格式的数据，例如 CSV、Excel、H5 等数据，其也支持自动检测数据源格式，如图 3-49 所示。

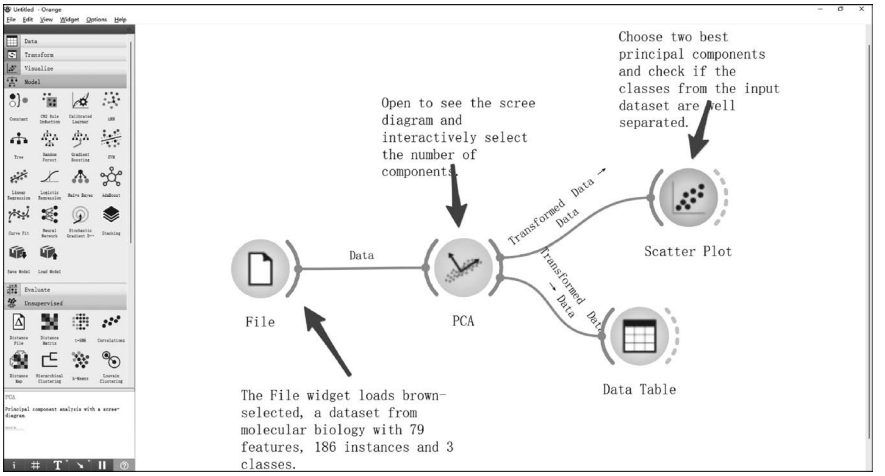


图 3-49　Orange 的主成分分析工作流图

类似地，双击 PCA 组件则能看到 PCA 的相关设置，例如降维后的维数等，每次为 PCA 进行不同的设置时，在勾选 Apply Automatically 选项之后都会自动生效。经过 PCA 降维处理后的数据流向了两个节点，分别是 Scatter Plot 和 Data Table，其中 Scatter Plot 会以散点图的形式绘制出降维后的数据，如图 3-50 所示，而 Data Table 则会以表格的形式展示降维后的数据，如图 3-51 所示。

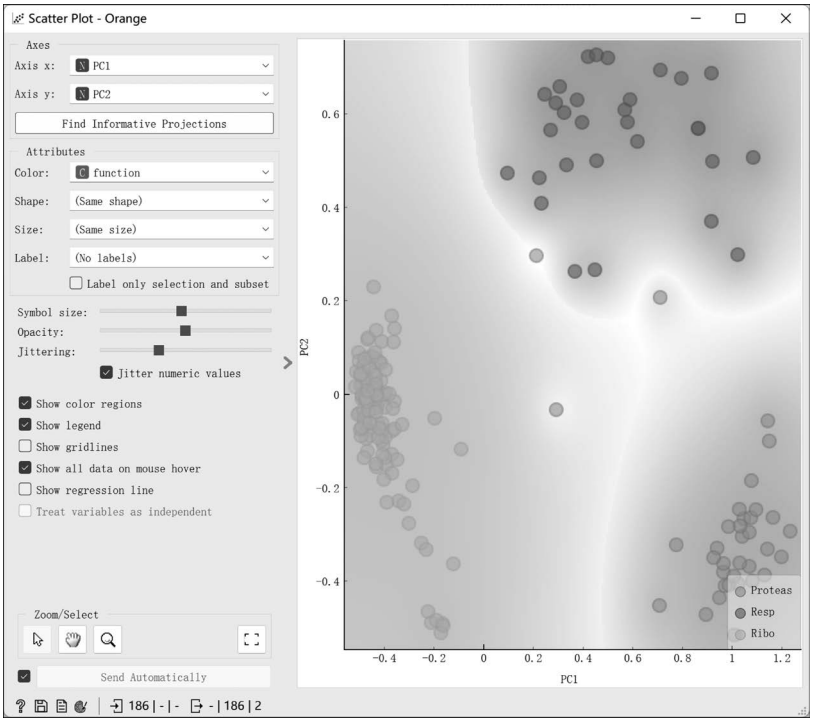
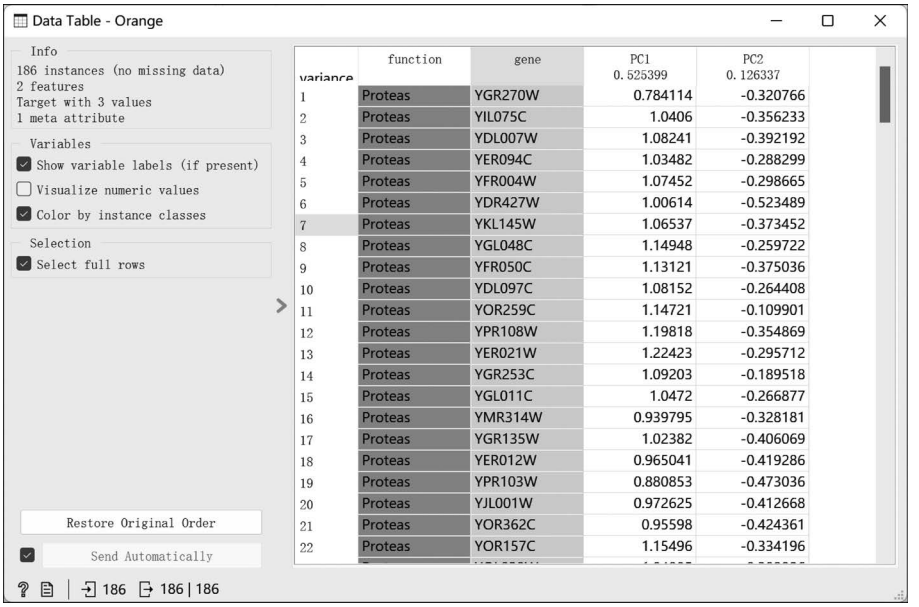


图 3-50　Orange 绘制的散点图



variance	function	gene	PC1 0.525399	PC2 0.126337
1	Proteas	YGR270W	0.784114	-0.320766
2	Proteas	YIL075C	1.0406	-0.356233
3	Proteas	YDL007W	1.08241	-0.392192
4	Proteas	YER094C	1.03482	-0.288299
5	Proteas	YFR004W	1.07452	-0.298665
6	Proteas	YDR427W	1.00614	-0.523489
7	Proteas	YKL145W	1.06537	-0.373452
8	Proteas	YGL048C	1.14948	-0.259722
9	Proteas	YFR050C	1.13121	-0.375036
10	Proteas	YDL097C	1.08152	-0.264408
11	Proteas	YOR259C	1.14721	-0.109901
12	Proteas	YPR108W	1.19818	-0.354869
13	Proteas	YER021W	1.22423	-0.295712
14	Proteas	YGR253C	1.09203	-0.189518
15	Proteas	YGL011C	1.0472	-0.266877
16	Proteas	YMR314W	0.939795	-0.328181
17	Proteas	YGR135W	1.02382	-0.406069
18	Proteas	YER012W	0.965041	-0.419286
19	Proteas	YPR103W	0.880853	-0.473036
20	Proteas	YJL001W	0.972625	-0.412668
21	Proteas	YOR362C	0.95598	-0.424361
22	Proteas	YOR157C	1.15496	-0.334196

图 3-51　Orange 展示的表格数据

从图 3-50 不难看出，由于降维维数默认为 2，因此绘制散点图时的 x 轴和 y 轴分别对应着 PC1 和 PC2 这两个降维后的生成数据，从绘制的散点图来看，将原本包含 79 维的特征降到二维后，其在平面内也是可分的，因此降维效果十分显著。

使用 Data Table 展示数据则允许用户查看所有降维后的数据，相较于散点图的展示形式，使用表格能够展示更多数据的具体值。不难发现，Orange 可以将上一个节点处理的数据结果输入若干不同的下游节点，极大地方便了用户验证自己的思路。

3.14.2　创建自己的工作流

本节将说明如何使用 Orange 创建自己的工作流，如果目前想验证使用 ARIMA 模型对基金复权净值的时间序列预测是否有效，则应如何使用 Orange 进行快速验证呢？最终总体的工作流图如图 3-52 所示。

总体来看，工作流大致分为 4 部分：数据读取、数据预处理、模型预测、结果可视化，数据读取的组件使用的是 CSV File Import，读取的文件为 3.9 节中使用的 202001.csv，接着使用 Select Columns 组件从源数据中选取并指定数据分析中忽略的列、特征值列及目标值列。在进行时间序列分析之前，需要将输入数据通过 As Timeseries 组件将数据转换为时间序列的数据，至此已经可以获取时间序列数据。在此之后，如图 3-53 所示，延伸出了 7 条不同分支，其中每个分支都是对时间序列数据的一种分析尝试，例如查看时间序列数据的周期图、自相关图、格兰杰因果关系检验、原数据一阶差分的 ARIMA 模型预测结果等，详细的原理在此不进行说明。

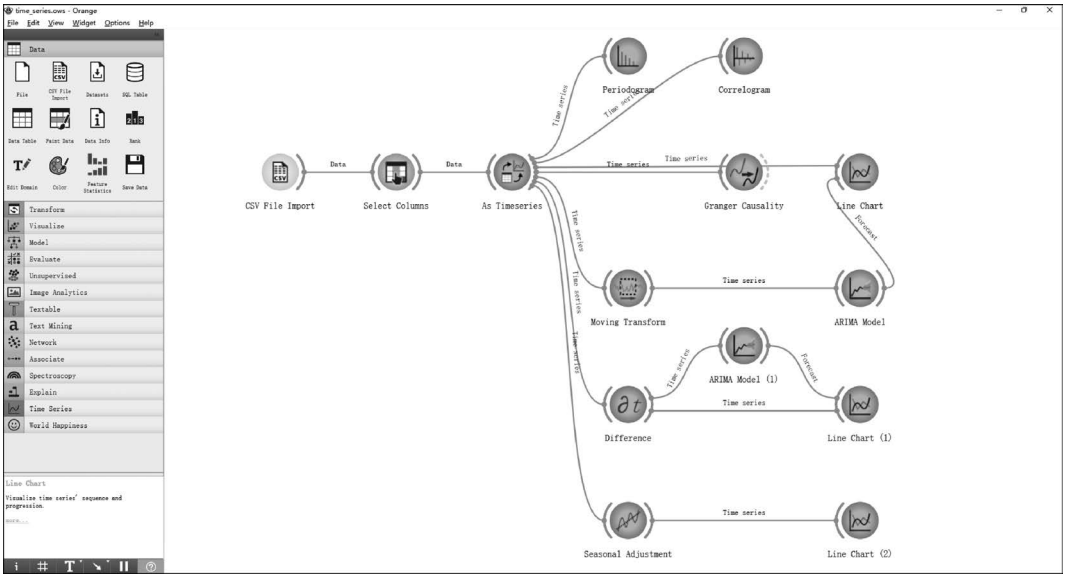


图 3-52　使用 Orange 完成 ARIMA 模型对时间序列的预测

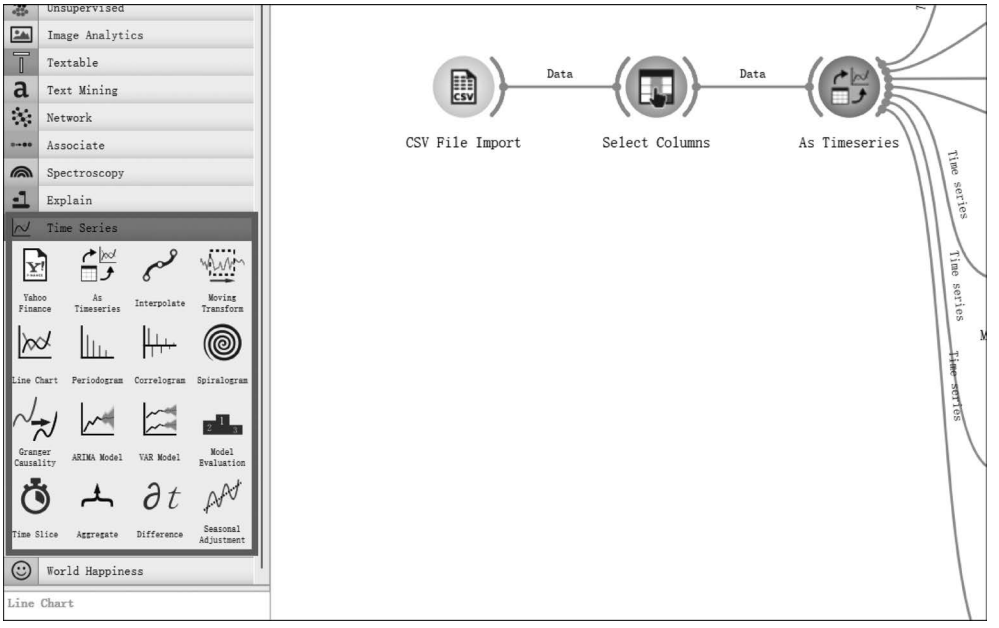


图 3-53　Orange 中时间序列相关组件

与时间序列数据处理相关的组件位于左侧的工具箱中，如图 3-53 所示，从图中可以看出其包含雅虎财经的数据源组件（Yahoo Finance），以及可以对时间序列进行内插值的组件（Interpolate）等，在确定需要使用的组件后，可以直接单击组件或将组件拖入右侧的画布中，再使用箭头将不同的组件连接起来，从而得到完整的数据流图。

读者可以在安装 Orange 之后打开 ch3/time_series.ows 文件，自行尝试使用 Orange 进行测试。

3.15　Optunity

Optunity 是一个包含用于超参数调整的各种优化器的库。无论是监督还是非监督学习方法，超参数调优都是必须解决的问题。

超参数的优化问题的目标函数通常是非凸的、非光滑的且难以直接求解其极值点的。Optunity 则提供了一种数值优化的方法，以此进行优化问题的求解，Optunity 由 Python 编写，不过其也能很方便地集成于 R 或 MATLAB 中。

Optunity 需要待优化的函数返回一个数值类型的值，并支持最大化或最小化目标函数值，在指定了优化算法后能够从优化器中获得最终最优参数的结果及优化过程的中间结果，十分直观与方便。

本节以常用的粒子群(Particle Swarm)优化算法，简要说明 Optunity 的使用。以优化函数 $y=(\sin(x)+2)\times x^2$ 为例，可以先用 Matplotlib 绘制出该函数的图像，代码如下：

```
//ch3/test_optunity.py
import optunity
import numpy as np
import matplotlib.pyplot as plt


def func(x):
    """ 待优化的函数 """
    return (np.sin(x) + 2) * x ** 2

x_range = [-100, 100]
xs = list(range(*x_range))
ys = [func(i) for i in xs]

#绘制待优化函数的图像
plt.plot(xs, ys)
plt.show()
```

运行以上程序得到的函数图像如图 3-54 所示。

从函数图像不难发现，其最小值应该位于 $x=0$ 附近，接下来使用 Optunity 寻找极值：

```
//ch3/test_optunity.py
opt = optunity.minimize(
            func, num_evals=500, solver_name='particle swarm', x=x_range
    )
opt_params, details, suggestion = opt
print(opt_params)
print(details)
print(suggestion)
```

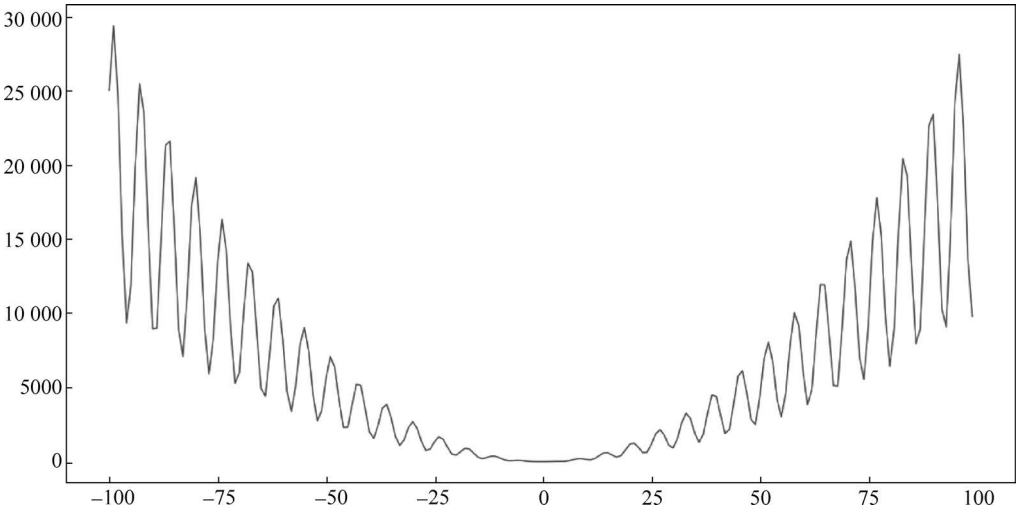


图 3-54　待优化函数图像

如上代码所示，由于搜寻的是目标函数的最小值，所以使用了 optunity.minimize 方法，类似地，在寻找最大值的时候可以使用 optunity.maximize 方法，其中 num_evals 表示优化过程中允许调用优化目标函数的最大次数，在指定了优化器方法 solver_name 后，需要传入优化目标函数的参数值的搜寻范围，如代码中将参数 x 的搜寻范围指定为[－100，100]。运行以上程序可以观察到其打印的 opt_params 结果为{'x'：－0.0025152026389108073}，是一个近似 $x=0$ 的结果。

得益于 Optunity 框架的灵活性，其只需用户传入一个可以返回数值类型的待优化目标函数，并指定该函数的输入参数的取值范围，因此在进行量化交易的回测时，可以编写函数返回收益率或回撤值作为待优化参数，对该函数进行最大值或最小值优化。

3.16　Optuna

类似于 3.15 节中的 Optunity，Optuna 也是一个参数优化工具，其支持自定义剪枝函数等，并且支持对于不同类型取值的初始化，例如枚举值、整型值、浮点型值等都有不同的参数值初始化取值方法。下面的代码说明了如何使用 Optuna 对数值型的函数进行优化：

```
//ch3/test_optuna.py
import optuna
import numpy as np

def func(x):
    """ 待优化的函数 """
    return (np.sin(x) + 2) * x ** 2

def objective(trial):
```

```
    x_range = trial.suggest_uniform('x', -100, 100)
    return func(x_range)

study = optuna.create_study()
study.optimize(objective, n_trials=500)
print(study.best_params)
```

代码中选用的待优化目标函数与3.15节中一样，在Optuna中，需要先定义一个学习任务study，在创建study时可以为其传入direction参数以表示优化方向（最小化minimize或最大化maximize），默认值为minimize。

创建学习任务后，再向该任务中添加优化目标objective，需要在目标中使用suggest相关的方法（如代码中的suggest_uniform则是以均匀分布生成随机数）为目标函数生成初始值，再将这些初始值传入目标函数，对得到的目标值进行评价，由于在上述代码中目标函数的评价值即为函数值，因此无须再进行额外的评价过程。

在调用optimize方法时，需要将n_trials指定为实验次数，完成优化后，打印study.best_params则是优化后的最优参数值。除此之外，还可以打印best_value、best_trial等，分别表示最优参数下的最优函数值及最优的一次实验相关信息等。

在进行量化回测方面的相关优化时，其通常只涉及对于数值的优化，因此Optuna与Optunity在使用成本上相似，读者可以根据自身习惯进行选取。

3.17　小结

本章介绍了常用于投研的Python库和工具，包括数据处理与可视化、Python编程辅助包、金融数据获取与分析的工具等几大类。读者可以根据自身情况学习与使用这些Python工具，为后面的章节打下基础。