0. 准备工作

使用`os.chdir('C:\\')`设定工作目录，根目录必须使用双斜线`\\`

1. 注释代码

单行注释：以`#`和空格开头，可以注释掉从#开始后面一整行的内容。
多行注释：`"""`三个引号开头，三个引号结尾，通常用于添加多行说明性内容。

2. 数据类型

2.1 数据类型

1. 整型int: Python中可以处理任意大小的整数, 而且支持二进制。
2. 浮点型float: 浮点数即小数。
3. 字符串型str: 字符串是以单引号或双引号括起来的任意文本。
4. 布尔型bool: True or False ## 2.2 变量命名规则 (variable) 硬性规则：
5. 变量名由字母、数字和下划线构成，数字不能开头，不能使用!、@、#等特殊字符。字母指采用Unicode字符，中文、日文、希腊字母等都可以作为变量名中的字符。
6. 大小写敏感。
7. 变量名不能和Python语言的关键字和保留字发生重名的冲突。

2.3 数据类型的转换

- `type()` 识别变量类型
- `int()` 将一个数值或字符串转换成整数，可以指定进制
- `float()` 将一个字符串转换成浮点数。
- `str()` 将指定的对象转换成字符串形式，可以指定编码
- `chr()` 将整数转换成该编码对应的字符串（一个字符）
- `ord()` 将字符串（一个字符）转换成对应的编码（整数）
- `as_integer_ratio()` 将浮点数转换成分数
- `bool()` 将给定的值转换为布尔类型，0被认为是False值，任何非零值或非空对象都被认为是True。

In [ ]:
```
# 2. 数据类型bool()函数

bool(0) # False
bool(1) # True
bool(-1) # True
```

3. 运算符 (operator)

- `[] [:]` 下标，切片（`[:]`选中所有）
- `+ - * ** / % //` 算术运算符（`**`指数运算，`%`求余数，`//`整除）
- `>> <<` 右移，左移
- `~ & ^ |` 按位运算符（`~`按位取反，`&`按位与，`^`按位异或，`|`按位或）
- `<= < > >= == !=` 比较运算符，输出布尔值

- `is, is not` 身份运算符
- `in, not in` 成员运算符
- `not, or, and` 逻辑运算符
- `= += -= *= /= %= //= **= &=` 赋值运算符
- 优先级从上到下

In [ ]:
```
# 3. 运算符

# 按位运算符

# 按位与运算符会将两个数的二进制表示的每一位进行比较，如果两个数的相应位都为1，那么结果的相应位
print(1 & 2)  # 0
# 按位或运算符会将两个数的二进制表示的每一位进行比较，如果两个数的相应位都为0，那么结果的相应位
# 1的二进制为0001，2的二进制为0010。按位或运算得到的结果是0011，对应十进制数为3。
print(1 | 2)  # 3

# 逻辑运算符

# 当 "and" 运算符的两个操作数都为真（非零）时，运算结果为真（非零）；否则，结果为假（0）。
print(1 and 2) # 2
print(1 and 0) # 0
```

4. 占位符 (placeholder)

- `%s`：字符串
- `%d`：整数
- `%f`：浮点数
- `%x`：十六进制数
- `%o`：八进制数
- `f{}`：f-string（格式化字符串字面值）

In [ ]:
```
# 4. 占位符

name = "Tom"
age = 25
height = 1.75

# 使用占位符进行格式化输出
print("My name is %s, I'm %d years old, and my height is %.2f meters." % (name, age, height))

# 使用 f-string 进行格式化输出
# 其中，{height:.2f} 中的 .2 表示保留两位小数
print(f"My name is {name}, I'm {age} years old, and my height is {height:.2f} meters.")
```

```
My name is Tom, I'm 25 years old, and my height is 1.75 meters.
My name is Tom, I'm 25 years old, and my height is 1.75 meters.
```

5. 分支结构

`if, elif, else` 使用了缩进的方式来表示代码的层次结构，连续的代码保持了相同的缩进那么它们属于同一个代码块。

In [ ]:
```
# 5. 分支结构
# 计算二叉树

def option_profit(option_type, price, strike):
    # option_type must be one of 'call' or 'put'
```

```
    if option_type == 'call':
        if price > strike:
            return price - strike
        else:
            return 0
    elif option_type == 'put':
        if price < strike:
            return strike - price
        else:
            return 0
    else:
        print('Review inputs')

option_profit('call', 20, 10)
```

Out[ ]:
```
10
```

6. 循环结构

- `for in` 循环，知道循环执行的次数
- `while` 循环，通过一个能够产生bool值的表达式来控制循环
- `break, continue`，通过可以 `while True` 构造条件恒成立的循环，如果不做特殊处理，循环是不会结束。`break` 关键字可以提前结束循环。需要注意的是，`break` 只能终止它所在的那个循环，`continue` 关键字可以用来放弃本次循环后续的代码直接让循环进入下一轮。

In [ ]:
```
# 6. 循环结构
# 用for循环实现1~100求和

total = 0
for x in range(1, 101):
    total += x
print(total) # 5050
# range(开始，结束，步长)，前面是闭区间，后面是开区间

# 用for循环遍历列表
cashflows = [10, 3, 9, 20, 5]
for item in cashflows:
    print(item * 2)            # 20 6 18 40 10
```

In [ ]:
```
# 6. 循环结构
# while循环猜数字

import random

# 产生一个1-100范围的随机数
answer = random.randint(1, 100)
counter = 0
while True:
    # while True即条件恒成立循环
    counter += 1
    number = int(input('请输入: '))
    if number < answer:
        print('大一点')
    elif number > answer:
        print('小一点')
    else:
        print('恭喜你猜对了!')
        break
# 当退出while循环的时候显示用户一共猜了多少次
print(f'你总共猜了{counter}次')

# 使用while循环计算PV
t = 1
```

```
PV = 0
r = 0.1
while(t <= len(cashflows)):
    PV_t = cashflows[t] / (1 + r) ** (t + 1)
    PV = PV + PV_t
    t = t + 1
```

7. 数据结构：字符串 (string)

7.1 字符串类型

- 由零个或多个字符组成的有限序列，把单个或多个字符用单引号或者双引号包围起来。
- 转义字符：`\n` 换行符，`\t` 制表符。如果字符串本身又包含了 `'`、`"`、`\` 这些特殊的字符，必须要通过 `\` 进行转义处理。
- 原始字符串：以r或R开头的字符串被称为原始字符串，即每个字符都是它本来的含义
- 在 `\` 后面输入个八进制或者十六进制数来表示字符

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.1 转义字符与原始字符串

# 字符串s1中\t是制表符，\n是换行符
s1 = '\time up \now'
print(s1)

# 字符串s2中没有转义字符，每个字符都是原始含义
s2 = r'\time up \now'
print(s2)
```

```
        ime up
ow
\time up \now
```

7.2 字符串运算

- 拼接和重复：`+ *`
- 比较运算：
  - 可以直接使用比较运算符 `<= < > >= == !=` 比较两个字符串的相等性或大小，字符串的大小比较比的是每个字符对应的编码的大小。
  - 可以使用身份运算符 `is` 比较两个变量对应的字符串对象的内存地址。
- 成员运算：可以用成员运算符 `in、not in` 判断一个字符串中是否存在另外一个字符或字符串。
- 获取字符串长度：`len()`
- 索引和切片：
  - 通过 `[n]` 进行索引，其中n是一个整数。假设字符串的长度为N，正向索引为0到N-1的整数，负向索引为-1到-N的整数。
  - 通过 `[开始:结束:步长]` 进行索引，前闭后开。

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.2 字符串运算：拼接和重复

s1 = "hello"
s2 = "world"
print(s1 + " " + s2) # hello world
print(s1 * 3)        # hellohellohello
```

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.2 字符串运算：比较运算
```

```
# 比较运算
s1 = "a"
s2 = "b"
print(ord(s1), ord(s2))      # 97 98
print(s1 < s2)               # True

# 身份运算
s3 = "hello world"
s4 = "hello world"
s5 = s4
print(s5 == s3, s4 is s3)    # True False
```

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.2 字符串运算：成员运算

s = 'hello world'
print('wo' in s)       # True
```

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.2 字符串运算：长度

s = 'hello world'
print(len(s))          # 11
```

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.2 字符串运算：索引和切片
s = 'abc123456'
N = len(s1)

# 获取第一个字符
print(s[0], s[-N])     # a a

# 获取最后一个字符
print(s[N-1], s[-1])   # 6 6

# 从2到4，步长1的正向切片操作
print(s[2:5])          # c12

# 倒数-7到-4，步长1的正向切片操作
print(s[-7:-4])        # c12

# 倒数-7到开头，步长1的正向切片操作，即选择全部
print(s[-7::2])        # c246

# 选择全部，步长为-2的负向切片
print(s[::-2])         # 642ca
```

7.3 字符串方法

采用 `变量名.方法名()` 的方式来调用它的方法。

- 大小写操作：`.capitalize()`，`.title()`，`.upper()`，`.lower()`
- 查找操作：`.find()`，`.index()`
- 替换操作：`.replace("被替换", "替换内容", 替换次数)` 默认全部替换。
- 修剪操作：`.strip()` 将原字符串修剪掉左右两端空格之后的字符串，`.lstrip()`，`.rstrip()`。
- 拆分合并操作：`.split("分隔符", 拆分次数)`（默认为空格）将一个字符串拆分为多个字符串并放在一个列表中，`.join()` 将列表中的多个字符串连接成一个字符串。

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.3 字符串方法：大小写

s1 = 'hello, world!'
```

```
print(s1.capitalize()) # 使用capitalize方法获得首字母大写后的字符串
print(s1.title()) # 使用title方法获得每个单词首字母大写后的字符串
print(s1.upper()) # 使用upper方法获得大写后的字符串

s2 = 'GOODBYE'
print(s2.lower()) # 使用lower方法获得小写后的字符串
```

```
Hello, world!
Hello, World!
HELLO, WORLD!
goodbye
```

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.3 字符串方法：查找

s = 'hello, world!'

# 找到了返回字符串中另一个字符串首字符的索引
print(s.find('or'))        # 8
print(s.index('or'))       # 8
# 找不到返回-1
print(s.find('shit'))      # -1
# 找不到引发异常
print(s.index('shit'))     # ValueError: substring not found
```

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.3 字符串方法：替换

s = 'hello, world'
print(s.replace('o', '@'))     # hell@, w@rld
print(s.replace('o', '@', 1))  # hell@, world
```

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.3 字符串方法：修剪

s = '   helloworld '
print(s.strip())
```

```
helloworld
```

In [ ]:
```
# 7. 数据结构：字符串 (string)
# 7.3 字符串方法：拆分合并

# 使用split拆分
s = 'Hello world'
s_list = s.split()
print(s_list)

# 使用join连接
print('@'.join(s_list))
```

```
['Hello', 'world']
Hello@world
```

8. 数据结构：列表 (list)

8.1 定义和使用列表

列表是由一系元素按特定顺序构成的数据序列。一个列表类型的变量可以保存多个数据，而且允许有重复的数据。

- 使用`[]`字面量语法来定义列表，列表中的多个元素用逗号进行分隔。
- 使用`list()`函数将其他序列变成列表。

In [ ]:
```
# 8. 数据结构：列表 (list)
# 8.1 定义和使用列表

list_1 = [1, 2, 3, 4]
list_2 = list(range(1, 10))
```

8.2 列表运算

和字符串类型一样，列表也支持拼接、重复、成员运算、索引和切片以及比较运算。

In [ ]:
```
# 8. 数据结构：列表 (list)
# 8.2 列表的运算符

list_1 = [1, 2, 3, 4, 5, 6]
list_2 = [7, 8, 9]

# 列表的拼接
list_3 = list_1 + list_2
print(list_3)                   # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# 列表的重复
list_4 = ['hello'] * 3
print(list_4)                   # ['hello', 'hello', 'hello']

# 列表的成员运算
print(100 in list_3)            # False
print('hello' in list_4)        # True

# 获取列表的长度(元素个数)
print(len(list_3))              # 9

# 列表的索引
print(list_1[0])                # 1

# 列表的切片
print(list_1[:5])               # [1, 2, 3, 4, 5]

# 列表的比较运算
list_5 = [1, 2, 3, 4]
list_6 = list(range(1, 5))
# 两个列表比较相等性比的是对应索引位置上的元素是否相等
print(list_6 == list_5)         # True
list_7 = [3, 2, 1]
# 两个列表比较大小比的是对应索引位置上的元素的大小
print(list_5 <= list_5)         # True
```

8.3 列表方法

- 添加和删除元素：`.append()`, `.insert()`, `.remove()`, `.pop(索引，若索引为空的返回值)`, `.clear()`
- 元素位置和次数：`.index('元素', 从此索引往后)`, `.count('元素')`
- 元素排序和反转：`.sort()` 实现列表元素的排序，`.reverse()` 实现元素的反转

In [ ]:
```
# 8. 数据结构：列表 (list)
# 8.3 列表方法：添加和删除元素

list = ['Python', 'Java', 'Go', 'Kotlin']

# 使用append方法在列表尾部添加元素
list.append('Swift')
print(list)                 # ['Python', 'Java', 'Go', 'Kotlin', 'Swift']
# 使用insert方法在列表指定索引位置插入元素
```

```
list.insert(2, 'SQL')
print(list)              # ['Python', 'Java', 'SQL', 'Go', 'Kotlin', 'Swift']

# 删除指定的元素
list.remove('Java')
print(list)              # ['Python', 'SQL', 'Go', 'Kotlin', 'Swift']
# 删除指定索引位置的元素，并返回值
list.pop(0)
print(list)              # ['SQL', 'Go', 'Kotlin', 'Swift']

# 清空列表中的元素
list.clear()
print(list)              # []
```

In [ ]:
```
# 8. 数据结构：列表 (list)
# 8.3 列表方法：元素位置和次数

items = ['Python', 'Java', 'Java', 'Go', 'Kotlin', 'Python']

# 查找元素出现的次数
print(items.count('Python'))        # 2
# 查找元素的索引位置
print(items.index('Python', 2))     # 5
# 注意：虽然列表中有'Java'，但是从索引为3这个位置开始后面是没有'Java'的
print(items.index('Java', 3))       # ValueError: 'Java' is not in list
```

In [ ]:
```
# 8. 数据结构：列表 (list)
# 8.3 列表方法：元素排序和反转

items = ['Python', 'Java', 'Go', 'Kotlin', 'Python']

# 排序
# sort()方法默认按照升序进行排序。
items.sort()
print(items)                # ['Go', 'Java', 'Kotlin', 'Python', 'Python']
items.sort(key = len)
print(items)                # ['Go', 'Java', 'Kotlin', 'Python', 'Python']
items.sort(reverse = True)
print(items)                # ['Python', 'Python', 'Kotlin', 'Java', 'Go']

# 反转
items.reverse()
print(items)                # ['Go', 'Java', 'Kotlin', 'Python', 'Python']
```

9. 数据结构：元祖 (tuple)

9.1 定义和使用元祖

- 在Python中，元组也是多个元素按照一定的顺序构成的序列。元组和列表的不同之处在于，元组是不可变类型，这就意味着元组类型的变量一旦定义，其中的元素不能再添加或删除，而且元素的值也不能进行修改。
- 定义元组通常使用`()`字面量语法。
- 需要提醒大家注意的是，`()`表示空元组，但是如果元组中只有一个元素，需要加上一个逗号，否则()就不是代表元组的字面量语法，而是改变运算优先级的圆括号，所以`('hello', )`和`(100, )`才是一元组，而`('hello')`和`(100)`只是字符串和整数。
- 列表是可变数据类型，元组是不可变数据类型。 ## 9.2 元组的应用场景
- 打包和解包操作：当我们把多个用逗号分隔的值赋给一个变量时，多个值会打包成一个元组类型；当我们把一个元组赋值给多个变量时，元组会解包成多个值然后分别赋给对应的变量。
- 通过星号表达式`*`，可以让一个变量接收多个值。

In [ ]:
```
# 9. 数据结构：元祖 (tuple)
# 元组的应用场景：打包和解包操作

#  打包
a = 1, 10, 100
print(type(a), a)
#  解包
i, j, k = a
print(i, j, k)
#  星号表达式
a = 1, 10, 100, 1000
i, j, *k = a
print(i, j, k)
```

```
<class 'tuple'> (1, 10, 100)
1 10 100
1 10 [100, 1000]
```

10. 数据结构：集合 (set)

10.1 定义和使用集合

- 集合中的各个事物通常称为集合的元素。集合应该满足以下特性：
  1. 无序性：一个集合中，每个元素的地位都是相同的，元素之间是无序的。
  2. 互异性：一个集合中，任何两个元素都是不相同的，集合不能够支持索引运算
  3. 确定性：给定一个集合和一个任意元素，该元素要么属这个集合，要么不属于这个集合，二者必居其一。集合的成员运算在性能上要优于列表的成员运算。
- 定义集合通常使用`{}`字面量语法。{}中需要至少有一个元素，没有元素的{}并不是空集合，而是一个空字典。
- 使用`set()`函数将其他序列变成集合。
- 集合中的元素必须是可哈希(hashable)类型。

In [ ]:
```
# 10. 数据结构：集合 (set)
# 10.1 定义和使用集合

# 创建集合的字面量语法(重复元素不会出现在集合中)
set1 = {1, 2, 3, 3, 3, 2}
print(set1)         # {1, 2, 3}

# 创建集合的构造器语法
set2 = set('hello')
print(set2)         # {'h', 'l', 'o', 'e'}
```

10.2 集合运算

集合可以进行成员运算、交集运算、并集运算、差集运算、比较运算（相等性、子集、超集）等。

In [ ]:
```
# 10. 数据结构：集合 (set)
# 10.2 集合运算：交并差运算

set1 = {1, 2, 3, 4, 5, 6, 7}
set2 = {2, 4, 6, 8, 10}

# 交集
# 方法一：使用 & 运算符
print(set1 & set2)                      # {2, 4, 6}
# 方法二：使用intersection方法
print(set1.intersection(set2))          # {2, 4, 6}
```

```
# 并集
# 方法一：使用 | 运算符
print(set1 | set2)                          # {1, 2, 3, 4, 5, 6, 7, 8, 10}
# 方法二：使用union方法
print(set1.union(set2))                     # {1, 2, 3, 4, 5, 6, 7, 8, 10}

# 差集
# 方法一：使用 - 运算符                       # {1, 3, 5, 7}
print(set1 - set2)
# 方法二：使用difference方法
print(set1.difference(set2))                # {1, 3, 5, 7}

# 对称差
# 方法一：使用 ^ 运算符
print(set1 ^ set2)                          # {1, 3, 5, 7, 8, 10}
# 方法二：使用symmetric_difference方法
print(set1.symmetric_difference(set2))      # {1, 3, 5, 7, 8, 10}
# 方法三：对称差相当于两个集合的并集减去交集
print((set1 | set2) - (set1 & set2))        # {1, 3, 5, 7, 8, 10}
```

In [ ]:
```
# 10. 数据结构：集合 (set)
# 10.2 集合运算：复合赋值运算

set1 = {1, 3, 5, 7}
set2 = {2, 4, 6}
# 将set1和set2求并集再赋值给set1
# 也可以通过set1.update(set2)来实现
set1 |= set2
print(set1)                 # {1, 2, 3, 4, 5, 6, 7}
set3 = {3, 6, 9}
# 将set1和set3求交集再赋值给set1
# 也可以通过set1.intersection_update(set3)来实现
set1 &= set3
print(set1)                 # {3, 6}
```

In [ ]:
```
# 10. 数据结构：集合 (set)
# 10.2 集合运算：比较运算

set1 = {1, 3, 5}
set2 = {1, 2, 3, 4, 5}
set3 = set2
# <运算符表示真子集，<=运算符表示子集
print(set1 < set2, set1 <= set2)    # True True
print(set2 < set3, set2 <= set3)    # False True
# 通过issubset方法也能进行子集判断
print(set1.issubset(set2))          # True

# 反过来可以用issuperset或>运算符进行超集判断
print(set2.issuperset(set1))        # True
print(set2 > set1)                  # True
```

10.3 集合方法

- 添加元素：`.add(), .update()`
- 删除元素：`.discard(), .remove(), .pop(), .clear()`

11. 数据结构：字典 (dictionary)

11.1 定义和使用字典

- 字典以键值对（键和值的组合）的方式把数据组织到一起，以通过键找到与之对应的值并进行操作。字典中的键必须是可哈希(hashable)类型。
- 定义字典通常使用`{键 : 值}`字面量语法。
- 使用`dict(键 = 值)`函数将其他序列变成集合。

In [ ]:
```
# 11. 数据结构：字典 (dictionary)
## 11.1 定义和使用字典

# 创建字典的字面量语法
dict_1 = {
    'a' : '1',
    'b' : '2',
    'c' : '3'
}
print(dict_1)           # {'a': '1', 'b': '2', 'c': '3'}

# 创建字典的构造器语法
dict_2 = dict(d = '4', e = '5', f = '6')
print(dict_2)           # {'d': '4', 'e': '5', 'f': '6'}

#通过zip压缩两个序列并创建字典
dict_3 = dict(zip('ghi', '789'))
print(dict_3)           # {'g': '7', 'h': '8', 'i': '9'}
```

11.2 字典的运算与方法

In [ ]:
```
# 11. 数据结构：字典 (dictionary)
## 11.2 字典的运算与方法

# 成员运算
print('a' in dict_1)           # True

# 索引运算
if 'a' in dict_1:
    dict_1['a'] = 11
dict_1['g'] = 7
print(dict_1)                  # {'a': 11, 'b': '2', 'c': '3', 'g': 7}

# 使用get方法通过键获取对应的值
print(dict_2.get('d'))         # 4

# 使用keys方法获得字典中所有的键
print(dict_2.keys())           # dict_keys(['d', 'e', 'f'])
# 使用values方法获得字典中所有的值
print(dict_2.values())         # dict_values(['4', '5', '6'])
# 使用items方法获得字典中所有的键值对
print(dict_2.items())          # dict_items([('d', '4'), ('e', '5'), ('f', '6')])

# 使用pop方法通过键删除对应的键值对并返回该值
s = dict_1.pop('a')
print(s)                       # 11

# 使用pop方法通过键删除对应的键值对
del dict_1['b']
print(dict_1)                  # {'c': '3', 'g': 7}

# 使用update更新字典元素，相同的键会用新值覆盖掉旧值，不同的键会添加到字典中
dict_2.update(dict_3)
print(dict_2)                  # {'d': '4', 'e': '5', 'f': '6', 'g': '7', 'h': '8', 'i': '9'}
```

12. 函数

12.1 定义和使用函数

- 使用`def 函数名(自变量):`关键字来定义函数。
- Python中函数的自变量称为函数的参数，而因变量称为函数的返回值。
- 在没有特殊处理的情况下，函数的参数(argument)都是位置参数，也就意味着传入参数的时候对号入座即可

In [ ]:
```
# 12 函数
## 12.1 定义和使用函数

def FV_list(CF, r):
    fv = 0
    for i in range(len(CF)):
        fv += float(CF[i]) * (1 + r) ** (len(CF) - i - 1)
    return fv
FV_list(CF, r)
```

12.2 可变参数

通过星号表达式语法来支持可变参数，即在调用函数时，可以向函数传入0个或任意多个参数。

In [ ]:
```
# 12 函数
## 12.2 可变参数

# 用星号表达式来表示args可以接收0个或任意多个参数
def add(*args):
    total = 0
    # 可变参数可以放在for循环中取出每个参数的值
    for val in args:
        if type(val) in (int, float):
            total += val
    return total

print(add(1))    # 1
print(add(1, 2)) # 3
```

12.3 直接赋值变量的函数

`变量名 = 函数名 参数 : 运算过程`

In [ ]:
```
# 12 函数
## 12.3 直接赋值变量的函数

annual_rtn = lambda r_daily : r_daily * 365
annual_rtn(0.05)  # 18.25
```

13. 读写txt文件

- 通过`open(文件名, 读写类型, encoding = 'utf-8')`打开文件。
  - 读写类型包括：`w`写；该文件已经存在，它将被覆盖，并且如果该文件不存在，它将被创建。`wb`以二进制模式写入；二进制模式允许您以非文本模式(例如图像或声音文件)将数据写入文件中。`r`读。
  - encoding='utf-8'意味着可以在文件中写入任意Unicode字符。
- `.write()`将数据写入该文件。
- `.read()`读取字符；从指针开始读取相应位数字符；当括号内数值为空时，读取到文档末尾。
- `.readline()`读取并返回文件中的一行文本数据，并将文件指针移动到下一行的开头。

- `.tell()` 获取当前文件指针的位置。
- `.seek(偏移量, 基准点)` 将文件指针移动到指定位置。基准点为0（开头）、1（目前指针位置）、2（结尾）。
- `.close()` 关闭文件。

In [ ]:
```
# 13. 读写文件

# 通过open打开
file_1 = open('data.txt', 'wb')
file_1.write('abc'.encode()) #采用wb模式，需要使用encode()方法将文本字符串转换为字节字符串
file_1.close()

# 通过with as语句打开
with open('data.txt', 'w') as file_2:
    file_2.write('abc')
file_2.close()
```

14. 读写csv/excel表格

- 通过 `open(文件名, 读写类型, encoding = 'utf-8')` 打开文件。
- 使用pandas库内 `pd.read_excel()`或者`pd.read_csv()` 打开表格。该命令可选参数包括：
  - `file_name_in` 要读取到的Excel文件名
  - `header` 指定将文件中的哪一行用作列名（默认为0，即使用第一行作为列名）
  - `index_col` 指定要用作行索引的列的名称或位置（默认为None）
  - `usecols` 指定要读取的列的名称或位置（默认为所有列）
  - `skiprows` 指定要跳过的行数（默认为0，即不跳过任何行）
  - `parse_dates` 指定哪些列应该被解析为日期（默认为False）
  - `skip_blank_lines` 指定是否跳过空行（默认为True）
  - `encoding` 指定文件的编码类型（默认为UTF-8）
  - `sheet_name` 指定要读取的工作表的名称或位置（默认为0，即第一个工作表）（仅仅excel）
  - `engine` 指定要使用的Excel解析引擎，可以是"openpyxl"（默认）或"xlrd"
- 使用 `to_excel()` 导出表格。该命令可选参数包括：
  - `file_name_out` 要导出到的Excel文件名
  - `sheet_name` 要将数据框写入的工作表名称
  - `index` 是否将数据框的索引写入Excel文件中
  - `header` 是否将数据框的列名写入Excel文件中
  - `startrow` 从哪一行开始写入数据框，默认是第0行
  - `startcol` 从哪一列开始写入数据框，默认是第0列
- 使用with as语句和 `pd.ExcelWriter()` 方法创建向Excel文件中写入多个工作表的对象。ExcelWriter()可以设定多个可选参数。

In [ ]:
```
# 14. 读写csv/excel表格

import pandas as pd

# 读取表格文件
df1 = pd.read_csv('data.csv', index_col = 0, parse_dates = True)
df2 = df1 - df1.shift(1) # 通过shift方法对df1数据框进行偏移，以计算相邻行之间的差异。

# 导出表格文件
file_name_out = 'output.xlsx'
df1.to_excel(file_name_out, 'Sheet1')
df2.to_excel(file_name_out, 'Sheet2')

# 通过with as导出表格文件
with pd.ExcelWriter(file_name_out, engine="openpyxl", mode='a') as writer:
```

```
    df1.to_excel(writer, sheet_name='Sheet1')
    df2.to_excel(writer, sheet_name='Sheet2')
```

In [ ]:
```
# 14. 读写csv/excel表格
# 通过.shift(移动步长)计算差值

f = pd.read_csv('sp500index.csv', index_col = 0, parse_dates = True)
f_shifted = f.shift(1)
f_diff_price = f - f_shifted
f_diff_rate = (f_diff_price / f) * 100
f_diff_price = f_diff_price.rename(columns={"Day Close Price": "Daily Difference"})
f_diff_rate = f_diff_rate.rename(columns={"Day Close Price": "Daily Return Rate"})
print(f_diff_price)
print(f_diff_rate)
```

15. 其他模块

15.1 使用decimal模块进行高精度数学计算

In [ ]:
```
# 15.1 使用Decimal模块进行高精度数学计算

import decimal
from decimal import Decimal

# 使用.getcontext().prec设置精度
decimal.getcontext().prec = 4  # 保留4位小数
e = Decimal(1) / Decimal(11)
print(e)    # 0.09091

decimal.getcontext().prec = 50  # 保留50位小数
e = Decimal(1) / Decimal(11)
print(e)    # 0.090909090909090909090909090909090909090909090909
```

15.2 使用math模块

- `math.ceil(x)` 返回不小于x的最小整数
- `math.floor(x)` 返回不大于x的最大整数
- `math.sqrt(x)` 返回x的平方根
- `math.pow(x, y)` 返回x的y次方
- `math.exp(x)` 返回e的x次方
- `math.log(x[, base])` 返回x的对数，默认以e为底数，可以指定底数base
- `math.sin(x)`、`math.cos(x)`、`math.tan(x)` 返回x的正弦、余弦、正切值
- `math.pi` 圆周率
- `math.e` 自然对数

In [ ]:
```
## 15.2 使用math模块

import math

print(round(math.exp(1), 2)) # 2.72
pi = math.pi
print(round(pi, 2))          # 3.14
```

15.3 使用datetime模块获取日期与时间

In [ ]:
```
# 15.3 使用datetime模块获取日期与时间
```

```
import datetime
print(datetime.datetime.now())         # 2023-02-27 08:57:21.961563
print(datetime.datetime.now().date()) # 只返回日期 2023-02-27
type(datetime.datetime.now().date())  # datetime.date
```

16. Numpy

- NumPy是一个Python科学计算库
- Python列表可以包含不同类型的元素，而NumPy数组要求所有元素都是相同的类型。 ## 16.1 创建NumPy数组
- 创建数组：
  - `np.array()` 创建数组，可转换成数组的对象包括列表，元祖等。
  - `np.arange(开始, 结束, 步长)` 前闭后开，创建等差数列数组。
  - `np.linspace(开始, 结束, 样本数量)` 默认前后均为闭区间，创建等差数列数组。
- 创建特殊数组
  - `np.zeros((行, 列), 数据类型)` 和 `np.ones((行, 列), 数据类型)` 创建全为0或1的数组。
  - `np.ones_like()` 和 `np.zeros_like()` 创建一个与给定数组形状和数据类型相同的全1或全0数组。
  - `np.empty((行, 列))` 创建空数组。
  - `np.ones_like()` ， `np.zeros_like()` ， `np.empty_like()` 创建与给定数组形状和数据类型相同的全1/0/空数组。
  - `np.eye(边长)` 创建一个单位矩阵，即主对角线上的元素为1，其他元素为0的方阵。
  - `np.random.randn()` 创建符合正态分布的随机数组。
  - `np.random.binomial(n, p, size)` 创建符合二项分布的随机数组。二项分布的参数：试验次数 n 和成功概率 p
  - `np.random.default_rng().integers()` 创建随机整数数组。

In [ ]:
```
# 16.1 创建NumPy数组
import numpy as np

# 创建数组
a_list = [1, 2, 3]
a_array = np.array(a_list)
b_array = np.array([[1, 2, 3], [1, 2, 3]])  # 创建一个二维数组，注意两个中括号

# 创建等差数列数组
c_array = np.arange(0, 10, 2)     # [0 2 4 6 8]
d_array = np.linspace(0, 8, 5)    # [0. 2. 4. 6. 8.]

# numpy中的数组要求元素的数据类型必须全部一致
# 包含了两种数据类型，numpy将所有元素类型转换为通用的object类型对象，该数组无法进行相关计算
f_array = np.array([1, 2, 3, 'a'])

# 创建全为0或1的数组
zero_array = np.zeros((2, 3), dtype = int)
ones_array = np.ones((2, 3), dtype = float)

# 创建给定形状和数据类型相同的全1数组
a_ones_array = np.ones_like(a_array)      # [1 1 1]

# 3x3的正态分布随机数组
normal_arr = np.random.randn(3, 3)  # 创建一个 3x3 的正态分布随机数组
print(normal_arr)

# 3x3的二项分布随机数组
n, p = 10, 0.5  # 二项分布的参数：试验次数 n 和成功概率 p
binomial_arr = np.random.binomial(n, p, size = (3, 3))
print(binomial_arr)
```

```
# 创建随机数生成器
rng = np.random.default_rng()
low, high, size = 0, 10, (3, 3)                # 指定范围[low, high)和数组大小
integer_arr = rng.integers(low, high, size)  # 创建一个 3x3 的随机整数数组
print(integer_arr)
```

```
[[-0.11833739 -2.66915202  2.2741429 ]
 [-1.63913326  1.22836076 -0.13486999]
 [ 0.06710902  0.51800726  0.67243329]]
[[6 5 5]
 [6 5 3]
 [3 5 4]]
[[7 3 4]
 [1 7 2]
 [6 0 3]]
```

16.2 访问数组中的元素

- 通过 `array[开始, 结束, 步长]` 切片选择数组的子集。
- 通过 `array[行, 列]` 访问数组中的元素。
- `[0, :]`：访问第0行的所有元素。
- `[:, 0]`：访问第0列的所有元素。

16.3 数组的属性

- `.shape` 或 `np.shape()` 返回一个元祖，包含数组元素在每个维度上的数量。如果数组没有任何元素，`.shape` 会返回 (0,) 的元组。
- `.dtype` 返回数组内元素的类型，，如int、float等。
- `.ndim` 返回数组的维度，即有几个轴。
  - `.size` 或 `np.prod(数组.shape)` 返回数组内包含的总元素数量。

In [ ]:
```
# 16.3 数组的属性

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

print(a.shape)        # (3, 3)

print(type(a))        # <class 'numpy.ndarray'>

print(a.dtype)        # dtype('int32')数组中的元素类型为整型（32位表示）

print(a.ndim)         # 2

print(a.size)         # 9
```

```
(3, 3)
<class 'numpy.ndarray'>
int32
2
9
```

16.4 数组方法：拼接与删除

- `np.append()` 拼接数组。将一个数组的元素添加到另一个数组的末尾。
- `np.concatenate()` 拼接数组。可以将两个或多个数组在指定轴上连接，堆叠的轴上应当具有相同的大小。
- `np.hstack()` 水平拼接数组。
- `np.vstack()` 垂直拼接数组。
- `np.delete()` 删除数组的元素或子数组。可以根据指定的索引和轴删除数组中的元素。

In [ ]:
```
# 16.4 数组方法：拼接

# 定义两个数组
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[7, 8, 9], [10, 11, 12]])

# np.append()，在数组a的末尾添加数组b的所有元素
c = np.append(a, b)
print(c)                        #  [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12]
```

In [ ]:
```
# 16.4 数组方法：拼接
# np.concatenate()，在轴0（行）上连接数组a和数组b
# 数组堆叠方向的边长大小应该一致。
d = np.concatenate((a, b), axis = 0)
print(d)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

In [ ]:
```
# 16.4 数组方法：拼接

# np.vstack()，在垂直方向上堆叠数组a和数组b
e = np.vstack((a, b))
print(e)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

In [ ]:
```
# 16.4 数组方法：删除

# np.delete，删除数组f的第1行（轴0）
f = np.delete(e, 1, axis = 0)
print(f)
```

```
[[ 1  2  3]
 [ 7  8  9]
 [10 11 12]]
```

16.5 数组方法：转置，排序与重塑

- `.T` 或 `.transpose()` 转置数组。对于二维数组，结果相同.对于多维数组，`.transpose()` 可以指定要交换的轴。
- `.reshape(行, 列)` 改变数组的形状。
- `np.expand_dims(数组)` 为数组增加行或列维度。
- `.flatten()` 将多维数组转换为一维数组。
- `np.flip()` 反转数组的元素顺序。

- `.sort()` 对数组进行排序，可以指定排序的轴和排序算法，此操作会修改原数组。

In [ ]:
```
# 16.5 数组方法：转置

# .transpose转置数组
# 定义一个三维数组
a = np.array([[[1, 2, 3], [4, 5, 6]],
              [[7, 8, 9], [10, 11, 12]],
              [[13, 14, 15], [16, 17, 18]]])

b = a.transpose((0, 2, 1))  # 转置多维数组，交换第二个和第三个轴
print(b)
```

```
[[[ 1  4]
  [ 2  5]
  [ 3  6]]

 [[ 7 10]
  [ 8 11]
  [ 9 12]]

 [[13 16]
  [14 17]
  [15 18]]]
```

In [ ]:
```
# 16.5 数组方法：转置，排序与重塑

# np.arange改变数组形状
arr = np.arange(6)
print(arr)

# 让数组变成2行3列
reshaped_arr = arr.reshape(2, 3)
print(reshaped_arr)
```

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

In [ ]:
```
# 16.5 数组方法：重塑

# np.expand_dims()增加行或列维度
arr = np.array([1, 2, 3])
print("Original array:", arr)

# 将一维数组转换为一个列向量（在行上添加一个新维度）
row_vector = np.expand_dims(arr, axis = 0)
print("Row vector:\n", row_vector)

# 将一维数组转换为一个行向量（在列上添加一个新维度）
column_vector = np.expand_dims(arr, axis = 1)
print("Column vector:\n", column_vector)
```

```
Original array: [1 2 3]
Row vector:
 [[1 2 3]]
Column vector:
 [[1]
 [2]
 [3]]
```

In [ ]:
```
# 16.5 数组方法：重塑

# np.expand_dims()增加行或列维度
arr = np.array([[1, 2], [3, 4]])
print("Original 2D array:\n", arr)

# 在二维数组的行上添加一个新维度
# 这将原始的 2D 数组 (2, 2) 转换为一个 3D 数组 (1, 2, 2)
```

```
expanded_arr = np.expand_dims(arr, axis = 0)
print("Expanded array:\n", expanded_arr)
```

```
Original 2D array:
 [[1 2]
 [3 4]]
Expanded array:
 [[[1 2]
  [3 4]]]
```

In [ ]:
```
# 16.5 数组方法：重塑

# .flatten()将多维数组转换为一维数组
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Original array:\n", arr)

flattened_arr = arr.flatten()
print("Flattened array:", flattened_arr)
```

```
Original array:
 [[1 2 3]
 [4 5 6]]
Flattened array: [1 2 3 4 5 6]
```

In [ ]:
```
# 16.5 数组方法：重塑

# np.flip()反转数组的元素顺序
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Original array:\n", arr)

# 反转数组内的行顺序（axis = 0）
flipped_arr = np.flip(arr, axis = 0)
print("Flipped along axis 0:\n", flipped_arr)
```

```
Original array:
 [[1 2 3]
 [4 5 6]]
Flipped along axis 0:
 [[4 5 6]
 [1 2 3]]
```

16.6 数组方法：条件查询

- 直接使用布尔表达式筛选数组元素。
- `np.nonzero(条件表达式)` 返回非零元素的索引，将布尔表达式作为参数传递给 `np.nonzero()`，以获得满足条件的元素的索引。
- `np.where(条件表达式, x, y)` 替换满足条件的元素。x和y可以是数组或者元素。当条件为True时，返回来自x数组的元素，或者替换成x元素。False则执行y。

In [ ]:
```
# 16.6 数组方法：条件查询

# 可以使用逻辑运算符连接多个条件
a = np.array([1, 3, 5, 7, 9])
print(a[(a > 2) & (a <= 7)])     # [3 5 7]


# 返回非零元素的索引
indices = np.nonzero(a > 5)
print(indices)                   # (array([3, 4])

# 数组a中所有大于5的元素替换为1，其他元素替换为0
result = np.where(a > 5, 1, 0)
print(result)                    # [0 0 0 1 1]

# 当数组a中的元素大于5时，result数组中相应元素将取自数组a，否则取自数组b
b = np.array([9, 7, 5, 3, 1])
```

```
result = np.where(a > 5, a, b)
print(result)                    # [9 7 5 7 9]
```

```
[3 5 7]
(array([3, 4], dtype=int64),)
[0 0 0 1 1]
[9 7 5 7 9]
```

16.7 数组数学运算的广播机制

广播（Broadcasting）机制：

- 当对两个数组进行操作时，Numpy会逐元素比较它们的形状。从尾（即最右边）维度开始，然后向左逐渐比较。
- 只有当 两个维度相等 ，或者 其中一个维度是1时 ，这两个维度才会被认为是兼容，可以进行计算。
- 在广播过程中，维度较小的数组将沿着大小为1的维度复制扩展，使其形状与较小的数组相同。

In [ ]:
```
# 16.7 数组数学运算的广播机制

a = np.array([[1, 2, 3], [4, 5, 6]])
c = np.array([1, 2])
# 此时直接运算，将返回ValueError

# 使数组c的形状与a兼容，添加一个新维度
c_compatible = c[:, np.newaxis]
print(c_compatible)

# 现在，c_compatible的形状为(2, 1)，可以与a进行广播
# c_compatible将沿着列复制
result = a + c_compatible
print(result)

# [[1 2 3]    [[1 1 1]    [[2 3 4]
#  [4 5 6]]  +  [2 2 2]] =  [6 7 8]]
```

```
[[1]
 [2]]
[[2 3 4]
 [6 7 8]]
```

16.8 数组数学运算的函数

- 数组求和 `.sum()`
- 数组标准差 `.std()`
- 数组均值 `.mean()`
- 数组累积和 `.cumsum()`
- 数组元素积 `.prod()`
- 数组累积积 `.cumprod()`
- 指数函数 `np.exp(数组)`
- 平方根 `np.sqrt(数组)`
- 以上函数的参数包括：
  - `axis` 指定沿哪个轴进行求和。默认值为None，表示对所有元素求和。0为行，1为列。
  - `dtype` 指定结果的数据类型。默认值为None，表示自动确定数据类型。
  - `out` 指定一个用于存储结果的数组。默认值为None，表示创建一个新的数组来存储结果。
  - `keepdims` 默认值为False。如果为True，则保持结果数组的维度与输入数组相同。
- `np.unique()` 用于查找数组中的唯一元素。参数 `return_index` 返回唯一元素的索引，参数 `return_counts` ，返回唯一元素的计数。

In [ ]:
```python
# 16.8 数组数学运算的函数

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

# 所有元素求和
total_sum = a.sum()                 # 21

# 沿行求和，即列相加
sum_along_rows = a.sum(axis = 0)    # array([5, 7, 9])

# 沿列求和，即行相加
sum_along_columns = a.sum(axis=1)  # array([6, 15])

# 元素累积和
cumulative_sum = a.cumsum()         # array([ 1,  3,  6, 10, 15, 21])

# 沿行累积和
cumulative_sum_along_rows = a.cumsum(axis = 0)        # array([[1, 2, 3],
                                                      #        [5, 7, 9]])
# 沿列累积和
cumulative_sum_along_columns = a.cumsum(axis = 1)     # array([[1, 3, 6],
                                                      #        [4, 9, 15]])
```

In [ ]:
```python
# 16.8 数组数学运算的函数

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

# 查找唯一值
unique_values = np.unique(a)
print(unique_values)                # [1 2 3 4 5 6 7]

# 返回唯一值及其在原数组中的索引
unique_indices = np.unique(a, return_index = True)
print(unique_indices)               # (array([1, 2, 3, 4, 5, 6, 7]), array([ 2,  1,  0,  5,  6,

# 返回唯一值及其在原数组中的计数
unique_values, unique_counts = np.unique(a, return_counts = True)
print(unique_counts)                # [1 2 2 2 2 2 1]
```

17. Pandas序列（Series）

Pandas提供了多种数据结构，包括序列（Series）和数据框（DataFrame）。

- Series是带有标签的一维数组，可以存储任何数据类型。每个元素都有一个唯一的标签（索引），可以通过该标签来访问和操作数据。
- DataFrame是带有标签的二维表格，可以看作是由多个Series组成的字典。每个列可以具有不同的数据类型，并且可以使用标签或位置索引来访问和操作数据。

17.1 序列（Series）的基本属性

- 在pandas中，可以从多种数据源中导入序列，例如列表、字典、数组等。
  - 创建序列 `pd.Series(数值, 索引, 名称)`
  - 从数据框中某一列创建序列 `col_series = df['col']`
  - 从数据框中某一行创建序列 `row_series = df.loc['row']`
- 序列的基本属性：
  - `.head()` 显示序列的前几个元素。
  - `.tail()` 显示序列的后几个元素。
  - `.index` 返回序列的索引。

    - `.dtypes` 获取数据框中每列的数据类型。
    - `.size` 返回序列中元素的总数。
    - `.shape` 返回序列的形状，即一个元组：第一位为序列的元素总数（size/len()），第二位为1。
    - `.unique()` 返回序列中唯一值。
- `.values` 将数据框中的数据转化为Numpy数组

In [ ]:
```
# 17.1 序列（Series）的基本属性

s = pd.Series(['A', 'B', 'C', 'A', 'B', 'A', 'D', 'C', 'A'])

# 返回一个新的序列，其中每个唯一的值作为索引，它们出现的频率作为值。
freq = s.value_counts()
print(freq)
```

```
A    4
B    2
C    2
D    1
dtype: int64
```

17.2 序列的描述性统计

- `.min()` 返回序列中最小值。
- `.max()` 返回序列中最大值。
- `.argmin()` 返回最小值所在的位置。
- `.argmax()` 返回最大值所在的位置。
- `.mean()` 返回均值。
- `.median()` 返回中位数。
- `.std()`：返回标准差。
- `.sum()`：返回序列中所有元素的和。
- `.cumsum()`：返回一个新的序列，其中每个元素表示原序列中前面所有元素的累加和。
- `.prod()`：返回序列中所有元素的乘积。
- `.cumprod()`：返回一个新的序列，其中每个元素表示原序列中前面所有元素的累积积。
- `.describe()`：返回序列的描述性统计信息，包括计数、均值、标准差、最小值、最大值、25% 分位数、50% 分位数和 75% 分位数。
- `.quantile(q)`：返回 Series 的分位数，其中 q 是一个浮点数或浮点数列表，表示要计算的分位数的位置。

In [ ]:
```
# 17.2 序列的描述性统计

s = pd.Series([1, 2, 3, 4, 5])

# 计算累加和、累积积
print(s.cumsum())
print(s.cumprod())

# 计算描述性统计信息和分位数
print(s.describe())

# 计算中位数和 25%、75% 分位数
print(s.quantile(0.5))
print(s.quantile([0.25, 0.75]))
```

```
0     1
1     3
2     6
3    10
4    15
dtype: int64
0      1
1      2
2      6
3     24
4    120
dtype: int64
count    5.000000
mean     3.000000
std      1.581139
min      1.000000
25%      2.000000
50%      3.000000
75%      4.000000
max      5.000000
dtype: float64
3.0
0.25    2.0
0.75    4.0
dtype: float64
```

17.3 序列中缺失值和异常值处理

- `s.isnull()` 返回一个布尔型序列，其中每个元素表示s中是否为缺失值。
- `s.notnull()` 返回一个布尔型序列，其中每个元素表示s中是否不是缺失值。
- `s.hasnans` 返回一个布尔值，表示s中是否存在缺失值。
- `s.fillna(x)` 返回一个新的序列，其中将s中的缺失值用x填充。
- `s.replace(x, y)` 返回一个新的序列，其中将s中的x值用y替换。
- `s.where(条件, x)` 返回一个新的序列，其中将满足条件的值保留，不满足条件的值用x替换。
- `s.mask(条件, x)` 返回一个新的序列，其中将满足条件的值用x替换，不满足条件的值保留。
- `s.dropna()` 返回一个新的序列，其中删除包含缺失值的行。

In [ ]:
```python
# 17.3 序列中缺失值处理

# 创建一个包含缺失值序列
s = pd.Series([1, 2, np.nan, 4, 5, np.nan])

# 判断s中是否存在缺失值
print(s.hasnans)                    # 输出 True

# 使用0填充s中的缺失值
s_filled = s.fillna(0)
print(s_filled)                     # 输出 [1. 2. 0. 4. 5. 0.]

# 将s中的缺失值替换为-1
s_replaced = s.replace(np.nan, -1)
print(s_replaced)                   # 输出 [1.  2. -1.  4.  5. -1.]

# 保留s中大于3的值，其余值用0替换
s_where = s.where(s > 3, 0)
print(s_where)                      # 输出 [0. 0. 0. 4. 5. 0.]
```

17.4 序列的数学运算

- Pandas序列可以采用特定的语法与Python提供的基础运算符进行逐元素数学运算。
- 如果需要对两个序列对象进行运算时，必须使用特定语法。

- `s.add(x)` 返回一个新的序列，其中每个元素都加上x。 `s + x`
- `s.subtract(x)` 返回一个新的序列，其中每个元素都减去x。 `s - x`
- `s.mul(x)` 乘法， `s * x`
- `s.div(x)` 除法， `s / x`
- `s.gt(x)` 返回一个布尔型序列，其中每个元素表示s中的元素是否大于x， `s > x`
- `s.eq(x)` 等于x， `s == x`
- `s.lt(x)` 小于x， `s < x`
- `s.ne(x)` 不等于x， `s != x`

17.5 序列的链式操作（Chaining Methods）

- Pandas序列支持链式操作，即在一个语句中连续调用多个方法。

In [ ]:
```
# 17.5 序列的链式操作

s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# 链式操作：将s中的偶数元素乘以2，然后再加上1
result = s[s % 2 == 0].mul(2).add(1)
print(result)           # 输出 [5, 9, 13, 17, 21]
```

18. Pandas数据框

18.1 Pandas数据框（DataFrame）的基本属性与方法

- 在pandas中，可以从多种数据源中导入数据框，例如字典、数组、CSV文件、SQL数据库等。
  - 从字典等导入 `pd.DataFrame(data)`
  - 从csv导入 `pd.read_csv('data.csv')`
- 数据框的基本属性：
  - `df.head()` 显示数据框的前几行数据。
  - `df.tail()` 显示数据框的后几行数据。
  - `df.index` 返回行索引，返回数据结构为Index，即Pandas中专门用于存储轴标签的数据结构。
  - `df.columns` 返回列名，返回数据结构为Index。
  - `df.dtypes` 获取数据框中每列的数据类型。
  - `df.info()` 获取数据框架的总体信息，包括每列的名称、非空值数量、每列的数据类型和占用内存的总数。
  - `df.size` 返回数据框中元素的总数，即行与列的乘积。
  - `df.shape` 返回数据框的形状，即一个元组：第一位为行，第二位为列
  - `df.ndim` 返回数据框的维度，即轴数。
  - `len(df)` 返回数据框的行数。
- 将数据框转换成numpy数组：
  - `.values` 或 `df.to_numpy()` 将数据框转内的值转换成numpy数组，不包括索引与列名。
- 描述性统计：
  - `df.describe()` 获取描述性统计，通过 `include = ['col1', 'col2']` 或 `exclude = []` 参数选择具体的列。
- 转置
  - `df.T` 转置数据框，即将行与列交换，此操作不会修改原数据框。
- 排序
  - `df.sort_index()` 按照索引值对数据框进行排序，参数包括：
    - `axis` 指定排序的方向，默认为0，即按照行索引进行排序，1则表示按照列索引进行排序。

    - `level` 当数据框有多层索引时，指定要排序的层级。
    - `ascending` 指定排序顺序，如果为True则按照升序排序，为False则按照降序排序。
- `df.sort_values()` 按照指定列的值对数据框进行排序，参数包括：
    - `by` 指定要排序的列名，可以指定单个列名或多个列名，多个列名使用列表传递。
    - `axis` 指定排序的方向，默认为0，即按照行索引进行排序，1则表示按照列索引进行排序。
    - `ascending` 指定排序顺序，如果为True则按照升序排序，为False则按照降序排序。
    - `na_position` 指定缺失值的位置，可以为'last'或'first'，分别表示将NA值放在最后或最前。

In [ ]:
```python
# 18.1 数据框（DataFrame）的基本属性

import pandas as pd

df = pd.read_csv("data")     # 读取CSV文件

index = df.index             # 获取行索引，返回Index
columns = df.columns         # 获取列名，返回Index

index.values                 # 获取行索引的值，返回Numpy数组
columns.values               # 获取列名的值，返回Numpy数组

df.dtypes.value_counts()     # 统计每种数据类型的列数
```

18.2 从数据框中选择行和列

- 通过 `df['col']` 选择数据框中的列。 如果要选择多列，参数需要作为列表传入 `df[['col1', 'col2']]` 。
- 通过 `df.loc[]` 根据行和列的标签（名称）进行选择。如果要选择多行多列，参数需要作为列表传入，或者使用 `[ : ]` 。
- 通过 `df.iloc[]` 根据行和列的整数位置（索引）进行选择。如果要选择多行多列，参数需要作为列表传入，或者使用 `[ : ]` 。
- 通过 `.select_dtypes()` 选择特定数据类型类型的列。参数为 `include/exclude = ['数据类型']`

In [ ]:
```python
# 18.2 从数据框中选择行和列
# 通过`df['col']`选择数据框中的列

df = pd.DataFrame(data, index=[10001, 10002, 10003, 10004, 10005, 10006])

# 选择多列数据，参数为列表。
cols = ['CUSIP', 'HSNAICS']
stock_ind = df[cols]
print(stock_ind)
```

In [ ]:
```python
# 18.2 从数据框中选择行和列
# 通过.loc或.iloc进行选择

# 选择单个元素
print(df.loc[10001, 'CUSIP'])
print(df.iloc[0, 0])

# 选择单行数据
print(df.loc[10001])
print(df.iloc[0])

# 选择多行数据，使用[:]，前后均为闭区间
print(df.loc[10001:10003])
print(df.iloc[0:3])

# 选择多列数据，":"表示输出所有行
```

```
print(df.loc[:, 'CUSIP':'HSNAICS'])
print(df.iloc[:, 0:2])

# 选择多行多列数据
print(df.loc[[10001, 10002]])
print(df.iloc[0:2, 0:2])

# 选择倒数第4到倒数第2行数据
print(df.iloc[-4:-1])

# 选择0到9行的间隔为2的数据
print(df.iloc[0:10:2])
```

In [ ]:
```
# 18.2 从数据框中选择行和列

# 选择数据类型为浮点数的列
print(df.select_dtypes(include = ['float']))
```

18.3 复制数据框并更改元素

- 使用 `df.iloc[] = x` 或 `df.loc[] = x` 来修改单个元素或一组元素的值。这一操作会直接修改原始数据框。
- 使用 `.copy()` 方法来创建一个原始数据框的副本。

In [ ]:
```
# 18.3 复制数据框并更改元素

# 复制数据框
df_copy = df.copy()

# 修改元素值
df_copy.iloc[0, 0] = 0
```

18.4 数据框的描述性统计

- 数据框统计的语法和序列相似，包括 `.count()`、`.max()`、`.min()`、`.sum()` 等。
  - 计算默认针对所有列，将返回所有列的统计结果。
  - 参数 `axis = 0` 代表所有列， `axis = 1` 代表所有行。
- `df.describe()` 获取描述性统计，
  - 通过 `include = ['col1', 'col2']` 或 `exclude = []` 参数选择具体的列。
  - 通过 `include = [np.number, np.object, pd.Categorical]` 参数选择具体的数据类型。
  - 对于非数值列（如对象和分类数据类型）， `.describe()` 将返回计数、唯一值数量、众数、众数出现的频率。
- `df.nlargest(n, 列)` 与 `df.nsmallest(n, 列)` 基于某个列的值选择最大的n行。

In [ ]:
```
# 18.4 数据框的描述性统计

# 计算每列的最小值
df.min()

# 计算每行的最小值
df.min(axis = 1)

# 通过链式操作返回制定列的最大值和最小值
df.loc[:, ['col1', 'col2']].max().min()

# 获取数据框中非数值列的描述性统计信息，并对结果进行转置（.T）以获得更好的可读性
df.describe(include = [np.object, pd.Categorical]).T

# 根据col1列的值选择最大的前5行，在返回的行中只保留col2列数据
df.nlargest(5, 'col1')['col2']
```

18.5 数据框的数学运算

- 和序列一样，数据框也可以进行逐元素的算术运算。如果两个对象的形状不同，Pandas会尝试将它们广播（broadcasting）到相同的形状。
- 特定语法与Python提供的基础运算符都可以在数据框中进行数学运算。
- 但是在使用 `.add()` 等方法时，可以通过 `fill_value` 参数指定如何处理缺失值，通过 `axis` 参数指定沿着行或列广播。
- `pct_change()` 计算数据框一列或行的百分比变化。

In [ ]:
```
# 18.5 结合数学运算与统计的筛选

# 选择前20%的mkvalt
cutoff = df['mkvalt'].quantile(0.8)
select = df['mkvalt'].ge(cutoff)

# 选择20% - 80%的mkvalt
con1 = df['mkvalt'].ge(df['mkvalt'].quantile(0.2))
con2 = df['mkvalt'].le(df['mkvalt'].quantile(0.8))

con_final = con1 & con2
df.loc[con_final]
```

In [ ]:
```
# 18.5 数据框的数学运算
# 计算数据框中每一列的百分比变化

df = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])

df_pct = df.pct_change()
print(df_pct)
```

```
          A         B         C
0       NaN       NaN       NaN
1 -0.375583 -1.467983 -2.269107
2 -0.651249 -1.943618 -2.276322
3  4.621311 -1.795195 -2.168751
4 -2.493172  0.772822 -3.013284
```

18.6 数据框的运用函数

- `df.apply()` 方法可以将一个函数应用于数据框的每一行或每一列，并返回一个新的数据框。

In [ ]:
```
# 18.6 运用函数

# 创建一个包含随机数的数据框
df = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])

# 每一列求累计和
cumsum_df = df.apply(np.cumsum)

# 每一行求最大值和最小值之差
max_min_diff = df.apply(lambda x: x.max() - x.min(), axis=1)
```

18.7 数据框的分组聚合运算

- 在groupby方法中，可以使用一个或多个列名作为参数来指定分组列，然后使用agg方法来指定需要进行的聚合计算，语法如下：
- `df.groupby(key).agg(func)` 其中，key表示用于分组的列名或列名列表，func表示需要对每个分组进行的聚合操作。

In [ ]:
```
# 18.7 数据框的分组聚合运算

# 对每个分组计算多个聚合函数
df.groupby('column').agg(['mean', 'count', 'max'])

# 对每个分组指定不同的聚合函数
df.groupby('column').agg({'column1':'mean', 'column2':'sum'})

# 使用自定义函数对每个分组进行聚合
def custom_agg(x):
    return x.sum() / x.count()

df.groupby('column').agg(custom_agg)
```

18.8 从数据框创建透视表

透视表（Pivot Table）是一种对数据进行多维汇总的交叉表格，其语法结构为：
`pd.pivot_table(data, values, index, columns)` 其中，常见参数包括：

- `data` 需要进行透视操作的数据框
- `values` 需要聚合的列，即透视表中每一个元素的数值
- `index` 指定透视表中的行索引
- `columns` 指定透视表中的列索引
- `aggfunc` 用于聚合的函数

In [ ]:
```
## 18.8 从数据框创建透视表

# 按照年份（'fyear'）和股票代码（'tic'）对数据进行透视，聚合的列是销售额（'sale'）
pivot_sale = pd.pivot_table(df, index = 'fyear', columns = 'tic', values = 'sale')
```

18.9 异常值与缺失值的处理

- 检查数据框中的缺失值
  - `df.isnull()` 返回一个和df相同大小的布尔值数据框，每个元素表示原始数据中对应位置是否为缺失值（即NaN）。
  - `df.notnull()` 同上，对应位置是否有有效值（即非NaN值）。
  - `df.hasnans` 返回一个布尔值，表示df中是否存在缺失值。
- 替换缺失值
  - `df.fillna(x)` 用x填充缺失值，默认返回一个新的数据框。
  - `df.replace(np.nan, x)` 用x替换缺失值，默认返回一个新的数据框。
- 删除缺失值
  - `df.dropna()` 删除数据框中包含缺失值的行或列，返回一个新的数据框。默认删除行，通过参数 `axis=1` 则删除列。
- 检查重复值
  - `df.duplicated()` 返回布尔值数据框，如果一行是重复行，则对应位置的值为True，否则为False。

- 删除重复值
  - `df.drop_duplicates()` 删除重复值。其常用参数包括：
    - `subset` 指定用于删除重复行的列或列的组合。默认值为None，表示检查所有列。
    - `keep` 指定要保留的重复行的第一个或最后一个。默认值为'first'，表示保留第一次出现的重复行。'last'则保留最后一次。False则删除所有重复行。
    - `inplace` 是否要修改原始数据框。默认值为False，即返回一个新的数据框。
- 异常值处理，缩尾处理（winsorization）
  - `df.where(条件, x)` 默认返回新数据框，其中将满足条件的值保留，不满足条件的值用x替换。
  - `df.mask(条件, x)` 默认返回新数据框，其中将满足条件的值用x替换，不满足条件的值保留。
  - `df.clip(lower = x, upper = y)` 默认返回新数据框，小于lower值的元素将被x替换，大于upper值的元素将被y替换。

In [ ]:
```
# 18.9 异常值与缺失值的处理
# 处理缺失值与重复值

# 缺失值统计
df.isnull().sum()          # 数据框中每一列的缺失值
df.isnull().sum().sum()    # 数据框中的总缺失值

# 重复值统计
df.duplicated().sum()      # 重复行数
df.T.duplicated()          # 通过转置，返回重复列数

# 基于多个列删除重复值
df.drop_duplicates(subset=['col1', 'col2'])
```

In [ ]:
```
# 18.9 异常值与缺失值的处理
# 异常值缩尾处理

# 计算 20% 和 80% 分位数
q02 = df['mkvalt'].quantile(0.2)
q08 = df['mkvalt'].quantile(0.8)

# 生成布尔值数据框，选择20%和80%分位数之间的元素
con_final = (df['mkvalt'] >= q02) & (df['mkvalt'] <= q08)

# 选出符合条件的元素，生成新的数据框
df_screened = df.loc[con_final]

# 使用where()函数对数据进行缩尾处理
df_wz = df['mkvalt'].where(df['mkvalt'] >= q02, other = q02).where(df['mkvalt'] <= q08, other

# 使用mask()函数对数据进行缩尾处理
df_wz2 = df['mkvalt'].mask(df['mkvalt'] < q02, other = q02).mask(df['mkvalt'] > q08, other = q

# 使用clip()函数对数据进行缩尾处理
df_wz3 = df['mkvalt'].clip(lower = q02, upper = q08)
```

19. 例题

In [ ]:
```
# 例题1 计算列表中的几何平均数与算术平均数

return_list = input('Enter return rate').split()
return_list = [float(i) for i in return_list]

def avg_return(return_list):
    aar = round(sum(return_list) / len(return_list), 2)
    gav_sum = return_list[0]
    for i in range(len(return_list)):
        gav_sum *= return_list[i]
```

```
    gav = round((gav_sum ** (1 / len(return_list))) - 1, 2)
    return aar, gav

avg_return(return_list)
```

In [ ]:
```
# 例题2 分类列表中的奇偶数并求和

list_1 = [1, 2, 3, 4, 5]
list_1_even = list()
list_1_odd = list()
for i in range(len(list_1)):
    if(int(list_1[i]) % 2 == 0 ):
        list_1_even.append(int(list_1[i]))
    else:
        list_1_odd.append(int(list_1[i]))

print(sum(list_1_even))
```

In [ ]:
```
# 例题3 计算FV

# for loop
cf_input = input("Enter cash flow, separated by commas")
r = float(input("Enter interest rate"))/100
CF = cf_input.split()

def FV_list(CF, r):
    fv = 0
    for i in range(len(CF)):
        fv += float(CF[i]) * (1 + r) ** (len(CF) - i - 1)
    return fv
FV_list(CF, r)

#while loop
t = 0
fv = 0
while t <= (len(CF) - 1):
    fv += float(CF[- t - 1]) * (1 + r) ** t
    t += 1
print(fv)
```

In [ ]:
```
# 例题4 计算PV

# for loop
def PV_list(CF, r):
    pv = 0
    for i in range(len(CF)):
        pv += float(CF[len(CF) - i -1]) / (1 + r) ** (len(CF) - i - 1)
    return pv
PV_list(CF, r)

# while loop
t = 0
fv = 0
while t <= (len(CF) - 1):
    fv += float(CF[t]) / (1 + r) ** t
    t += 1
print(fv)
```

In [ ]:
```
# 例题5 使用numpy计算PV

import numpy as np

PV_table = np.zeros(shape=(5, 4), dtype=float)
PV_table[:, :3] = [[1000, 10, 0.05],
                   [2000, 10, 0.03],
                   [1000, 5, 0.07],
                   [2000, 5, 0.02],
```

```
                    [3000, 3, 0.1]]

for i in range(len(PV_table)):
    FV = PV_table[i][0]
    n = PV_table[i][1]
    r = PV_table[i][2]
    PV = round(FV / (1 + r)**n, 2)
    PV_table[i][3] = PV

print(PV_table)
```

In [ ]:

```
# 例题6 pandas应用

# Convert to float
will5000['WILL5000IND'] = will5000['WILL5000IND'].astype(float)

# Calculate daily returns
will5000['daily_return'] = will5000['WILL5000IND'].pct_change()

# Compute geometric average return
n1 = len(will5000['daily_return'])
will5000_gar = 1
for i in range(n1 - 1):
   will5000_gar *= (1 + will5000['daily_return'][i + 1])
will5000_gar = will5000_gar ** (1/n1) - 1
```