

Python数据分析基础

讲师名称：李川

Can//ay 嘉为数字咨询

数 字 化 人 才 培 养 先 行 者



目录

1

函数的高级
应用

2

列表与字典
高级应用

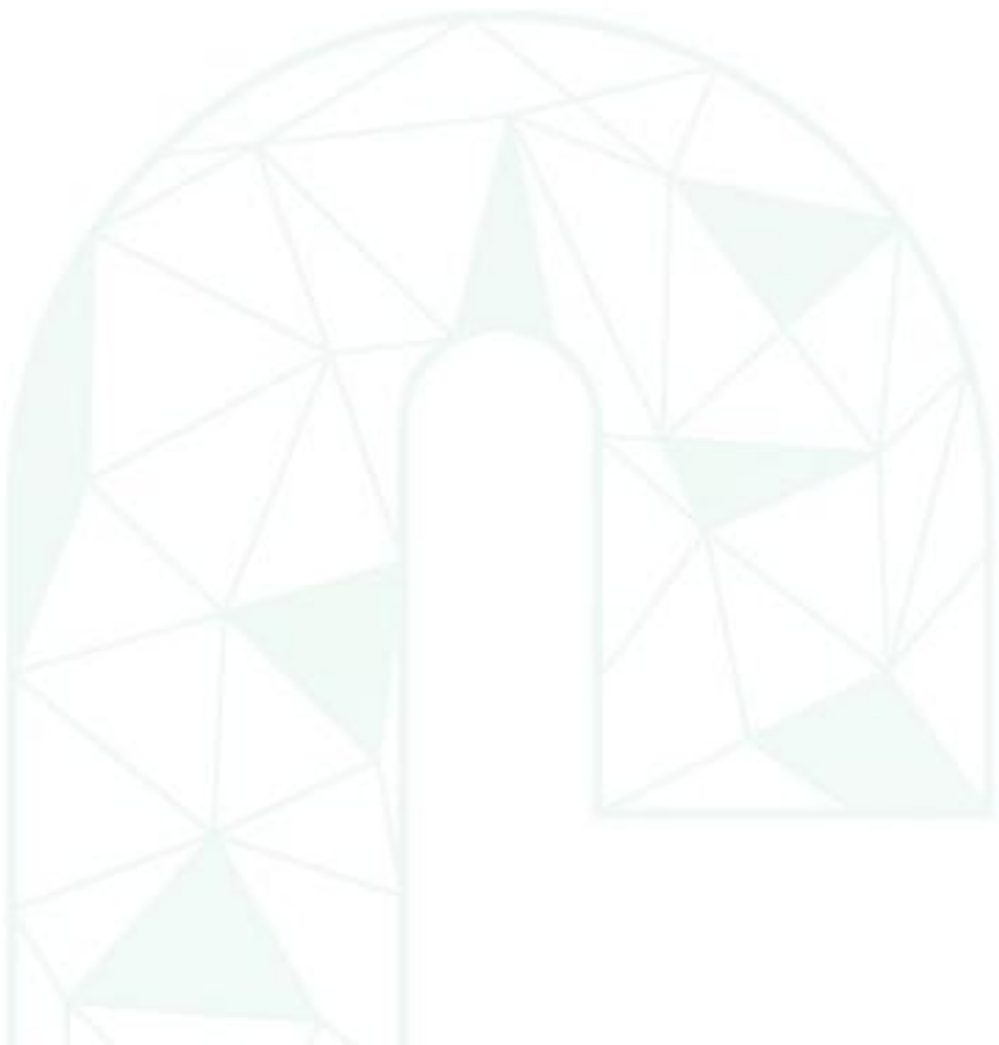
3

集合高级应
用

4

Pandas应
用

CONTENTS



/01

函数的高级应用

1.1函数

❖函数类似一个工具，本质是对输入进行处理并产生输出的可重用代码块或数学映射关系。其核心意义在于封装逻辑、提高效率、增强可读性和实现模块化。

❖函数定义语法：

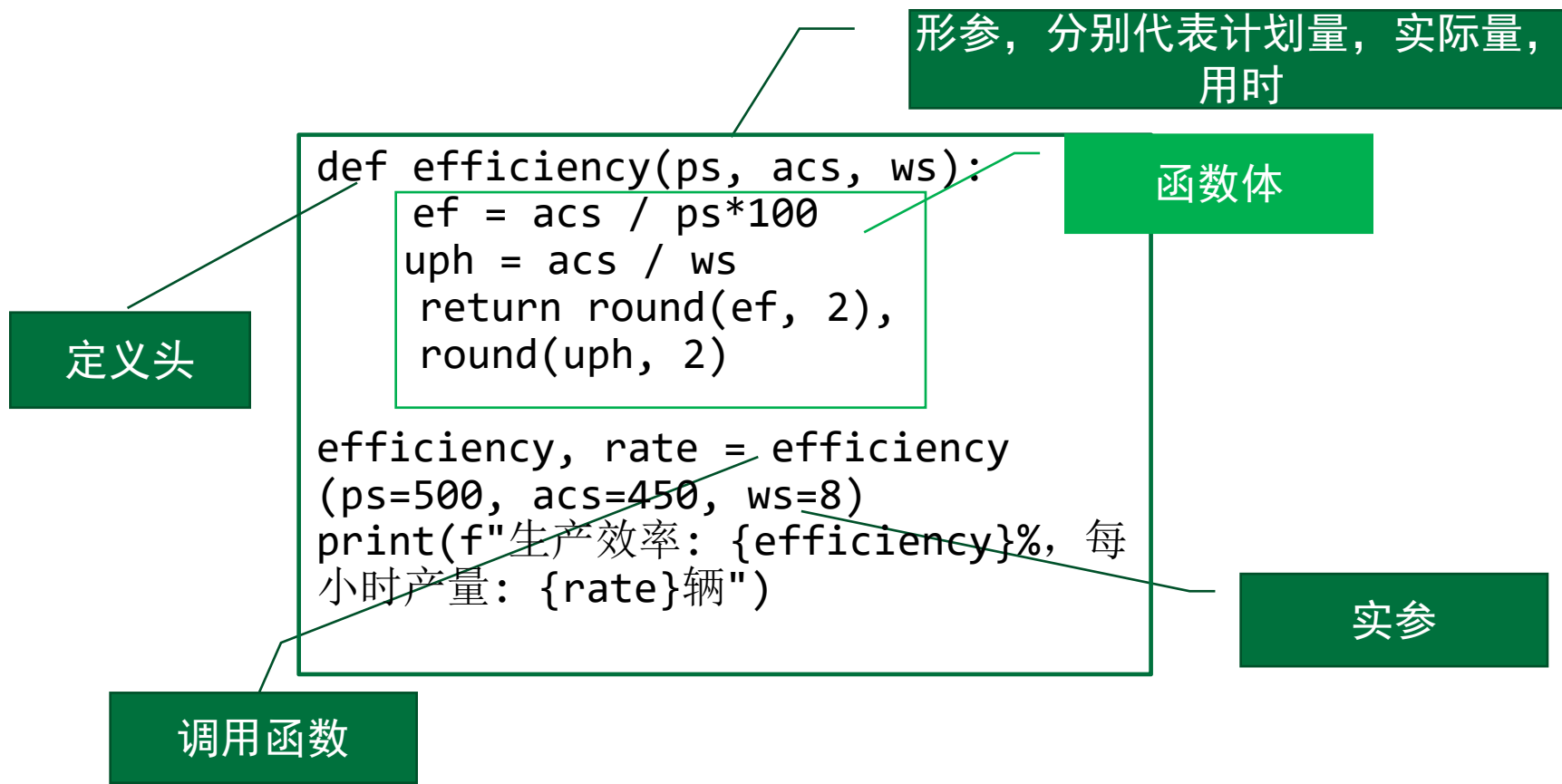
```
def 函数名([参数列表]):  
    '''注释'''  
    函数体
```

❖注意事项

- ✓函数形参不需要声明类型，也不需要指定函数返回值类型
- ✓即使该函数不需要接收任何参数，也必须保留一对空的圆括号
- ✓括号后面的冒号必不可少
- ✓函数体相对于def关键字必须保持一定的空格缩进
- ✓Python允许嵌套定义函数

1.1.1 函数定义与调用基本语法

例1-1 计算产品生产线的生产效率。



1.1.2 函数参数

- 函数定义时圆括弧内是使用逗号分隔开的形参列表（parameters），函数可以有多个参数，也可以没有参数，但定义和调用时一对圆括弧必须要有，表示这是一个函数并且不接收参数。
- 调用函数时向其传递实参（arguments），根据不同的参数类型，将实参的引用传递给形参。
- 定义函数时不需要声明参数类型，解释器会根据实参的类型自动推断形参类型。

1.1.2函数参数-位置参数

- 位置参数 (positional arguments) 是比较常用的形式，调用函数时实参和形参的顺序必须严格一致，并且实参和形参的数量必须相同。

```
>>> def demo(a, b, c):  
    print(a, b, c)
```

```
>>> demo(3, 4, 5)
```

按位置传递参数

```
3 4 5
```

```
>>> demo(3, 5, 4)
```

```
3 5 4
```

```
>>> demo(1, 2, 3, 4)
```

实参与形参数量必须相同

```
TypeError: demo() takes 3 positional arguments but 4 were given
```

1.1.2函数参数-默认值参数

- 在调用函数时，可以不用为设置了默认值的形参传递实参，此时函数将会直接使用函数定义时设置的默认值，当然也可以通过显式赋值来替换其默认值。

在调用函数时是否为默认值参数传递实参是可选的。

- 需要注意的是，在定义带有默认值参数的函数时，任何一个默认值参数右边都不能再出现没有默认值的普通位置参数，否则会提示语法错误。

1.1.2函数参数-默认值参数

- 带有默认值参数的函数定义语法如下：

```
def 函数名(....., 形参名=默认值):  
    函数体
```

1.1.2函数参数-默认值参数

```
>>> def say(message, times=1):  
    print((message+' ') * times)
```

```
>>> say('hello')
```

```
hello
```

```
>>> say('hello', 3)
```

```
hello hello hello
```

1.1.2函数参数-关键参数

- 通过关键参数可以按参数名字传递实参，明确指定哪个实参传递给哪个形参，**实参顺序可以和形参顺序不一致**，但不影响参数值的传递结果，避免了用户需要牢记参数位置和顺序的麻烦，使得函数的调用和参数传递更加灵活方便。

```
>>> def demo(a, b, c=5):  
    print(a, b, c)
```

```
>>> demo(3, 7)
```

```
3 7 5
```

```
>>> demo(a=7, b=3, c=6)
```

```
7 3 6
```

```
>>> demo(c=8, a=9, b=0)
```

```
9 0 8
```

1.2 lambda表达式

- lambda表达式可以用来声明匿名函数，也就是没有函数名字的临时使用的小函数，尤其适合需要一个函数作为另一个函数参数的场合。也可以定义具名函数。
- lambda表达式只可以包含一个表达式，该表达式的计算结果可以看作是函数的返回值，不允许包含复合语句，但在表达式中可以调用其他函数。

1.2.1 lambda表达式应用

```
>>> f = lambda x, y, z: x+y+z
```

```
>>> f(1,2,3)
```

```
6
```

```
>>> g = lambda x, y=2, z=3: x+y+z
```

```
>>> g(1)
```

```
6
```

```
>>> g(2, z=4, y=5)
```

```
11
```

可以给lambda表达式起名字

像函数一样调用

参数默认值

关键参数

1.2.1 lambda表达式应用

```
>>> L = [1,2,3,4,5]
>>> print(list(map(lambda x: x+10, L)))
[11, 12, 13, 14, 15]
>>> L
[1, 2, 3, 4, 5]
```

模拟向量运算

1.2.2 lambda表达式实例

例1-2 新能源汽车充电时，计算不同充电费用：

高峰时段：8 点到 11 点和 18 点到 23 点，每度电的收费大约在 1.6 元到 1.8 元之间。

平时段：7 点到 8 点、11 点到 18 点，每度电的收费则在 1.3 元到 1.5 元之间。

低谷时段：23 点到 7 点，每度电的收费相对较低，大概在 0.9 元到 1.2 元之间。

时段价差规则：高峰 + 0.3 元 / 度，低谷 - 0.2 元 / 度，平峰不变

输出结果直接展示三个时段的充电总费用

1.2.2 lambda表达式实例

公共充电桩费用计算

```
calc_charge = lambda capacity, base, service, period: capacity * (  
    base + service + (0.3 if period == 'peak' else -0.2 if period == 'valley' else 0)  
)
```

参数定义

```
battery_capacity = 50 # 电池容量(kWh)
```

```
base_price = 0.8      # 基础电价(元/度)
```

```
service_fee = 0.7     # 服务费(元/度)
```

计算不同时段费用

```
periods = {'peak': '高峰', 'flat': '平峰', 'valley': '低谷'}
```

```
for period, name in periods.items():
```

```
    cost = calc_charge(battery_capacity, base_price, service_fee, period)
```

```
    print(f"{name}时段充电费用: {cost:.2f}元")
```




/02

列表与字典高级应用

2.1 列表高级应用

列表 (List) 是 Python 中最基础、最常用的数据结构之一，用于存储有序、可修改、可重复的元素集合。

列表的创建与初始化

通过 `[]` 或 `list()` 函数创建列表，元素可以是任意数据类型（整数、字符串、甚至其他列表）。

1. 包含相同类型元素

```
numbers = [1, 2, 3, 4, 5] # 整数列表
```

```
fruits = ["apple", "banana", "cherry"] # 字符串列表
```

2. 包含不同类型元素（列表支持混合类型）

```
mixed = [1, "hello", 3.14, True] # 整数、字符串、浮点数、布尔值
```

3. 嵌套列表（列表中的元素可以是列表）

```
nested = [[1, 2], [3, 4], [5, 6]] # 二维列表（类似矩阵）
```

2.1.1 列表切片

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# 基本切片: [start:end] (包含start, 不包含end)
print(numbers[2:5]) # [2, 3, 4] (索引2到4的元素)
print(numbers[:3])  # [0, 1, 2] (从开头到索引2)
print(numbers[7:])   # [7, 8, 9] (从索引7到结尾)

# 步长切片: [start:end:step]
print(numbers[0:10:2]) # [0, 2, 4, 6, 8] (隔1个取1个)
print(numbers[::-1])   # [9,8,7,6,5,4,3,2,1,0] (反转列表)
```

2.1.2 列表推导式

列表推导式 (List Comprehension)

用简洁的语法创建列表，替代繁琐的 for 循环，效率更高。

基本语法：

[表达式 for 元素 in 可迭代对象 if 条件]

2.1.2 列表推导式

```
# 生成1-10的平方列表（带条件：只保留偶数的平方）
squares = [x**2 for x in range(1, 11) if x % 2 == 0]
print(squares) # [4, 16, 36, 64, 100]

# 嵌套列表推导式（二维列表转一维）
matrix = [[1, 2], [3, 4], [5, 6]]
flattened = [num for row in matrix for num in row]
print(flattened) # [1, 2, 3, 4, 5, 6]

# 字符串处理（提取列表中所有字符串的首字母大写）
words = ["apple", "banana", "cherry"]
capitalized = [word.capitalize() for word in words]
print(capitalized) # ["Apple", "Banana", "Cherry"]
```

2.2 字典高级应用

字典 (Dictionary) 是 Python 中以键值对 (Key-Value) 形式存储数据的结构, 类似现实中的 “字典” (关键词对应解释)。它的核心特点是通过 “键” 快速查找 “值”, 适合存储有映射关系的数据 (如用户信息、商品属性等)。

```
user = {  
    "name": "Alice", # 键"name"对应值"Alice"  
    "age": 25,      # 键"age"对应值25  
    "is_student": False  
}
```

嵌套字典 (值可以是字典, 实现多层结构)

```
school = {  
    "name": "Sunny School",  
    "students": {  
        "student1": {"name": "Bob", "grade": 3},  
        "student2": {"name": "Charlie", "grade": 2}  
    }  
}
```

2.2.1字典推导式

字典推导式 (Dictionary Comprehension)

快速创建字典，支持键值对的动态计算。

基本语法：

{键表达式: 值表达式 for 元素 in 可迭代对象 if 条件}

从列表生成字典 (键为元素，值为元素长度)

```
words = ["apple", "banana", "cherry"]
```

```
word_lengths = {word: len(word) for word in words}
```

```
print(word_lengths) # {"apple":5, "banana":6, "cherry":6}
```

字典过滤 (保留值大于5的键值对)

```
filtered = {k: v for k, v in word_lengths.items() if v > 5}
```

```
print(filtered) # {"banana":6, "cherry":6}
```

2.3 列表与字典的高效操作函数

- 常用的几个内建高阶函数：**filter, map, reduce**
- 内置高阶函数的优势在于：
 - 无需手动写循环，代码更简洁；
 - 逻辑清晰，将“做什么”（func）与“遍历 / 排序”等流程分离；
 - 配合 lambda 匿名函数，可一行代码实现复杂逻辑。

2.3.1 filter函数

filter(bool_func,seq): 此函数的功能相当于过滤器。调用一个布尔函数bool_func来迭代遍历每个seq中的元素; 返回一个使bool_seq返回值为true的迭代器。

其中, func_name为自定义的函数名, seq为待过滤的序列。func_name()的定义方式如下。

```
func_name(x):
```

```
    函数体
```

```
    return bool
```

使用filter()函数删除列表中的浮点数的示例如下。

```
def delete_num(x):
```

```
    return not isinstance(x, float) #isinstance() 函数判断x是否为float类型
```

```
l = [1, 1.5, 2, 2.5, 3]
```

```
print(list(filter(delete_num, l))) #将序列l中的浮点数“过滤”掉
```

程序的执行结果如下: [1, 2, 3]

使用匿名函数过滤奇数:

```
list(filter(lambda x:x%2==0,[1,2,3,4,5]))
```

```
[2, 4]
```

2.3.1 filter函数

例1-3：某员工近期想购买一辆价格在15~ 20 万元的汽车

先定义 cars 变量（汽车数据列表）

```
cars = [  
    {"name": "比亚迪 唐", "price": 18.8},  
    {"name": "本田雅阁", "price": 23.1},  
    {"name": "哈弗H6", "price": 12.5}  
]
```

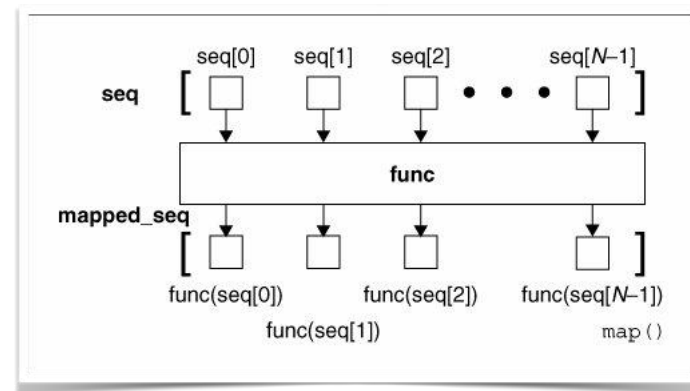
```
lowprice_cars = filter(lambda car :15<=car["price"]<=20, cars)
```

```
print([car["name"] for car in lowprice_cars])
```

2.3.2 map函数

- **map(func,seq1[,seq2...])**: 将函数func作用于给定序列的每个元素，并返回迭代器；如果func为None，func表现为身份函数，作用同zip()。返回一个含有每个序列中元素集合的n个元组的列表。

```
list(map(lambda x : None, [1, 2, 3, 4]))  
[None, None, None, None]  
list(map(lambda x : x * 2, [1, 2, 3, 4]))  
[2, 4, 6, 8]  
list(map(lambda x : x%2, [1, 2, 3, 4]))  
[1, 0, 1, 0]  
list(map(lambda x : x * 2, [1, 2, 3, 4, [5, 6, 7]]))  
[2, 4, 6, 8, [5, 6, 7, 5, 6, 7]]  
list(map(None, [1, 2, 3], [4, 5, 6]))  
[(1, 4), (2, 5), (3, 6)]  
list(zip([1, 2, 3], [4, 5, 6]))  
[(1, 4), (2, 5), (3, 6)]
```



2.3.2 map函数

当seq多于一个时，map可以并行地对每个seq执行如下图所示的过程：

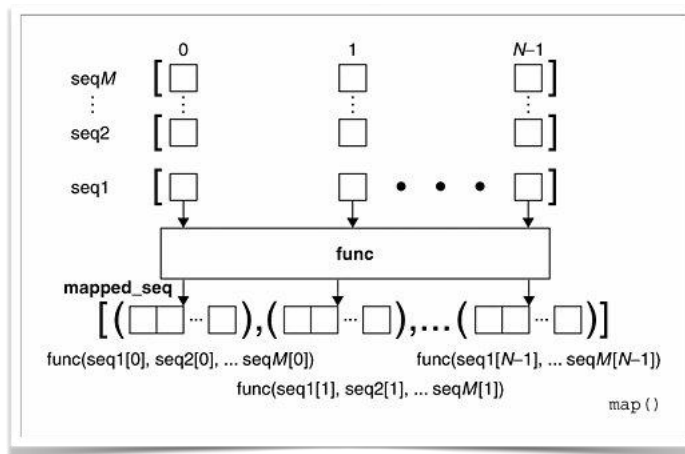
也就是说每个seq的同一位置的元素在执行过一个多元的func函数之后，得到一个返回值，这些返回值放在一个结果列表中。

```
list(map(lambda x,y:x*y, [1, 2, 3], [4, 5, 6]))
```

```
[4, 10, 18]
```

```
list(map(lambda x, y: ( x * y, x + y), [1, 2, 3], [4, 5, 6]))
```

```
[(4, 5), (10, 7), (18, 9)]
```



2.3.2 map函数

例1-4：计算不同核反应堆的年度发电量

```
def calculate_annual_output(reactor):
    """计算单座核反应堆的年度发电量（单位：兆瓦时）"""
    # 发电量 = 装机容量(MW) × 年运行小时数 × 利用率
    capacity, hours, efficiency = reactor
    return round(capacity * hours * efficiency)

# 核反应堆数据：(装机容量MW, 年运行小时数, 利用率)
reactors = [
    (1200, 7000, 0.9), # 反应堆A
    (1000, 7200, 0.88), # 反应堆B
    (1400, 6800, 0.92) # 反应堆C
]

# 使用map计算所有反应堆的年度发电量
annual_outputs = list(map(calculate_annual_output, reactors))

# 输出结果
for i, output in enumerate(annual_outputs, 1):
    print(f"反应堆{i} 年度发电量: {output} 兆瓦时")
```

2.3.3 reduce函数

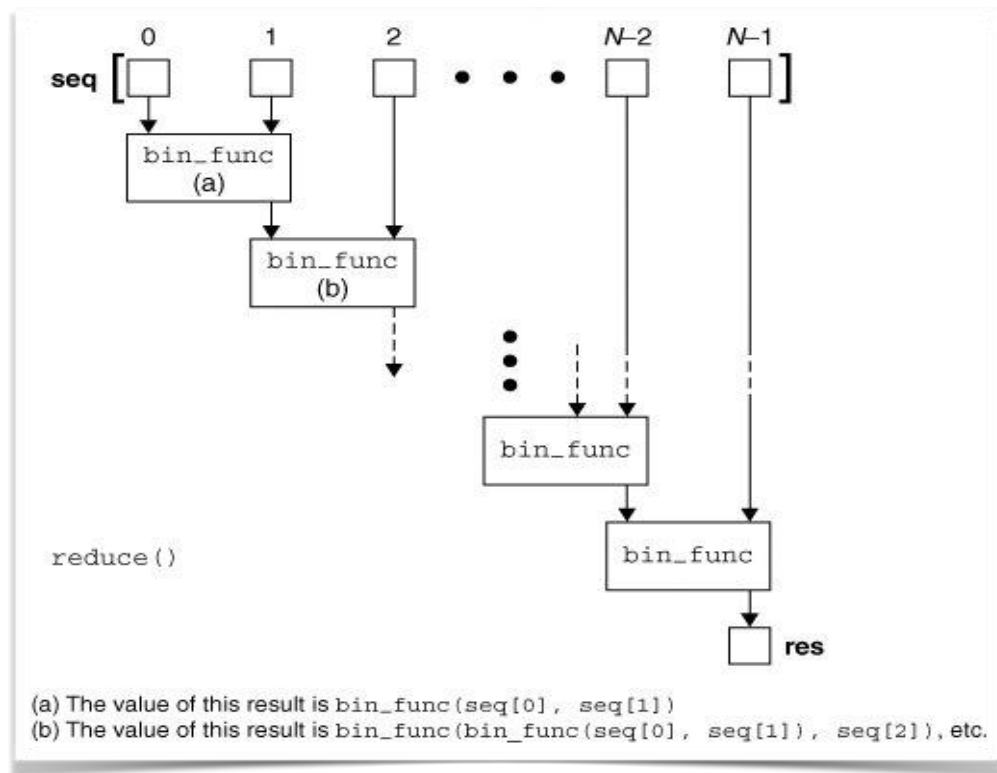
reduce(func,seq[,init]): func为二元函数，将func作用于seq序列的元素，每次携带一对（先前的结果以及下一个序列的元素），连续的将现有的结果和下一个值作用在获得的随后的结果上，最后化简序列为一个单一的返回值：如果初始值init给定，第一个比较会是init和第一个序列元素而不是序列的头两个元素。**reduce函数在python3中被放入functools包中，需用如下语句对其进行调用（python2中是基本库函数）：**

```
from functools import reduce
```

如：

```
reduce(lambda x,y:x+y,[1,2,3,4])
```

```
10
```



2.3.4 sorted函数

`sorted(iterable, key=func, reverse=False)`

功能：对可迭代对象排序，key 函数用于指定排序依据，reverse 控制升序（默认升序）。

例1-5：汽车销售价排序

先定义 cars 变量（汽车数据列表）

```
cars = [  
    {"name": "卡罗拉", "price": 15.8},  
    {"name": "本田雅阁", "price": 23.1},  
    {"name": "哈弗H6", "price": 12.5}  
]
```

按价格排序（key指定排序依据为price字段）

```
sorted_cars = sorted(cars, key=lambda x: x["price"], reverse=True)
```

打印排序结果

```
for car in sorted_cars:  
    print(f"{car['name']}: {car['price']}万")
```



/03

集合高级应用

3.1 集合简介

集合 (Set) 是 Python 中专门用于处理唯一性数据和集合运算的数据结构，其核心特性是元素不可重复且无序，非常适合实现去重、交集、并集等操作。

集合的基本特性

元素唯一：自动去重，重复元素只会保留一个

无序：不支持索引访问，不能保证元素顺序

可迭代：支持 for 循环遍历

元素类型限制：只能包含不可变类型（如整数、字符串、元组等），不能包含列表、字典等可变类型

3.2 利用集合去重

集合的自动去重特性可以快速处理列表、元组等可迭代对象中的重复元素

例1-6

列表去重

```
nums = [1, 2, 2, 3, 3, 3, 4]
```

```
unique_nums = list(set(nums)) # 先转集合去重, 再转回列表
```

```
print(unique_nums) # [1, 2, 3, 4] (注意: 顺序可能变化)
```

字符串列表去重

```
words = ["apple", "banana", "apple", "cherry", "banana"]
```

```
unique_words = list(set(words))
```

```
print(unique_words) # ["apple", "banana", "cherry"] (顺序不定)
```

保持去重后的顺序 (Python 3.7+ 可用字典特性)

```
nums = [1, 2, 2, 3, 3, 3, 4]
```

```
unique_nums_ordered = list(dict.fromkeys(nums)) # 字典键唯一且保留插入顺序
```

```
print(unique_nums_ordered) # [1, 2, 3, 4] (顺序与原列表一致)
```

3.3 集合的交集、并集、差集运算

1. 并集 (Union) : 两个集合中所有的元素 (去重)
2. 交集 (Intersection) : 两个集合中共同的元素
3. 差集 (Difference) : 属于一个集合但不属于另一个集合的元素
4. 对称差集 (Symmetric Difference) : 两个集合中互不相同的元素

3.4 集合运算的实际应用案例

例1-7：分析用户行为数据

假设有两种商品的三组用户数据：

浏览商品A的用户 (view_users_A)

商品A加入购物车的用户 (cart_users_A)

完成购买商品A的用户 (buy_users_A)

浏览商品B的用户 (view_users_B)

商品B加入购物车的用户 (cart_users_B)

完成购买商品B的用户 (buy_users_B)

通过集合运行分析用户的购买情况

3.4 集合运算的实际应用案例

例1-7 使用集合分析用户购买力

模拟商品A数据

```
view_users_A = {"u1", "u2", "u3", "u4", "u5", "u6"}
```

```
cart_users_A = {"u2", "u3", "u5", "u6", "u7"}
```

```
buy_users_A = {"u2", "u5", "u7"}
```

1. 浏览但未加入购物车的用户（潜在流失用户）

```
view_not_cart_A = view_users_A - cart_users_A
```

```
print("浏览未加购: ", view_not_cart_A) # {'u1', 'u4'}
```

2. 加入购物车但未购买的用户（高潜力转化用户）

```
cart_not_buy_A = cart_users_A - buy_users_A
```

```
print("加购未购买: ", cart_not_buy_A) # {'u3', 'u6'}
```

3. 完整转化路径用户（浏览→加购→购买）

```
full_path_A = view_users_A & cart_users_A & buy_users_A
```

```
print("完整转化用户: ", full_path_A) # {'u2', 'u5'}
```

4. 购买用户占浏览用户的比例（转化率）

```
conversion_rate_A = len(buy_users_A & view_users_A) / len(view_users_A) * 100
```

```
print(f"浏览到购买转化率: {conversion_rate_A:.1f}%") # 33.3%
```

集合运算的实际应用案例

```
# 模拟商品B数据
view_users_B = {"u1", "u2", "u4", "u6", "u7"}
cart_users_B = {"u2", "u6", "u7"}
buy_users_B = {"u2", "u6"}
# 5. 潜在关联用户
# 计算共同浏览用户 (交集)
common_users = view_users_A & view_users_B
print(f"同时浏览过A和B的用户: {common_users}, 共 {len(common_users)} 人")
# 计算A到B的相关性
if len(view_users_A) > 0:
    corr_A_to_B = len(common_users) / len(view_users_A)
    print(f"商品A到B的相关性: {corr_A_to_B:.2f} ({corr_A_to_B*100:.1f}%) ")
else:
    print("商品A没有浏览用户, 无法计算相关性")
```

集合运算的实际应用案例

```
# 计算B到A的相关性
if len(view_users_B) > 0:
    corr_B_to_A = len(common_users) / len(view_users_B)
    print(f"商品B到A的相关性: {corr_B_to_A:.2f} ({corr_B_to_A*100:.1f}%) ")
else:
    print("商品B没有浏览用户，无法计算相关性")
# 对比分析
print("\n相关性对比分析:")
if corr_A_to_B > corr_B_to_A:
    print(f"A到B的相关性 ({corr_A_to_B*100:.1f}%) 高于B到A ({corr_B_to_A*100:.1f}%) ")
elif corr_A_to_B < corr_B_to_A:
    print(f"B到A的相关性 ({corr_B_to_A*100:.1f}%) 高于A到B ({corr_A_to_B*100:.1f}%) ")
else:
    print(f"A到B与B到A的相关性相同 (均为{corr_A_to_B*100:.1f}%) ")
```



/04

Pandas应用

4 Pandas基本应用

Pandas 是 Python的核心数据分析支持库，提供了快速、灵活、明确的数据结构以及大量处理数据的函数和方法，旨在简单、直观地处理关系型、标记型数据，包括SQL、Excel表数据、各种数据文件数据（csv、hdf5等）。Pandas 的目标是成为 Python 数据分析实践与实战的必备高级工具，其长远目标是成为最强大、最灵活、可以支持任何语言的开源数据分析工具。

Pandas主要包括两种数据结构： Series和DataFrame

4.1 Series对象

在Pandas里面，Series 是一种一维标记数组,它类似于 Excel 中的一列数据，但功能更强大，需要通过pandas.Series()构造函数来创建Series对象。

pandas.Series()可以用Python中的列表、字典、常量、Numpy中的数组来创建Series对象。语法格式如下：

```
pd. Series([data, index, dtype, name, copy])
```

核心特点,包含数据和索引：

数据：可以是数值、字符串、布尔值等任意类型。

索引 (index)：每个元素对应的标签，默认是从 0 开始的整数，也可自定义（如日期、名称等）。

4.1.1 Series操作

常用操作

1. 取值与切片

类似列表，但可通过索引标签或位置访问

2. 运算与统计

支持直接对元素进行批量操作

3. 条件筛选

快速筛选符合条件的元素

4.1.2 Series应用实例

例1-8 使用 pandas 的 Series 设计一个汽车销售处理一维数据的应用

```
import pandas as pd
# 1. 创建广汽本田主要车型的Series (不同维度的数据)
# 车型指导价 (万元)
guide_price = pd.Series(
    data=[12.98, 17.98, 25.58, 37.98, 13.98],
    index=["飞度", "雅阁", "冠道", "奥德赛", "凌派"]
)
# 2. 查看Series基本信息
print("=== 广汽本田车型指导价 (万元) ===")
print(guide_price)
print("\n=== 数据类型 ===", guide_price.dtype)
print("=== 索引 ===", guide_price.index.tolist())
# 3. 基本操作: 取值与筛选
print("\n=== 雅阁的指导价 ===", guide_price["雅阁"], "万元")
print("=== 价格在20万元以上的车型 ===")
print(guide_price[guide_price > 20])
```

4.1.2 Series应用实例

```
# 5. 组合查询：获取冠道的完整信息
print("\n=== 冠道完整信息 ===")
print(f"车型：冠道")
print(f"指导价：{guide_price['冠道']}万元")
# 6. 切片查询：
print("\n=== 切片信息 ===")
print(f"1到3行")
print(f"{guide_price.iloc[1:4]}")
print(f"1, 3行")
print(f"{guide_price.iloc[[1,3]]}")
print(f"前3行")
print(f"{guide_price.iloc[:3]}")
print(f"后3行")
print(f"{guide_price.iloc[-3:]}")
print(f"输出飞度和凌派")
print(f"{guide_price.loc[['飞度','凌派']]}")
```

4.1.2 Series应用实例

7. 统计分析

```
print("\n=== 价格统计 ===")
print(f"平均指导价: {guide_price.mean():.2f}万元")
print(f"最高指导价: {guide_price.max()}万元 (车型: {guide_price.idxmax()}) ")
print(f"最低指导价: {guide_price.min()}万元 (车型: {guide_price.idxmin()}) ")
```

8. 数据修改与删除

```
print("\n=== 数据修改 ===")
print("\n雅阁涨价10%")
guide_price['雅阁']=guide_price['雅阁']*1.1
print(guide_price)
print("\n删除雅阁")
guide_price.pop('雅阁')
print(guide_price)
```

4.2 DataFrame

在 pandas 中，DataFrame 是一个二维表格型数据结构，可以理解为“带标签的表格”，类似于 Excel 工作表或数据库表。它是 pandas 中最核心的数据结构，几乎所有数据分析操作都围绕它展开。

核心特点

二维结构：包含行和列，每行代表一条记录，每列代表一个特征（如“车型”“价格”“类型”）。

标签索引：

行索引（index）：每条记录的标识（默认是 0 开始的整数）。

列索引（columns）：每列的名称（如“价格”“发动机类型”）。

异质性：不同列可以有不同的数据类型（如数值型、字符串、日期等）。

灵活操作：支持增删行列、筛选、分组、合并等多种表格操作。

4.2.1 DataFrame的属性

DataFrame对象的属性见下表，除了columns之外的其它属性，Series对象也有。

属性	说明
values	获取DataFrame中的数据，得到的是一个ndarray类型的对象
index	获取行索引
columns	获取列索引
dtypes	获取元素的类型
size	获取元素个数
ndim	维度数
shape	维度(行数和列数)

4.2.2 DataFrame常见操作

操作	DataFrame 对应操作
查看工作表	<code>print(df)</code> 或 <code>df.head()</code>
筛选行	<code>df[条件]</code>
选择列	<code>df[["列名1", "列名2"]]</code>
新增列并计算	<code>df["新列名"] = 计算表达式</code>
按列排序	<code>df.sort_values(by="列名")</code>
数据透视表	<code>df.groupby("分组列")["值列"].聚合函数()</code>

4.2.3 DataFrame应用实例

例1-9：对核工业企业的生产数据使用 DataFrame 的创建、查询、修改、排序、透视等常用操作

```
import pandas as pd
import numpy as np
# 1. 创建DataFrame：核燃料元件生产数据
data = {
    '批次号': ['FN2023001', 'FN2023002', 'FN2023003', 'FN2023004', 'FN2023005',
               'FN2023006', 'FN2023007', 'FN2023008', 'FN2023009', 'FN2023010'],
    '燃料类型': ['UO2', 'MOX', 'UO2', 'UO2', 'MOX', 'UO2', 'MOX', 'UO2', 'MOX', 'UO2'],
    '生产车间': ['一车间', '二车间', '一车间', '三车间', '二车间', '三车间', '二车间', '一车间', '三车间', '一车间'],
    '生产数量': [500, 300, 450, 520, 280, 480, 320, 510, 290, 490],
    '合格率': [0.98, 0.96, 0.99, 0.97, 0.95, 0.98, 0.97, 0.99, 0.96, 0.98],
    '生产周期(天)': [15, 20, 14, 16, 21, 15, 19, 14, 22, 15],
    '生产成本(万元)': [1250, 1800, 1125, 1300, 1680, 1200, 1920, 1275, 1740, 1225]
}
df = pd.DataFrame(data)
print("1. 原始生产数据：")
print(df.head(), "\n")
```

4.2.3 DataFrame应用实例

2. 数据查询与筛选

2.1 筛选合格率98%及以上的批次

```
high_quality = df[df['合格率'] >= 0.98]
```

```
print("2.1 合格率≥98%的批次：")
```

```
print(high_quality[['批次号', '燃料类型', '合格率']], "\n")
```

2.2 筛选UO2类型且生产数量超过480的批次

```
uo2_large = df[(df['燃料类型'] == 'UO2') & (df['生产数量'] > 480)]
```

```
print("2.2 UO2类型且产量>480的批次：")
```

```
print(uo2_large[['批次号', '生产数量', '生产车间']], "\n")
```

4.2.3 DataFrame应用实例

3. 数据修改与新增列

3.1 计算合格数量和单位成本（确保无NaN后再转换为int）

```
df['合格数量'] = (df['生产数量'] * df['合格率']).astype(int)
```

```
df['单位成本(元/个)'] = (df['生产成本(万元)'] * 10000 / df['生产数量']).round(2)
```

3.2 修改生产周期异常值（超过20天的修正为20）

```
df.loc[df['生产周期(天)'] > 20, '生产周期(天)'] = 20
```

```
print("3. 修改后的数据（新增列和修正后）：")
```

```
print(df[['批次号', '合格数量', '单位成本(元/个)', '生产周期(天)']].head(), "\n")
```

4.2.3 DataFrame应用实例

4. 数据排序

按合格率降序、生产数量升序排序

```
sorted_df = df.sort_values(by=['合格率', '生产数量'], ascending=[False, True])
```

```
print("4. 按合格率降序、生产数量升序排序：")
```

```
print(sorted_df[['批次号', '燃料类型', '合格率', '生产数量']].head(), "\n")
```

4.2.3 DataFrame应用实例

5. 数据分组与聚合

5.1 按燃料类型分组, 计算各指标平均值

```
fuel_type_stats = df.groupby('燃料类型').agg({  
    '生产数量': 'mean',  
    '合格率': 'mean',  
    '生产成本(万元)': 'sum',  
    '合格数量': 'sum'  
}).round(2)
```

```
print("5.1 按燃料类型统计: ")
```

```
print(fuel_type_stats, "\n")
```

5.2 按车间和燃料类型分组, 计算平均生产周期

```
workshop_fuel_stats = df.groupby(['生产车间', '燃料类型'])['生产周期(天)'].mean().unstack()
```

```
print("5.2 车间×燃料类型的平均生产周期: ")
```

```
print(workshop_fuel_stats.fillna('-'), "\n")
```

4.2.3 DataFrame应用实例

7. 数据删除

7.1 删除列 (删除"合格数量"列)

```
df = df.drop(columns=["合格数量"])
```

7.2 删除行 (删除生产成本高于1900万元的数据)

```
df = df.drop(df[df["生产成本(万元)"] >= 1900].index)
```

```
print("7.删除后的数据：")
```

```
print(df)
```

```
print(df[["批次号", "燃料类型", "生产成本(万元)"]])
```

4.2.3 DataFrame应用实例

6. 透视表分析（修复NaN转换问题）

```
pivot_table = df.pivot_table(  
    index='生产车间',  
    columns='燃料类型',  
    values='合格数量',  
    aggfunc='sum',  
    margins=True,  
    margins_name='合计'  
)
```

处理透视表中的NaN值后再转换为整数

```
pivot_table_clean = pivot_table.fillna(0).astype(int) # 关键修复：先用0填充NaN  
print("6. 各车间不同燃料类型的合格数量汇总：")  
print(pivot_table_clean)
```


4.3 pandas读写文件数据

- 在Pandas中，可通过read_csv()函数完成csv文件的读取，可通过to_csv()函数完成csv文件的写入。
- 在Pandas中，可通过read_excel()函数实现Excel文件的读取，可通过to_excel()函数实现Excel文件的写入。
- 需要通过Pandas中的read_json()函数来实现json文件的读取，通过DataFrame_obj的to_json()方法来实现json文件的写入。

The logo features a central green diamond containing the text. To the left of the diamond is a dark green chevron pointing right. To the right is a light green chevron pointing left. A horizontal line passes through the center of the composition. Several small diamonds in dark green, light green, and grey are positioned around the main elements.

Canllay 嘉为数字咨询

数字化人才培养
先行者