第 3 章

CHAPTER 3

Python 编程基础知识

本章聚焦于量化交易中所使用的基本编程技术，力求通过最小的篇幅来展示量化编程所需要的基础知识点，让读者更好地上手 Python 编程语言。

3.1 基本语法

掌握 Python 编程语言首先需要掌握基本的语法，主要包括注释、标识符命名规则、变量、常量及数据的输入和输出等。

3.1.1 注释

在 Python 语言中，注释用于解释代码，提高可读性，并帮助程序员理解代码逻辑。Python 支持单行注释和多行注释。尤其在量化程序中，由于复杂的逻辑和数据结构，量化机构为了让不同岗位的研究员和交易员可以沟通、完善代码，一般需要在量化策略代码中进行详细注释，这样既有利于部门间的沟通，也有利于程序员不断地改进代码。

根据注释的行数，一般可以分为单行注释和多行注释。

1. 单行注释

在 Python 语言中通过 # 进行标注，代码如下：

```
#这也是一个单行注释
print("Hello, World!")
```

2. 多行注释

在 Python 语言中使用三重引号('''或""")来注释多行文本，代码如下：

```
'''
这是一个多行注释
可以有多行内容
'''
print("Hello, World!")
```

3.1.2 标识符命名规则

标识符用于标识变量、函数、类、模块或其他对象的名称。在 Python 语言中变量命名需要基本的规则，主要规则如下：第一，只能包含字母(a～z，A～Z)、数字(0～9)和下画线(_)；第二，不能以数字开头；第三，变量名区分大小写，即 price 和 Price 是两个变量；第四，不可使用 Python 的关键字和内置函数名。

以上 4 条规则为 Python 变量命名必须遵守的规则。还有一些规范性要求：第五，类的首字母为大写，例如，Class Strategy；第六，变量名称尽量用小写，例如，close、open、high、low；第七，如果变量有多个单词，则中间以下画线连接，如 low_price。

关键字是 Python 语言保留的词汇，具有特定的功能。不能将关键字用作标识符，否则在程序中会报错。查询方法的代码如下：

```
import keyword
print(keyword.kwlist)
```

输出如下：

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in',
'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

3.1.3 变量与常量

1. 变量

变量是计算机中用来存储计算结果或数值的抽象概念，变量的值可以改变，变量通过变量名访问。在 Python 语言中，可以定义变量，但与其他语言不同的是，Python 不需要显式声明变量的类型。以一个最简单的示例来说明 Python 中的变量：

```
price = 31.2
print(f'股票价格是: {price}')
```

输出如下：

```
股票价格是: 31.2
```

在这个程序中，添加的股票价格 price 变量指向股价 31.2 这个值，值的类型为 float，这是一种用于表示浮点数(有小数点的数字)的数据类型。

```
赋值的基本格式为<变量> = <表达式>
```

其中，左边为一个变量，右边为表达式，=表示赋值语句。

在 Python 编程中，赋值分为单变量赋值和多变量赋值。单变量赋值是 Python 中最基础的赋值操作，它用于将一个值赋给单个变量。在金融场景中，单变量赋值常用于初始化各种数据，如股票价格、交易量、账户余额等，代码如下：

```
stock_price = 32.5                  #初始化股票价格为 32.5 元
initial_capital = 100000            #初始化账户余额为 100000 元
hold = False                        #初始不持仓
```

多变量赋值是 Python 的一种强大的特性，它允许在一行代码中同时为多个变量赋值。多变量赋值可以简化代码，提高编程效率，尤其在需要同时处理多变量时，代码如下：

```
#一次性给开盘价、最高价、最低价和收盘价赋值
open, high, low, close = 101.20, 103.50, 100.80, 102.90
print(f'开盘价：{open},最高价：{high},最低价：{low},收盘价：{close}')
```

输出如下：

```
开盘价：101.2,最高价：103.5,最低价：100.8,收盘价：102.9
```

由于量化金融数据经常通过应用程序接口（Application Program Interface，API）获取存在列表的股票数据，所以 Python 可以通过列表和元组拆包的方式给多个变量赋值，代码如下：

```
#通过列表进行赋值
price_list = [25.1, 25.6, 23.7, 24.9]
open, high, low, close = price_list
print(f'开盘价：{open},最高价：{high},最低价：{low},收盘价：{close}')
```

输出如下：

```
开盘价：25.1,最高价：25.6,最低价：23.7,收盘价：24.9
```

总之，Python 的多变量赋值非常灵活，既可以直接通过多个数据赋值，也可以通过列表、元组、字典等数据格式来赋值，极大地提高了金融数据的处理效率。

2. 常量

常量是值不可改变的数据容器。Python 中没有内置的常量类型，但可以通过命名约定来表示常量（通常使用全大写字母命名常量）。常量通常放置在代码的最上部，并作为全局使用，示例代码如下：

```
#定义常量
PI = 3.14159
MAX_SPEED = 120
```

3.1.4 数据的输入和输出

1. 数据输入

input()函数用于从标准输入（通常是键盘）读取数据。它会显示提示信息，并等待用户输入。在默认情况下，input()函数返回用户输入的数据作为一个字符串，代码如下：

```
#提示用户输入股票名
name = input("请输入股票名：")
print(f"股票名为 {name}!")
```

输出结果如图 3-1 所示。

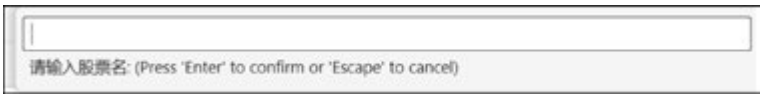

图 3-1 input()函数输入窗口

当在对话框里输入任一股票名时,如特斯拉,然后按 Enter 键,输出如下:

```
股票名为特斯拉!
```

2. 数据输出

在 Python 语言中,常用的数据输出函数为 print()函数,代码如下:

```
#打印不同类型的数据
print("Hello, World!")              #打印字符串
print(123)                          #打印整数
print(3.14)                         #打印浮点数
print([1, 2, 3])                    #打印列表
```

输出如下:

```
Hello, World!
123
3.14
[1, 2, 3]
```

3.2 数据类型与运算

熟悉数据类型并掌握基本的运算方法是量化程序开发的重要基础,本节介绍 Python 编程中几种常用的数据类型并通过简单案例进行运算分析。

3.2.1 数值类型

Python 提供多种数值类型,包括数值型(int、float 和 complex)类型。type()函数是 Python 中的一个内置函数,用于返回对象的类型。

1. 整数类型(int)

即没有小数部分的数字,整数可以是正数、0 或负数,代码如下:

```
x,y,z = 10, - 10,0                  #打印出各个变量的数值类型
print(f'X类型:{type(x)},Y类型:{type(y)},Z类型:{type(z)}')
```

输出如下:

```
X类型:<class 'int'>,Y类型:<class 'int'>,Z类型:<class 'int'>
```

2. 浮点数类型(float)

浮点数(float)是一种用于表示实数的数据类型,支持小数和科学记数法表示,代码如下:

```
pi = 3.14159
e = 2.71828
print(f'pi类型: {type(pi)},e类型: {type(e)}')
```

输出如下:

```
pi类型: <class 'float'>,e类型: <class 'float'>
```

3.2.2 字符串类型

Python中字符串是一种字符的序列,用单引号' '、双引号" "或者三引号括起来。例如,s1='hello',s2="world";字符串可以进行连接(拼接)、重复、切片等操作。

1. 拼接操作

在Python中,字符串拼接可以通过+运算符、join()方法或f-string(格式化字符串)实现。假设s1='hello',s2="world",代码如下:

```
#第3章/string_01.py
s1 = 'hello'
s2 = 'world'
#1. 通过运算符
greeting = s1 + " " + s2
print(greeting)
#2. 使用join方法
greeting1 = ''.join([s1, s2])                    #使用join() 拼接
print(greeting1)
#3. 使用f-string
greeting2 = f"{s1} {s2}"                         #使用f-string拼接
print(greeting2)
```

输出如下:

```
hello world
hello world
hello world
```

2. 重复操作

Python中可以通过*来实现字符串的乘法操作,代码如下:

```
repeated = s1 * 3
```

输出如下:

```
'hellohellohello'
```

3. 切片

切片用于从字符串中提取出一个子字符串，通过指定起始索引、结束索引和步长实现，其语法结构为 string[start:end:step]，start 为切片起始位置，end 为终止位置，step 为步长，注意切片包括起始位置，但不包括终止位置，代码如下：

```
substring = s1[1:4]
substring2 = s1[::-1]            #反转字符串
print(substring)
print(substring2)
```

输出如下：

```
ell
olleh
```

3.2.3 布尔类型

布尔类型是一种逻辑类型，主要用于表示真(True)和假(False)，布尔类型有两个值：True 和 False，常用于条件判断、逻辑运算。

在量化金融编程中，布尔类型是一种广泛使用的数据类型，主要用于条件判断和逻辑操作，常用在信号生成、风险管理、循环控制和股票筛选等过程中，从而优化交易策略。下面通过量化交易场景来熟悉布尔类型数据的运用。

1. 筛选股票

假设当前有几只股票的数据表 DataFrame，想筛选出那些满足特定条件的股票，筛选标准为当前价格高于 50 元且成交量大于 10000 股的股票，代码如下：

```
#第3章/stockscreen.py
import pandas as pd
data = {
    'stock': ['AAPL', 'GOOGL', 'MSFT', 'AMZN'],
    'price': [150, 2800, 300, 3400],
    'volume': [12000, 9000, 15000, 8000]
}
df = pd.DataFrame(data)
print(f'df:\n{df}\n')
print(f'筛选条件是价格大于50元且交易量大于10000股\n')
#使用布尔类型进行筛选
filtered_stocks = df[(df['price'] > 50) & (df['volume'] > 10000)]
print(filtered_stocks)
```

输出如下：

```
df:
   stock   price  volume
0   AAPL    150   12000
```

```
1  GOOGL   2800    9000
2   MSFT    300   15000
3   AMZN   3400    8000

筛选条件是价格大于 50 元且交易量大于 10000 股

  stock  price  volume
0  AAPL    150   12000
2  MSFT    300   15000
```

2. 生成交易信号

假设有一个简单移动均线(SMA)策略,当短期均线上穿长期均线时,生成买入信号;当短期均线下穿长期均线时,生成卖出信号。根据条件判断的结果,可以执行相应的交易操作。当 cross_above 为真时,执行买入操作;当 cross_below 为真时,执行卖出操作,代码如下:

```
#第 3 章/SMA.py
import pandas as pd
#假设有价格数据
data = {'price': [100, 101, 102, 105, 104, 106, 107, 110, 109, 108, 107, 109, 111, 113, 115]}

df = pd.DataFrame(data)
#定义短期和长期窗口
short_window = 3
long_window = 5
#计算短期和长期移动均线
df['short_ma'] = df['price'].rolling(window = short_window).mean()
df['long_ma'] = df['price'].rolling(window = long_window).mean()

#计算布尔值,判断买卖信号
df['cross_above'] = df['short_ma'] > df['long_ma']
df['cross_below'] = df['short_ma'] < df['long_ma']

#遍历数据并根据布尔值执行操作
for i in range(1, len(df)):
    if df.loc[i, 'cross_above'] and not df.loc[i - 1, 'cross_above']:
        print(f"买入信号: 价格 {df.loc[i, 'price']}")
    elif df.loc[i, 'cross_below'] and not df.loc[i - 1, 'cross_below']:
        print(f"卖出信号: 价格 {df.loc[i, 'price']}")
```

输出如下:

```
买入信号: 价格 104
卖出信号: 价格 107
买入信号: 价格 111
```

在这个简单移动均线策略代码中,布尔值用于判断买入和卖出信号,以决定是否执行交易操作。具体来看,①df['cross_above'] 是一个布尔值列,用于判断短期均线是否高于长期

均线；②df['cross_below']是另一个布尔值列，用于判断短期均线是否低于长期均线；③在遍历数据的 for 循环中，布尔值用于比较每行的 cross_above 和 cross_below 值与前一行布尔值。当 cross_above 从 False 变为 True 时，意味着短期均线上穿了长期均线，触发买入信号。当 cross_below 从 False 变为 True 时，表示短期均线下穿了长期均线，触发卖出信号。

3. 风险管理

假设量化策略有一个风险控制策略，当某只股票的波动率超过一定阈值时，触发止损，代码如下：

```
#示例股票波动率数据
volatility = np.array([0.02, 0.03, 0.05, 0.04, 0.06, 0.07, 0.03, 0.02])
threshold = 0.05
stop_loss_signal = volatility > threshold       #生成止损信号
print("Stop Loss Signal:", stop_loss_signal)
```

输出如下：

```
Stop Loss Signal: [False False False False True True False False]
```

3.2.4 列表、元组与集合

1. 列表（List）

列表是一种有序的可变的数据类型，可以存储任意类型的对象。在 Python 语言中列表用[]表示，用逗号将其中的元素隔开。列表在 Python 语言中具有丰富和灵活的操作方法，使其成为数据处理和结构管理中不可或缺的工具。列表中存储股票名字，代码如下：

```
stocks = ["Apple Inc.", "Microsoft Corporation", "Amazon.com, Inc."]
print(stocks)
```

输出如下：

```
['Apple Inc.', 'Microsoft Corporation', 'Amazon.com, Inc.']
```

以上展示了列表的定义和打印。接下来，需要了解列表的基本操作，使用列表来存储和操作数据。

（1）访问列表中的值。通过下标索引来访问列表中的值，索引从 0 开始。

代码如下：

```
stocks = ["Apple Inc.", "Microsoft Corporation", "Amazon.com, Inc."]
#访问第一只股票名字
print(stocks[0])
```

输出如下：

```
'Apple Inc. '
```

(2) 使用负索引从列表末尾开始访问元素。

代码如下：

```
#返回股票池中的最后一只股票
print(stocks[-1])
```

输出如下：

```
'Amazon.com, Inc.'
```

(3) 通过切片操作获取子列表。

在 Python 列表中，也可以通过切片轻松地获取列表中的一段子集，而不需要写循环。切片使用冒号(:)作为分隔符，其基本语法如下：

```
list[start:end:step]
```

其中，start 表示切片开始的索引(包含)，end 表示切片结束的索引(不包含)，step 表示步长，即每隔几个元素取一个，默认为 1。

假设在股票代码池 stocks 中存储了苹果、微软、亚马逊等公司的股票简称，对其进行基本的切片操作，代码如下：

```
stocks = ["Apple", "Microsoft", "Amazon", "Alphabet", "Tesla", "NVIDIA", "Meta"]
```

第一，基本切片，代码如下：

```
#从索引 1 切到 2
print(stocks[1:3])
```

输出如下：

```
['Microsoft', 'Amazon']
```

上例中，stocks [1:3]将返回从第 2 个到第 3 个元素的子列表。

第二，从头开始切片，代码如下：

```
stocks[:4]
```

输出如下：

```
['Apple', 'Microsoft', 'Amazon', 'Alphabet']
```

第三，切到结尾元素，代码如下：

```
stocks[1:]          #从第 2 个元素一直切到结尾
```

输出如下：

```
['Microsoft', 'Amazon', 'Alphabet', 'Tesla', 'NVIDIA', 'Meta']
```

第四，带步长的切片，即每隔几步切一次，代码如下：

```
stocks[::3]         #从头到尾每隔 3 个元素取一个
```

输出如下：

```
['Apple', 'Alphabet', 'Meta']
```

第五，反向切片。这种切片在对股票时序数据进行处理时常用，代码如下：

```
stocks[::-1]
```

输出如下：

```
['Meta', 'NVIDIA', 'Tesla', 'Alphabet', 'Amazon', 'Microsoft', 'Apple']
```

(4) 插入列表元素。在列表中，append()方法在列表末尾添加元素，代码如下：

```
stocks.append("Intel")
print(stocks)
```

输出如下：

```
['Meta', 'NVIDIA', 'Tesla', 'Alphabet', 'Amazon', 'Microsoft', 'Apple','Intel']
```

insert()方法可以在指定位置插入元素，例如，在索引 1 位置插入 "Oracle"，代码如下：

```
stocks.insert(1, "Oracle")
print(stocks)
```

输出如下：

```
['Apple', 'Oracle', 'Microsoft', 'Amazon', 'Alphabet', 'Tesla', 'NVIDIA', 'Meta']
```

(5) 删除列表元素。

del 语句和 remove()方法可以删除元素。如果需要删除指定位置的股票，则代码如下：

```
del stocks[3]
print(stocks)
```

输出如下：

```
['Apple', 'Microsoft', 'Amazon', 'Tesla', 'NVIDIA', 'Meta']
```

如果需要删除指定值的元素，则代码如下：

```
stocks = ["Apple", "Microsoft", "Amazon", "Alphabet", "Tesla", "NVIDIA", "Meta"]
stocks.remove("Tesla")
print(stocks)
```

输出如下：

```
['Apple', 'Microsoft', 'Amazon', 'Alphabet', 'NVIDIA', 'Meta']
```

(6) 更新和修改列表元素。

假设需要将"Amazon" 股票修改为 "AMD"股票，代码如下：

```
#将索引 2 的 "Amazon" 修改为 "AMD"
stocks[4] = "AMD"
print(stocks)
```

输出如下：

```
['Apple', 'Microsoft', 'AMD', 'Alphabet', 'NVIDIA', 'Meta']
```

(7) 列表排序。

列表排序主要介绍两种方法：sort()方法、sorted()函数。

第一，sort()方法用于对列表排序。sort()方法的示例代码如下：

```
stocks.sort()
print(stocks)
```

输出如下：

```
['AMD', 'Alphabet', 'Apple', 'Meta', 'Microsoft', 'NVIDIA']
```

默认为升序排序，当需要进行降序排序时代码为 stocks.sort(reverse=True)。

第二，使用 sorted()函数进行排序，示例代码如下：

```
sorted_stocks = sorted(stocks)
print(sorted_stocks)
```

输出如下：

```
['AMD', 'Alphabet', 'Apple', 'Meta', 'Microsoft', 'NVIDIA']
#原列表保持不变
print(stocks)
```

输出如下：

```
['AMD', 'Alphabet', 'Apple', 'Meta', 'Microsoft', 'NVIDIA']
```

默认为升序排序，当需要进行降序排序时需要对参数进行设置，代码如下：

```
sorted_stocks_desc = sorted(stocks, reverse = True)
print(sorted_stocks_desc)
```

输出如下：

```
['NVIDIA', 'Microsoft', 'Meta', 'Apple', 'Alphabet', 'AMD']
```

reverse 参数：当设置为 True 时，会将列表按降序排序。

2. 元组(Tuple)

元组是一种不可变的序列类型，用圆括号括起来。它们可以存储任意数量和类型的元素，但一旦创建，元素就不能修改。元组的主要操作包括访问、切片、拼接和计算等。在 Python 语言中，元组是通过圆括号()括起来的有序集合。它可以包含不同类型的元素，例如(5, "text", 6.98)。

元组这种数据结构主要具有 4 个特点，一是不可变性：元组的元素在初始化后不能更改。这意味着无法对元组进行添加、删除或改变元素操作。二是有序性：元组中的元素按照其位置顺序排列，每个元素都有固定的索引。三是可迭代性：元组的元素可以被遍历，支持所有的迭代操作。四是可嵌套性：元组可以在其元素中包含其他元组，形成多维元组。以下通过示例来介绍元组的基本操作。

1）创建元组

创建元组可直接使用圆括号括起来多个用逗号分隔的元素，代码如下：

```
stocks = ("Apple", "Microsoft", "Amazon", "Alphabet", "Tesla", "NVIDIA", "Meta")
```

以上生成了一个包含 7 只股票的元组。

2）访问和操作元素

访问和操作元组元素可使用索引访问元组中的元素，索引从 0 开始，代码如下：

```
first_stock = stocks[0]                    #访问第 1 个元素
print(first_stock)
```

输出如下：

```
"Apple"
```

```
last_stock = stocks[ - 1]                   #访问最后一个元素
print(last_stock)
```

输出如下：

```
"Meta"
```

3）切片操作

通过切片获取元组的部分元素，代码如下：

```
selected_stocks = stocks[1:4]          #获取索引 1 到 3 的元素
print(selected_stocks)
```

输出如下：

```
("Microsoft", "Amazon", "Alphabet")
```

4）拼接元组

使用加法运算符可以将两个元组连接成一个新的元组，代码如下：

```
additional_stocks = ("IBM", "Cisco")
all_stocks = stocks + additional_stocks
print(all_stocks)
```

输出如下：

```
("Apple", "Microsoft", "Amazon", "Alphabet", "Tesla", "NVIDIA", "Meta", "IBM", "Cisco")
```

5）重复元组

可以使用 * 运算符将元组重复多次，代码如下：

```
repeated_stocks = stocks * 2
print(repeated_stocks)
```

输出如下：

```
("Apple", "Microsoft", "Amazon", "Alphabet", "Tesla", "NVIDIA", "Meta", "Apple",
"Microsoft", "Amazon", "Alphabet", "Tesla", "NVIDIA", "Meta")
```

6）元组的长度

可以使用 len() 函数获取元组中元素的数量，代码如下：

```
length = len(stocks)
print(length)
```

输出如下：

```
7
```

7）元组不可变性

元组是不可变的，一旦创建就不能修改其内容。如果尝试修改，则会引发错误，代码如下：

```
stocks[0] = "Google"
#这行代码会引发 TypeError
```

输出如下：

```
TypeError                                 Traceback (most recent call last)
Cell In[20], line 1
----> 1 stocks[0] = "Google"

TypeError: 'tuple' object does not support item assignment
```

综上，元组作为 Python 中的一种重要的数据结构，其不可变性提供了一定的安全性和稳定性，适用于需要保护数据不被更改的场景。熟悉和掌握元组的操作，有助于提高量化编程的效率和代码的质量。

3. 集合（Set）

在 Python 中，集合是一种无序且不重复的元素集合，适合用来处理需要唯一性和去重的情况。集合是无序的，无法通过索引访问其中的元素，常见操作如下。

1）创建集合

可使用花括号{}或 set()方法来创建集合，代码如下：

```
stocks = {"Apple", "Microsoft", "Amazon", "Alphabet", "Tesla", "NVIDIA"}
#使用 set()将其他可迭代对象转换为集合
Stocks1 = ["Apple", "Microsoft", "Amazon", "Alphabet", "Tesla", "NVIDIA"]
Stocks1 = set(stocks)
```

2）添加元素

在集合中一般通过 add()和 update()方法来添加元素，其中，add()方法用于向集合中

添加单个元素，但若集合中已经有此元素，则不会重复添加，代码如下：

```
stocks.add("IBM")
print(stocks)
```

输出如下：

```
{'NVIDIA', 'IBM', 'Amazon', 'Microsoft', 'Alphabet', 'Apple', 'Tesla'}
```

update()方法用于将一个可迭代对象(列表、元组、字典等)的所有元素添加到集合中，如可迭代对象已存在，则不重复添加到集合，代码如下：

```
#需要添加的新股票列表
new_stocks = ["Facebook", "Twitter"]
#使用 update() 方法将 new_stocks 列表的元素添加到 stocks 集合(上文已定义)
stocks.update(new_stocks)
#输出更新后的集合
print(stocks)
```

输出如下：

```
{'Apple', 'NVIDIA', 'IBM', 'Facebook', 'Microsoft', 'Alphabet', 'Amazon', 'Twitter', 'Tesla'}
```

3) 删除元素

remove()方法可以删除指定元素，若元素不存在，则会引发错误，代码如下：

```
stocks.remove("Amazon")
print(stocks)
```

输出如下：

```
{'Apple', 'NVIDIA', 'IBM', 'Facebook', 'Microsoft', 'Alphabet', 'Twitter', 'Tesla'}
```

4) 集合运算

集合运算包括交集、并集及差集。

第一，交集返回两集合中共同存在的元素。假设求集合 portfolio_a 、portfolio_b 的交集，可使用 intersection() 或 & 运算符，代码如下：

```
#第 3 章/intersection.py
portfolio_a = {"Apple", "Microsoft", "Tesla"}
portfolio_b = {"Tesla", "NVIDIA", "IBM"}
common_stocks = portfolio_a.intersection(portfolio_b)
common_stocks1 = portfolio_a & portfolio_b
print(common_stocks)
print(common_stocks1)
```

输出如下：

```
{'Tesla'}
{'Tesla'}
```

第二，集合并集是指将两集合的所有元素并成一个新集合，并除去重复元素。假设求集

合 portfolio_a 、portfolio_b 的并集，可通过两种方法实现，代码如下：

```
all_stocks = portfolio_a.union(portfolio_b)
all_stocks1 = portfolio_a | portfolio_b
print(all_stocks)
print(all_stocks1)
```

输出如下：

```
{'NVIDIA', 'IBM', 'Microsoft', 'Apple', 'Tesla'}
{'NVIDIA', 'IBM', 'Microsoft', 'Apple', 'Tesla'}
```

第三，集合差集是指从一个集合中去掉另一个集合中存在的元素。集合的差集使用 difference()方法或－运算符。假设求集合 portfolio_a 与集合 portfolio_b 的差集，代码如下：

```
unique_to_a = portfolio_a.difference(portfolio_b)
unique_to_b = portfolio_a - portfolio_b
print(unique_to_a)
print(unique_to_b)
```

输出如下：

```
{'Microsoft', 'Apple'}
{'Microsoft', 'Apple'}
```

3.2.5 字典

在 Python 中，字典(dict)是一种可变的无序的集合类型，用于存储键-值对。字典的键必须是唯一的且不可变的(如字符串、数字或元组)，字典值可以是任何数据类型。字典使用花括号{}来定义，每个键-值对之间用逗号分隔，键和值之间用冒号分隔。

字典的主要特点有：第一，键值对结构，每个元素由唯一的键(Key)和对应的值(Value)组成。第二，动态可变性，支持动态增删改操作，如 dict[key]＝value。第三，值的多样性，值可以是任意 Python 对象(如数字、字符串、列表、其他字典等)。第四，可迭代性，支持通过循环遍历键、值或键值对。

1. 构建字典

第一，Python 中可用花括号{}或 dict()来定义字典。字典中可以存储不同类型的数据，如列表和字典，代码如下：

```
stock_info = {
    "AAPL": {"close": 231.41, "volume": 1000},
    "TSLA": {"close": 269.19, "volume": 500}
}
```

使用 dict()定义一个嵌套字典，字典中包含字典，代码如下：

```
stock_info = dict(AAPL = {"close": 231.41, "volume": 1000}, TSLA = {"close": 269.19,
"volume": 500})
print(stock_info)
```

输出如下：

```
{'AAPL': {'close': 231.41, 'volume': 1000},
'TSLA': {'close': 269.19, 'volume': 500}}
```

第二，用其他结构数据生成字典，如通过列表以字典推导式生成字典，代码如下：

```
tickers = ["AAPL", "TSLA","IBM"]
closes = [231.41, 269.19,214.67]
stock_closes = {tickers[i]: closes[i] for i in range(len(tickers))}
print(stock_closes)
```

输出如下：

```
{'AAPL': 231.41, 'TSLA': 269.19, 'IBM': 214.67}
```

2. 字典的基本操作

字典的基本操作包括访问字典的值、添加或更新键-值对、删除键-值对等操作。

1）访问字典的值

代码如下：

```
apple_close = stock_info["AAPL"]["close"]
print(f"AAPL 的价格是: {apple_close}")
```

输出如下：

```
AAPL 的价格是: 231.41
```

2）添加键-值对或更新字典值

代码如下：

```
#更新价格
stock_info["AAPL"]["close"] = 267
#添加新的股票
stock_info["GOOGL"] = {"close": 2800, "volume": 300}
print(stock_info)  #增加打印
```

输出如下：

```
{'AAPL': {'close': 267, 'volume': 1000},
'TSLA': {'close': 269.19, 'volume': 500},
'GOOGL': {'close': 2800, 'volume': 300}}
```

3）删除键-值对

可以使用 del 删除字典中的键-值对。如果要删除苹果股票相关信息，则代码如下：

```
del stock_info['AAPL']
print(stock_info)
```

输出如下：

```
{'TSLA': {'close': 269.19, 'volume': 500}, 'GOOGL': {'close': 2800, 'volume': 300}}
```

4）遍历字典

一是遍历字典的键、值或键-值对，代码如下：

```
#第 3 章/dict.py
stock_info = {'AAPL': {'close': 267, 'volume': 1000},
              'TSLA': {'close': 269.19, 'volume': 500},
              'GOOGL': {'close': 2800, 'volume': 300}}
for ticker, info in stock_info.items():
    print(f"股票: {ticker}, 收盘价: {info['close']}, 成交量: {info['volume']}")
```

输出如下：

```
股票: AAPL, 收盘价: 267, 成交量: 1000
股票: TSLA, 收盘价: 269.19, 成交量: 500
股票: GOOGL, 收盘价: 2800, 成交量: 300
```

二是遍历字典中的所有键，代码如下：

```
#遍历所有键
for ticker in stock_info.keys():
    print(ticker)                          #输出股票代码
```

输出如下：

```
AAPL
TSLA
GOOGL
```

三是遍历字典中的所有值，代码如下：

```
for info in stock_info.values():
    close_price = info['close']            #获取收盘价
    volume = info['volume']                #获取成交量
    print(f"收盘价: {close_price}, 成交量: {volume}")
```

输出如下：

```
收盘价: 267, 成交量: 1000
收盘价: 269.19, 成交量: 500
收盘价: 2800, 成交量: 300
```

综上，字典数据类型的特点包括无序性、可变性和唯一键，非常适合股票数据处理。

3.3 控制结构

计算机编程中的控制结构主要有顺序结构、选择结构和循环结构。这 3 种结构是构成程序逻辑的基础，Python 语言也不例外。

3.3.1 顺序结构

顺序结构是程序执行时按顺序从上到下逐行执行的结构。在顺序结构中，所有操作按给定顺序执行，没有分支或循环。这种结构适合简单的逻辑和流程，而复杂逻辑通常需要结合选择和循环结构。

本节使用简单的顺序结构来计算单个交易日的收益率。假设策略只处理两天的价格数据，通过简单的方法来计算收益率，代码如下：

```
price_1 = 100                                    #第 1 天的价格
price_2 = 102                                    #第 2 天的价格
daily_return = (price_2 - price_1) / price_1     #计算每日收益率
print("每日收益率:", daily_return)
```

输出如下：

```
每日收益率: 0.02
```

3.3.2 选择结构

选择结构用于根据条件的不同执行不同的代码块，而 if 语句就是实现选择结构的一种方式，其基本语法结构如下：

```
if 条件:
    #当条件为真时执行的代码
elif 另一个条件:
    #当另一个条件为真时执行的代码
else:
    #当所有条件都不满足时执行的代码
```

其中，if、elif 和 else 语句用于条件判断，根据条件的真假来执行不同的代码块。if 语句用于判断一个条件是否为真。如果条件为真，则执行 if 语句下的代码块；elif 是"else if"的缩写，用于在前面的 if 或 elif 条件不成立时，判断另一个条件。如果 elif 条件为真，则执行其下的代码块。在 if 和 elif 条件都不成立时执行 else 语句的代码块。

下面是一个使用 if 语句的简单量化金融例子，用于判断投资信号，代码如下：

```
#第 3 章/example01.py
#假设有一个 RSI 值
rsi = 28                #示例 RSI 值
#判断买入信号
if rsi < 30:
    signal = "买入"
elif rsi > 70:
    signal = "卖出"
else:
    signal = "持有"
```

```
#输出信号
print("交易信号:", signal)
```

输出如下：

```
交易信号:买入
```

在以上模拟的策略场景中，通过检查 RSI 值并与阈值对比来产生交易与否信号。这种结构就是典型的通过 if-elif-else 来进行条件判断的选择结构。

3.3.3 循环结构

Python 提供了两种基本的循环语句：for 循环和 while 循环。它们用于重复执行一段代码，直到满足某个条件。

1. for 循环

for 循环用于遍历一个序列。它会依次访问序列中的每个元素，并对每个元素执行相同的代码块。for 循环是一种高效且简洁的控制结构，循环变量在每次迭代中自动更新，不需要手动控制，适合用于已知迭代次数的场景，如遍历固定长度的列表。

假设以 for 循环来计算股票的每日收益率，并已获取过去 10 日的收盘价数据，现在需要算出每日收益率，代码如下：

```
#第 3 章/forloop.py
#定义过去 10 日的收盘价
closing_prices = [100, 102, 101, 105, 107, 106, 108, 110, 109, 111]

#计算每日收益率
daily_returns = []
#遍历收盘价列表，从第 2 天开始计算每天收益率并将其存储在 daily_returns 列表
for i in range(1, len(closing_prices)):
    daily_return = (closing_prices[i] - closing_prices[i - 1]) / closing_prices[i - 1]
    daily_returns.append(daily_return)

#输出结果
for day in range(1, len(closing_prices)):
    print(f"第{day + 1}天, 收益率: {daily_returns[day - 1]:.2 %}")
```

输出如下：

```
第 2 天, 收益率: 2.00 %
第 3 天, 收益率: -0.98 %
第 4 天, 收益率: 3.96 %
第 5 天, 收益率: 1.90 %
第 6 天, 收益率: -0.93 %
第 7 天, 收益率: 1.89 %
第 8 天, 收益率: 1.85 %
第 9 天, 收益率: -0.91 %
第 10 天, 收益率: 1.83 %
```

2. while 循环

while 循环是 Python 中的一种循环控制结构，用于在特定条件为真时重复执行一段代码。在条件控制中，循环会一直执行，直到条件变为假，因此 while 循环可能会导致无限循环：如果条件始终为真而没有适当的退出机制，则将导致无限循环，所以可在循环中使用 break 语句提前退出，或使用 continue 语句跳过当前迭代。while 循环的基本语法如下：

```
while condition:
    #代码块
```

以下是一个使用 while 循环来计算股票收益率的简单示例。假设目前有一组股票的每日收益率，策略希望计算累计收益率，直到达到一个设定的目标收益率为止，代码如下：

```
#第 3 章/whileloop.py
#假设每日收益率
daily_returns = [0.01, 0.02, -0.01, 0.03, 0.01, -0.02, 0.04]
target_return = 0.05              #目标收益率
cumulative_return = 0.0
days = 0

#使用 while 循环计算累计收益率
while days < len(daily_returns) and cumulative_return < target_return:
    cumulative_return += daily_returns[days]
    days += 1
#输出结果
print(f"达到目标收益率所需的天数: {days}")
print(f"累计收益率: {cumulative_return:.2%}")
```

输出如下：

```
达到目标收益率所需的天数: 5
累计收益率: 6.00%
```

在以上代码中，while 循环的作用是：第一，在还没有达到目标收益率且还有数据可用的情况下，不断累加每日收益率。第二，每次循环会更新累计收益率并推进到下一天。当循环结束时，cumulative_return 将达到或超过目标收益率，或已遍历所有收益率数据。

如果仔细观察，则大家会发现累计收益率是通过加法计算的，但这样计算会导致偏差。因为简单地将每日收益率相加，没有考虑收益再投资的影响。在实际投资中，收益会不断再投资，这种算法将忽视实际收益随时间复利增长，而且，如果某一天有负收益，则简单加法可能会导致累计收益率看起来更乐观。

假设投资第 1 天的收益率为 5%，第 2 天的收益率为−5%，用加法来算累计收益率，其结果为 0，而在实际投资中，真实收益率为 1×(1+5%)×(1−5%)=0.9975，累计收益率为−0.25%，因此可以对以上算法进行改进，代码如下：

```
#第 3 章/qexample.py
#每日收益率
daily_returns = [0.01, 0.02, -0.01, 0.03, 0.01, -0.02, 0.04]
```

```
target_return = 0.05                    #目标收益率
cumulative_return = 1.0                 #初始化为 1
days = 0

#使用 while 循环计算累计收益率(复利)
while days < len(daily_returns) and (cumulative_return - 1) < target_return:
    cumulative_return * = (1 + daily_returns[days])       #使用乘法计算复利
    days += 1

print(f"达到目标收益率所需的天数: {days}")
print(f"累计收益率: {cumulative_return - 1:.2%}")         #减去 1 得到累计收益率
```

输出如下：

```
达到目标收益率所需的天数: 4
累计收益率: 5.05%
```

与累加算法对比，通过累乘计算的收益率更快达到目标，与投资现实更加符合。

3. break 和 continue

break 和 continue 是 Python 中的两种循环控制语句，用于改变循环的执行流程，其中，break 语句用于立即终止循环，不管循环条件是否仍为 True 或序列是否已遍历完。控制流程跳出循环，继续执行循环后的代码。与 break 语句不同的是，continue 语句结束这次循环，继续进行剩下的循环，其中，continue 语句的语法如下：

```
for item in iterable:
    if condition:
        continue                        #跳过当前迭代,继续下一个迭代
```

break 语句的语法如下：

```
for item in iterable:
    if condition:
        break                           #退出循环
```

下面以一个通俗的例子来说明两者的区别。假设在筛选一堆橙子，只保留新鲜的橙子。当遇到一个坏橙子时，并不继续处理它，而是立刻跳过它，继续检查下一个橙子。

代码如下：

```
#第 3 章/continue_exam.py
oranges = ["新鲜橙子", "坏橙子", "新鲜橙子"]
for orange in oranges:
    if orange == "坏橙子":
        continue                        #跳过这个坏橙子
    print(f"挑选了 {orange}")           #处理新鲜的橙子
```

输出如下：

```
挑选了 新鲜橙子
挑选了 新鲜橙子
```

以上表明,在 continue 语句中,可以从一堆橙子中筛选出好橙子。

break 语句:从一堆水果中挑选出新鲜的橙子,并且决定只要找到一个坏橙子,就停止整个挑选过程,因为一旦发现坏橙子,就觉得这个水果堆里的橙子可能都不好,代码如下:

```
#第 3 章/whilexam.py
oranges = ["坏橙子", "新鲜橙子" ,"新鲜橙子"]

for orange in oranges:
    if orange == "坏橙子":
        print("发现坏橙子,停止挑选!")
        break                          #停止整个循环
    print(f"挑选了 {orange}")          #处理新鲜的橙子
```

输出如下:

```
发现坏橙子,停止挑选
```

以上表明,在 break 语句中,只要发现一个坏橙子,就立即停止整个循环。

综上,continue 遇到特定条件时跳过当前迭代,继续进行后续工作。break 在遇到特定条件时立即终止整个循环,停止所有后续工作。这两种机制的选择取决于程序的目标:是希望继续处理其他项,还是希望一旦发现问题就停止所有操作。

3.4 函数

函数是具有名字的代码块,用于执行特定任务,在编程中具有非常重要的作用。一是复用性。在 Python 程序中,如果需要反复调用同一项任务,则可以通过调用函数避免重复编写代码,提高运行效率。二是逻辑分离。通过将复杂的任务拆分成多个小的易于管理的函数来使程序的逻辑结构变得更加清晰。每个函数专注于一个特定的功能,使代码更易于理解和维护。三是提高可读性。函数使代码更具可读性,使开发者在阅读代码时,能够快速地理解函数的用途和实现逻辑。四是作用域管理。函数提供了局部作用域,这样函数内部定义的变量不会污染全局命名空间,使代码更加安全,避免了意外修改全局变量的风险。五是支持抽象和封装。函数可以作为抽象层,隐藏实现细节。调用者只需关注如何使用函数,而不必关心内部实现。这种封装使代码更加灵活,易于扩展和修改,因此函数在编程中具有非常重要的地位,一定要熟悉掌握其基本原理和使用方法。

3.4.1 函数定义

1. 定义函数

在每次交易前首先应该获得当日的时间,以为后期处理做准备。函数的语法如下:

```
def function_name(parameters):
    """文档字符串"""
    #函数体
    return value
```

在 Python 语言中，datetime 模块提供了用于处理日期和时间的类和函数，代码如下：

```
#第 3 章/date_exam.py
import datetime
def get_current_time():
    """
    获取当前时间并返回格式化的字符串。
    返回:
        str: 当前时间的字符串,格式为 YYYY-MM-DD HH:MM:SS
    """
    current_time = datetime.datetime.now()                    #获取当前时间
    #print(current_time)
    return current_time.strftime("%Y-%m-%d %H:%M:%S")

#示例调用
if __name__ == "__main__":
    print("当前时间:", get_current_time())
```

输出如下：

```
当前时间: 2024-10-27 09:18:57
```

以上函数提供了一个简单的方式来获取当前时间，并以标准的格式输出，适用于需要记录时间的场景，例如在量化交易中记录交易执行时间、日志时间戳等，其中，首先使用 strftime 方法将 current_time 格式转化为字符串，格式为“YYYY-MM-DD HH:MM:SS”，然后将这个格式转化后的字符串作为函数的返回值，使时间格式规整。

2. 向函数传入信息

假设现在需要设计一个 calc_return()函数来计算股票的投资回报率。这个函数需要接收初始投资金额、股票的初始价格和结束价格这 3 个参数，返回投资收益率，代码如下：

```
#第 3 章/calc_return.py
def calc_return(investment, start_price, end_price):
    """
    计算股票的投资回报率。

    参数:
        investment (float): 初始投资金额
        start_price (float): 股票的初始价格
        end_price (float): 股票的结束价格

    返回:
        float: 投资回报率(百分比)
    """
    #计算购买的股票数量
    shares = investment / start_price
    #计算最终投资价值
    final_value = shares * end_price
    #计算投资回报率
```

```
    return_rate = (final_value - investment) / investment * 100

    return return_rate

#示例调用
if __name__ == "__main__": #判断模块是否作为主程序执行
    inv = 10000                #初始投资金额
    sp = 50                    #股票的初始价格
    ep = 75                    #股票的结束价格

    r = calc_return(inv, sp, ep)
    print(f"投资回报率: {r:.2f} % ")
```

输出如下：

```
投资回报率: 50.00 %
```

3.4.2 参数类型

1. 形参

形参是在定义函数时声明的变量，用来接收调用时传入的值。它们相当于占位符，指示函数需要什么样的输入才能运行。当函数被调用时，实参的值会被赋给对应的形参。

假设需要不同种类的水果来制作一个水果沙拉，则函数的代码如下：

```
#第 3 章/fruit_para.py
def make_fruit_salad(apple, banana, orange):
    """
    制作水果沙拉。

    参数:
        apple (int): 苹果的数量
        banana (int): 香蕉的数量
        orange (int): 橙子的数量

    返回:
        str: 水果沙拉的描述
    """
    return f"水果沙拉包含 {apple} 个苹果,{banana} 个香蕉和 {orange} 个橙子。"
```

在以上代码中，make_fruit_salad()中的 apple、banana 和 orange 是形参，它们是占位符，表示函数需要的输入材料。

2. 实参

实参是在调用函数时传递给函数的实际值。它们是在函数被调用时提供给函数的输入数据。实参可以是常量、变量或表达式，它们的值将被用于函数内部的计算。

在做水果沙拉的示例中，当调用这个函数时，需要提供具体的水果数量，代码如下：

```
salad_description = make_fruit_salad(3, 3, 2)
```

在这个调用中：3、3 和 2 是实参，分别对应于苹果、香蕉和橙子的数量。这些实参的值会被传递给对应的形参。

综上，在函数中，形参是函数定义时的变量名，用于接收函数调用时传入的值。实参是在调用函数时实际传递的值。

3.4.3 参数传递

1. 位置参数

位置参数是指按照函数定义的参数顺序传递给函数的实际参数。在调用函数时，实参的顺序必须与形参的顺序相匹配，否则会报错，语法如下：

```
def function_name(param1, param2, ...):
    #函数体
    pass
#在调用函数时使用位置参数
function_name(value1, value2, ...)
```

以下以投资收益率函数为例来演示，代码如下：

```
#第 3 章/invest01.py
def calculate_invest_return(start_value, end_value):
    """
    参数:
        start_value (float): 投资的开始值
        end_value (float): 投资的终值
    返回:
        float: 投资收益率(百分比)
    """
    return_rate = ((end_value - start_value) / start_value) * 100
    return return_rate

#示例调用
if __name__ == "__main__":
    start_investment = 10000            #投资的开始值
    end_investment = 12000              #投资的终值
    roi = calculate_invest_return(start_investment, end_investment)
    print(f"投资收益率: {roi:.2f}%")
```

输出如下：

```
投资收益率: 20.00%
```

在 calculate_invest_return()函数中，start_value 和 end_value 是位置参数，它们在函数定义时指定，表示投资的开始值和终值。在调用函数时，必须按照定义时的顺序传递实参。roi=calculate_invest_return(10000, 12000)用于按照顺序传入实参值。

2. 关键字参数

关键字参数允许在调用函数时通过参数名指定实参值，而不必关心它们的顺序。这使函数调用更加清晰和灵活。基本语法结构如下：

```
def function_name(param1 = value1, param2 = value2, ...):
    #函数体
    pass
function_name(param1 = new_value1, param2 = new_value2)
```

假设继续以投资收益率计算为例，以关键字参数进行函数编写，代码如下：

```
#第 3 章/invest02.py
def calculate_invest_return(start_value = None, end_value = None):
    """
    参数:
        start_value (float): 投资的开始值
        end_value (float): 投资的终值
    返回:
        float: 投资收益率(百分比)
    """
    if start_value is None or end_value is None:
        raise ValueError("start_value 和 end_value 不能为空。")

    return_rate = ((end_value - start_value) / start_value) * 100
    return return_rate

#示例调用
if __name__ == "__main__":
    start_investment = 10000                #投资的开始值
    end_investment = 13000                  #投资的终值

    #使用关键字参数
    roi = calculate_invest_return(start_value = start_investment, end_value = end_
investment)
    print(f"投资收益率: {roi:.2f}%")
```

输出如下：

```
投资收益率: 30.00%
```

3. 默认值参数

在 Python 语言中，可使用默认参数给函数参数赋值。如果在调用函数时没有为默认参数提供具体的值，则 Python 将自动使用默认值。

假设有一个用于计算贷款的月供函数，可为利率和贷款期限设置默认值，代码如下：

```
#第 3 章/invest03.py
def calculate_monthly_payment(principal, a_i_rate = 5.0, loan_term = 30):
    """
    计算贷款的月供。
```

```
    :param principal: 贷款本金(单位: 元)
    :param a_i_rate: 年利率(默认值为 5.0%)
    :param loan_term: 贷款期限(默认值为 30 年)
    :return: 每月还款金额(单位: 元)
    """
    monthly_interest_rate = a_i_rate / 12 / 100
    total_payments = loan_term * 12
    monthly_payment = (principal * monthly_interest_rate) / (1 - (1 + monthly_interest_
rate) ** -total_payments)
    return monthly_payment
print(calculate_monthly_payment(1000000, 4.5, 20))
```

输出如下：

```
6326.493762199708
```

以上代码的 calculate_monthly_payment()函数有 3 个参数，其中，a_i_rate 和 loan_term 为默认值参数，默认参数值分别为 5.0%和 30 年。

3.4.4 返回值

函数可用于处理一些数据并相应地返回一个或一组值，该值即为返回值。函数一般通过 return 来返回值。该操作让复杂的操作封装在函数内部，只显示操作值，简化了主程序。

1. 返回简单值

返回简单值是最常见的情况，例如整数、浮点数、字符串等。函数通过 return 语句将数据类型的值返给调用者。假设以加法运算来展示返回简单值，代码如下：

```
def add_numbers(a, b):
    return a + b
result = add_numbers(5, 3)
print(result)
```

输出如下：

```
8
```

2. 返回字典

字典是一种包含键-值对的复杂数据结构，Python 中的函数也可以返回字典，以便将多个相关的数据一起返回，代码如下：

```
#第 3 章/re_dict.py
def create_person(name, age):
    return {
        'name': name,
        'age': age
    }
person_info = create_person('Alice', 30)
print(person_info)
```

输出如下：

```
{'name': 'Alice', 'age': 30}
```

3. 返回其他类型

除了简单值和字典类型数据，函数也可返回其他类型的数据，如列表、集合、元组等类型数据，代码如下：

```
def get_odd_numbers(numbers):
    return [num for num in numbers if num % 2 != 0]
odd_numbers = get_odd_numbers([1, 2, 3, 4, 5, 6])
print(odd_numbers)
```

输出如下：

```
[1, 3, 5]
```

在以上代码中，get_odd_numbers()函数接收一个数字列表 numbers，并通过列表推导式来筛选出其中的奇数，函数返回值是一个新的列表[1,3,5]。

3.4.5 匿名函数

匿名函数是指没有名称的函数，通常称为 lambda()函数。它们可以用于快速定义简单的函数，而无须使用 def 关键字来显式声明函数。lambda()函数是 Python 的一种表达式，而不是语句。它们用于创建小型的一次性的函数对象。由于其简单性和灵活性，其常用于需要函数作为参数的场景，例如排序、过滤等。

1. 定义

匿名函数使用 lambda 关键字定义，其语法如下：

```
lambda arguments: expression
```

lambda 是关键字，用于定义匿名函数；arguments 是传递给函数的参数，可以有多个，用逗号分隔。expression 是一个表达式，计算结果即为函数的返回值。这个表达式不能包含多行代码。假设需要用匿名函数来计算 x 的平方，代码如下：

```
square = lambda x: x ** 2
print(square(5))
```

输出如下：

```
25
```

2. 用在 sorted()函数里进行排序

匿名函数可用在 sorted()函数里解决排序问题，代码如下：

```
data = [(1, 2), (3, 1), (5, 0)]
sorted_data = sorted(data, key = lambda x: x[1])
print(sorted_data)
```

输出如下：

```
[(5, 0), (3, 1), (1, 2)]
```

3. 用在 map()函数中

map()函数用于将一个函数应用到一个可迭代对象(如列表)的每个元素上，并返回一个新的迭代器。假设通过 map()函数对列表元素进行数学运算，代码如下：

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)
```

输出如下：

```
[1, 4, 9, 16, 25]
```

通过上述案例可知，lambda()在 Python 中提供了一种简单而强大的方法来定义临时的小型函数。它的简洁性和灵活性使代码更清晰、更易于管理，尤其在高阶函数的使用场景中，能简化代码逻辑。

4. 用在 filter()函数中

filter()函数用于过滤可迭代对象中的元素，只保留那些使函数返回 True 的元素。假设匿名函数可运用 filter()函数对列表元素进行筛选，代码如下：

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,12]
even_numbers = filter(lambda x: x % 2 == 0, numbers)          #筛选出偶数
print(list(even_numbers))
```

输出如下：

```
[2, 4, 6, 8, 10,12]
```

3.5 模块与包

模块(Module)和包(Package)是 Python 组织代码的重要工具，它们提高了代码的可维护性、可重用性和可扩展性。

3.5.1 模块的概念

模块是 Python 中的一个文件，其文件扩展名为 .py，文件中包含 Python 代码。模块可以包含函数、类和可执行的代码块，能够帮助开发者将代码组织成逻辑上相关的组件。模块的优势在于通过代码重用、代码组织和封装显著地提升代码的写作效率。

假设有一个文件 maths.py，其主要功能是实现加、减、乘、除运算，代码如下：

```
#第 3 章/maths.py
def add(a, b):
    """返回两个数的和"""
    return a + b

def subtract(a, b):
    """返回两个数的差"""
    return a - b

def multiply(a, b):
    """返回两个数的积"""
    return a * b

def divide(a, b):
    """返回两个数的商。如果除数为 0,则返回 '除数不能为 0' 的错误提示信息"""
    if b == 0:
        return '除数不能为 0'
    return a / b
```

其中,math.py 是一个模块,它定义了函数 add、subtract、multiply、divide。

3.5.2 导入模块

要在 Python 程序中使用模块,可以使用 import 语句。import 语句允许开发者将模块的功能导入当前的代码中。接下来通过引入 3.5.1 节定义的模块来展示导入模块的方法。

1. 基本导入

使用 import 语句导入整个模块,并通过模块名来访问其中的函数,代码如下:

```
import maths
result = maths.add(5, 3)
print(result)
```

输出如下:

```
8
```

2. 导入模块特定函数

使用 from-import 语句导入模块中特定函数,代码如下:

```
from maths import add
result = add(5, 3)
print(result)
```

输出如下:

```
8
```

3.5.3 创建模块

创建模块非常简单，只需创建一个 .py 文件并在文件中编写 Python 代码。可以在模块中定义函数、类、变量等，并在其他模块或脚本中导入使用。

1. 创建模块

假设创建了问候语 Hi 和告别语 Goodbye，函数根据传入的名字进行问候和告别，代码如下：

```
#第 3 章/modul.py
def greet(name):
    """返回问候语"""
    return f"Hi, {name}!"

def farewell(name):
    """返回告别语"""
    return f"Goodbye, {name}!"
```

2. 调用模块

在 Python 程序中，引入 greeting 模块，可以通过调用 greet()和 farewell()函数来对 Mary 和 Bob 分别问候和告别，代码如下：

```
from modul import greet, farewell
print(greet("Mary"))
print(farewell("Bob"))
```

输出如下：

```
Hi, Mary!
Goodbye, Bob!
```

在 Python 语言中，无论是自己定义的模块，还是外部模块，调用方法是相同的。

3.5.4 Python 包

模块(Module)是一个 Python 文件，而包(Package)则是一个目录，它包含多个模块和子包。包是 Python 模块系统的一种层次化组织形式。一个包本质上就是一个包含多个模块(.py 文件)的目录，并且该目录下一般有一个名为__init__.py 的文件(即使是空文件)，该文件用于标识该目录是一个 Python 包。

假设现在有一个数学计算包 maths_p，其包含 add.py、subtract.py 等多个模块，以及一个包的初始化文件__init__.py，包的目录如下：

```
maths_p/
|
├── __init__.py
```

```
├── add.py
├── subtract.py
├── multiply.py
└── divide.py
```

1. __init__.py

表明这是一个 Python 包，可以包含初始化代码。在某些情况下，即使包中没有内容，也可创建空的 __init__.py 文件，表明该目录是一个包。

2. 创建包的模块

由于创建包的模块方法与之前相同，仅仅展示 add.py 中的 add 函数，代码如下：

```
#第 3 章/maths_p.py
#maths_p/add.py
def add(a, b):
    """返回两个数的和"""
return a + b

#maths_p/subtract.py
def subtract(a, b):
    """返回两个数的差"""
return a - b

#maths_p/multiply.py
def multiply(a, b):
return a * b

#maths_p/divide.py
def divide(a, b):
    if b == 0:
        raise ValueError("除数不能为零")
return a / b
```

3. 导入包和模块

在 Python 中可以在主程序中使用 import 语句导入包中的所有模块和函数，代码如下：

```
#第 3 章/package.py
#在主程序中导入整个包
"""
主程序：演示如何使用 maths_p 包
"""

#导入整个包
from maths_p import add,divide,multiply,subtract

def main():
    print(" === 数学运算包演示 === \n")
```

```
    #使用包中的函数
    a, b = 10, 4

    #加法
    result_add = add(a, b)
    print(f"加法: {a} + {b} = {result_add}")
    #减法
    result_subtract = subtract(a, b)
    print(f"减法: {a} - {b} = {result_subtract}")
    #乘法
    result_multiply = multiply(a, b)
    print(f"乘法: {a} × {b} = {result_multiply}")
    #除法
    try:
        result_divide = divide(a, b)
        print(f"除法: {a} ÷ {b} = {result_divide}")
    except ValueError as e:
        print(f"除法错误: {e}")


if __name__ == "__main__":
    main()
```

输出如下：

```
=== 数学运算包演示 ===

加法: 10 + 4 = 14
减法: 10 - 4 = 6
乘法: 10 × 4 = 40
除法: 10 ÷ 4 = 2.5
```

在复杂的项目中，可通过包中包含子包的方式来组织多层次的包和模块。总之，通过包的组织，代码的组织性、可维护性、可重用性更强，可实现代码的高效管理。

3.5.5 Python 标准库

Python 的标准库是非常丰富的模块集合，提供了大量的内置功能和模块，用于处理各种常见任务，如文件 I/O、网络编程、数据处理等，例如，标准库模块有 datetime 模块、os 模块等。

1. os 模块

提供与操作系统交互的功能，如文件和目录操作。返回当前工作目录的路径，代码如下：

```
import os
current_directory = os.getcwd()
print(f"当前工作目录是: {current_directory}")
```

输出如下：

```
当前工作目录是: c:\Users\yourname\Desktop
```

2. 将当前工作目录修改为指定路径

可以通过 os.chdir(path)函数将当前工作目录修改为指定路径，代码如下：

```
import os
#将工作目录修改为 /tmp
os.chdir('D:/Python')
print(f"改变后的工作目录是: {os.getcwd()}")
```

输出如下：

```
改变后的工作目录是: D:\Python
```

3. 列出目录内容

os.listdir(path)函数可以返回指定目录中的文件和目录列表，代码如下：

```
#第 3 章/oslist.py
import os
#使用当前目录作为示例，也可以修改为其他存在的目录路径
directory_path = '.' #当前目录
#或者使用 Windows 风格的绝对路径，例如: directory_path = r'C:\Users\csz\Desktop\配套代码'

try:
    directory_contents = os.listdir(directory_path) #获取目录文件和目录列表

    #打印目录内容
    print(f"目录 '{directory_path}' 的内容:")
    for item in directory_contents:
        print(item)

except FileNotFoundError:
    print(f"错误: 目录 '{directory_path}' 不存在")
except PermissionError:
    print(f"错误: 没有权限访问目录 '{directory_path}'")
except Exception as e:
    print(f"发生未知错误: {e}")
```

输出如下：

```
目录 '.' 的内容:
.vscode
ch10
ch11
ch12
ch2
ch3
```

```
ch4
ch5
ch6
ch7
ch8
ch9
...
```

3.6 文件操作

文件操作是编程中常见的任务，包括文件的创建、读取、写入和删除等。Python 提供了一组功能强大的内置函数和方法来执行这些操作。

3.6.1 文件读写

文件读写是指从文件中读取数据和向文件中写入数据。Python 提供了内置的 open() 函数来打开文件，并提供了相应的方法来读取和写入文件内容。

1. 打开文件读取

open()函数可打开文件并返回文件对象，此外还可指定文件的路径、打开模式，代码如下：

```
file = open('example.txt', 'r')
content = file.read()                    #读取整个文件内容
print(content)
file.close()                             #关闭文件
```

2. 写入文件内容

write()方法用于将指定的字符串写入打开的文件中，代码如下：

```
file = open('example.txt', 'w')
file.write('Hello, World!')              #将字符串写入文件
file.close()                             #关闭文件
```

需要注意的是，如果 example.txt 已存在，则文件内容会被完全清空，仅保留新写入的内容'Hello，World !'；如果 example.txt 不存在，则会新建文件并写入内容，所以，一般情况下，为了保证文件原来的内容不被删除，需要使用追加模式'a'。

3. 将内容追加到文件

追加可通过 with open as file 这种方式实现，语法如下：

```
file_path = 'example.txt'
with open('example.txt', 'a') as file:
    file.write("\nAppended text.")
```

(1) 'file_path' 是文件的路径，可以是相对路径或绝对路径。

(2) 'mode' 是打开文件的模式,a 表示追加模式。

(3) file 是文件对象的变量名,可在 with 语句块中访问文件内容。

(4) with 语句用于确保在文件操作完成后自动关闭文件,不需要显式调用 file.close()。这种方式确保即使发生异常,文件也能正常关闭,避免资源泄露,代码如下:

```
with open('example.txt','a') as f:
    f.write('\n How are you?')
```

4. 读取文件

可以使用 readlines()方法,一次性读取所有行,并将它们作为一个列表返回。每行都会被当作列表中的一个元素,包括换行符(\n)。假设 example.txt 文件中的内容如下:

```
Hello, Everyone!
Welcome to Quant World.
This is a test file.
```

代码如下:

```
with open('example.txt', 'r') as file:
    lines = file.readlines()
    print(lines)
```

输出如下:

```
['Hello,Everyone!\n', 'Welcome to Quant World.\n', 'This is a test file.\n']
```

但如果想要去掉换行符,只显示文本,则需进一步处理,代码如下:

```
with open('example.txt', 'r') as file:
    lines = file.readlines()
    for i in range(len(lines)):
        lines[i] = lines[i].rstrip('\n')
    print(lines)
```

输出如下:

```
['Hello,Everyone!', 'Welcome to Quant World.', 'This is a test file.']
```

3.6.2 文件模式

在打开文件时,必须指定文件模式。文件模式定义了文件的打开方式,如读取、写入、追加等,常见的文件模式及其功能如表 3-1 所示。

表 3-1 常见的文件模式及其功能

模 式	描 述	操 作
'r'	只读模式(默认)	读取
'w'	写入模式(会截断文件)	写入
'a'	追加模式(在文件末尾写入)	追加

续表

模　式	描　述	操　作
'b'	二进制模式	二进制
't'	文本模式(默认)	文本
'+'	读写模式	读写

3.7 错误和异常

了解并掌握 Python 程序在运行的过程中异常的概念及处理方法,可提高代码的健壮性和容错能力。

3.7.1 异常概念与处理

异常是指在程序的执行过程中发生的错误事件,这些事件会导致程序的正常执行流程被中断。如果异常没有被捕获和处理,则程序会终止并显示错误信息。常见的异常包括文件未找到、除零错误、值错误等。Python 提供了一个强大的异常处理机制,可以捕获和处理这些异常,避免程序崩溃。

Python 使用 try-except 块来捕获和处理异常,可以选择性地使用 else 和 finally 块。假设通过 try-except 块来处理文件操作中的异常,代码如下:

```
#第 3 章/exception.py
file = None
try:
    file = open('example.txt', 'r')
    content = file.read()
    print(content)
except FileNotFoundError:
    print("文件未找到")
except IOError:
    print("文件读写错误")
finally:
    #确保文件被关闭
    if file:
        file.close()
```

输出如下:

```
文件未找到
```

其中,try 块中的代码尝试打开和读取文件。如果文件不存在,则会引发 FileNotFoundError 异常,并执行 except 块中的代码进行处理。如果发生其他输入/输出错误,则会引发 IOError 异常,并执行相应的处理代码。无论是否发生异常,finally 块中的代码都会被执行,用于确保文件被正确关闭。总之,通过 try-except 保证了程序的稳健性,确保在异常发生时能够合理地进行处理,而不是让程序崩溃。这对开发健壮的程序非常重要。

3.7.2 常见异常类型

Python 提供了多种内置异常类型来表示不同类型的错误。常见异常类型包括零除错误 ZeroDivisionError、文件未找到错误 FileNotFoundError、值错误 ValueError、类型错误 TypeError、索引错误 IndexError 等。假设通过一个函数来处理零除错误，代码如下：

```
#第 3 章/exception02.py
def example_function():
    try:
        #获取用户输入
        num1 = float(input("请输入被除数: "))
        num2 = float(input("请输入除数: "))
        result = num1 / num2                #可能触发 ZeroDivisionError
        print(f"计算结果: {result}")
    except ZeroDivisionError:
        print("捕获到: 零除错误(除数不能为 0)")
    finally:
        print("程序执行结束(finally 块始终执行)")
example_function()
```

运行程序后，如果输入的除数为 0，则输出如下：

```
捕获到: 零除错误(除数不能为 0)
程序执行结束(finally 块始终执行)
```

3.8 面向对象编程

面向对象编程(Object Oriented Programming，OOP)是将数据和对数据的操作组织在一起的方法。通过类和对象，可以更好地组织代码，模型化现实世界的问题。本节将简要地介绍 Python 中的面向对象编程的基本概念和方法。

3.8.1 类和对象

类是定义对象的模板，它包含了对象的属性和方法。对象是类的实例，具体表示了类的特定实现。以下定义一个 Stock 类来表示股票，并创建具体的股票对象。

假设有一种策略定义股票类来管理股票价格和股票名字相关信息，代码如下：

```
#第 3 章/stockclass.py
class Stock:
    #类属性，代表股票所属的市场
    market = "US Stock Market"

    def __init__(self, symbol, close):
        #实例属性，股票代码
        self.symbol = symbol
        #实例属性，股票收盘价
```

```
            self.close = close

        def print_info(self):
            #打印股票信息,包括股票所属市场、股票代码和收盘价
            print(f"股票代码: {self.symbol}, 收盘价: ${self.close:.2f}")

    apple_stock = Stock("AAPL", 150.25)
    apple_stock.print_info()
```

输出如下：

```
股票代码: AAPL, 收盘价: $150.25
```

在以上代码中,Stock 类定义了股票的代码、收盘价等信息,方法为打印出股票信息。

(1) 类属性 market：在类内部,但在类方法之外,它是类的所有实例共享的属性。

(2) 类的初始化方法__init__()：用于初始化实例的属性,接收两个参数 symbol 和 close,并将它们分别赋值给实例属性 self.symbol 和 self.close。

(3) print_info()方法：打印股票相关信息,包括股票代码和收盘价。

(4) Stock("AAPL", 150.25) 创建了一个名为 apple_stock 的 Stock 类实例。

(5) 只要调用 apple_stock.print_info()方法,就会打印 AAPL 相关股票信息。

3.8.2 方法和属性

1. 基本定义和使用

属性可以是实例属性(每个对象独有)或类属性(所有对象共享),而方法则分为实例方法(作用于特定对象)、类方法(作用于类本身)和静态方法(不依赖于类或实例)。通过类的实例化,可以创建具有相同结构和功能的多个对象,从而实现代码的复用和组织。简言之,方法是定义在类中的函数,用于操作对象的数据。属性是存储在对象中的数据,用于描述对象的状态。对之前的类进行完善,代码如下：

```
#第 3 章/stockclass2.py
class Stock:
    market = "Global Stock Market"
    def __init__(self, symbol, price):
        #初始化股票代码
        self.symbol = symbol
        #初始化股票价格
        self.price = price

    def update_price(self, new_price):
        #直接更新股票价格
        self.price = new_price

    def get_price_change(self, old_price):
        #计算价格变化
        return self.price - old_price
```

```
    def __str__(self):
        #返回包含股票代码和当前价格的字符串
        return f"{self.symbol}: ${self.price:.2f}"


#类的实例化
stock = Stock("AAPL", 150.25)
print(stock)

#更新价格并计算涨幅
stock.update_price(155.75)
price_change = stock.get_price_change(150.25)
print(f"价格变化: ${price_change:.2f}")
print(stock)
```

输出如下：

```
ABC: 价格 $100.00
AAPL(科技): 价格 $150.25
价格变化: $5.50
AAPL(科技): 价格 $155.75
```

主要解释如下。

(1) update_price()方法：用于更新股票的价格。

(2) get_price_change()方法：计算并返回股票价格的变化。

(3) __str__()是一种特殊方法：打印 Stock 类的实例时会调用该方法。

2. 增加新的属性

一是动态地给对象添加新的属性，给股票对象添加一个 sector 属性，代码如下：

```
#动态添加新属性
stock.sector = "Technology"
print(f"{stock.symbol} is in the {stock.sector} sector.")
```

输出如下：

```
AAPL is in the Technology sector.
```

二是修改原来实例属性值，可通过实例直接访问来修改，将苹果股价改为 105.12。代码如下：

```
AAPL = Stock("AAPL", 100.00)
AAPL.price = 105.12
print(AAPL)
```

输出如下：

```
$105.12
```

3.8.3 类的继承

类的继承是 Python 面向对象编程的重要功能，其能较好地增强代码的重用性和灵活

性。继承允许一个类继承另一个类的属性和方法，从而形成层次结构。继承方被称为子类，被继承方被称为父类。子类可以重用父类的代码，并可以扩展或完善其功能。

在上例 Stock 类的基础上，定义一个 TechStock 类，它继承自 Stock 类，此外添加行业属性，代码如下：

```
#第 3 章/tradestock.py
class Stock:
    def __init__(self, symbol, price):
        #初始化股票代码
        self.symbol = symbol
        #初始化股票价格
        self.price = price

    def update_price(self, new_price):
        #直接更新股票价格
        self.price = new_price

    def get_price_change(self, old_price):
        #计算价格变化
        return self.price - old_price

    def __str__(self):
        #返回包含股票代码和当前价格的字符串
        return f"{self.symbol}: 价格 ${self.price:.2f}"

class TechStock(Stock):
    def __init__(self, symbol, price, industry):
        #调用父类的 __init__ 方法初始化父类的属性
        super().__init__(symbol, price)
        #初始化子类特有的属性: 行业
        self.industry = industry

    def __str__(self):
        #重写 __str__ 方法,使其包含行业信息
        return f"{self.symbol}({self.industry}): 价格 ${self.price:.2f}"

#创建 Stock 类的实例
regular_stock = Stock("ABC", 100)
print(regular_stock)

#创建 TechStock 类的实例
tech_stock = TechStock("AAPL", 150.25, "科技")
print(tech_stock)

#调用从父类继承的方法
tech_stock.update_price(155.75)
price_change = tech_stock.get_price_change(150.25)
print(f"价格变化: ${price_change:.2f}")
print(tech_stock)
```

输出如下：

```
ABC: 价格 $100.00
AAPL(科技): 价格 $150.25
价格变化: $5.50
AAPL(科技): 价格 $155.75
```

在以上代码中，子类 TechStock 既拥有了父类 Stock 的属性和方法，又添加了自己特有的属性，具体如下：

(1) 在构造函数中调用 super()以初始化父类的属性。

(2) 新增了 self.industry 属性，用于存储科技股票所属的行业。

(3) 重写了__str__方法，在返回的字符串中包含了行业信息。

3.9 本章小结

本章介绍了量化交易所需要掌握的基本 Python 编程知识和原理，主要包括语法结构、函数、模块和包、文件操作及面向对象的编程方式。首先介绍了如何使用包括列表、元组、集合等数据类型来处理金融数据，其次分析了如何采用 3 种不同控制结构进行策略的编写，然后介绍了 Python 各种标准库，以及对量化交易的编程效率的作用。最后，介绍了在量化策略中应多使用面向对象的编程思路来处理量化交易问题，提高代码复用性。