by zjn in 2025/5/29

Python与大数据分析 课程重点框架

Python与大数据分析 --python基本语法及变量

一、Python的基本语法

1. 注释 **(Comments)**
    - 作用：方便对代码的理解，对程序功能进行说明。
    - 单行注释：以 `#` 开头。

```
# 这是一行注释
a = 1 # 这也是注释
```

    - 多行注释：使用三个单引号 `'''` 或三个双引号 `"""` 将注释内容括起来。

```
'''
这是多行注释的第一行
这是多行注释的第二行
a = 3
b = 4
'''
"""
这也是一个多行注释的例子
"""
```

2. 行与缩进 **(Indentation)**
    - 作用：Python使用缩进来表示代码块，替代了其他语言中的大括号 `{}`。
    - 规则：同一代码块的语句必须包含相同的缩进空格数。通常使用4个空格作为一级缩进。
    - 示例：

```python
a = 3
b = 5
if a > b:
    print('a 大于 b') # 缩进表示这是if语句块的内容
    print('继续在if语句块中')
else:
    print('a 不大于 b') # 缩进表示这是else语句块的内容
    if b == 5:
        print('b 等于 5') # 嵌套代码块需要进一步缩进
    print('继续在else语句块中')
```

3. **print() 函数**

- 作用：将指定的内容输出到控制台。
- 基本用法：

```python
print("Hello, Python!")
x = 10
print(x)
```

- **控制换行 (end参数)**：默认情况下，`print()` 会在输出末尾添加换行符 `\n`。可以使用 `end` 参数指定其他结尾字符。

```python
print("Data", end=" ") # 输出末尾是空格，而不是换行
print("whale")
# 输出: Data whale

print("Data", end="***")
print("whale")
# 输出: Data***whale
```

- **输出多个内容 (sep参数)**：`print()` 可以一次输出多个值，默认使用空格分隔。可以使用 `sep` 参数指定其他分隔符。

```python
print("Data", "whale")
# 输出: Data whale

print("Data", "whale", sep="***")
# 输出: Data***whale
```

- **格式化输出 (f-strings)**：在字符串前加 `f` 或 `F`，可以在字符串中通过 `{}` 嵌入变量或表达式。

```python
x = 1
y = 2
print(f"一个简单的数学问题: '{x} + {y} = ?', 答案是 {x+y}!")
# 输出: 一个简单的数学问题: '1 + 2 = ?', 答案是 3!

name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
# 输出: My name is Alice and I am 30 years old.
```

- 字符串拼接与重复：

```python
print("Data" + "whale" + "P2S") # 字符串拼接
# 输出: DatawhaleP2S

print("p2s" * 2, "data" * 3, sep="////") # 字符串重复和自定义分隔符
# 输出: p2sp2s////datadatadata
```

- 多行字符串输出：使用三个引号可以定义多行字符串，`print()` 会按原样输出。

```python
print("""
Python is powerful... and fast;
plays well with others;
runs everywhere;
is friendly & easy to learn;
is Open.
""")
```

4. **input() 函数 (补充)**

- 作用：从控制台读取用户输入，返回的是字符串类型。
- 基本用法：

```python
name = input("请输入您的名字: ")
print(f"您好, {name}!")
```

- 读取多个输入值：结合 `split()` 方法可以读取一行中的多个值。

```python
# 假设用户输入: 1*2
a, b = input("请输入两个数字，用*分隔: ").split("*")
print(f"a = {a}, b = {b}")
# 注意 a 和 b 此时是字符串类型
# 如果需要进行数学运算，需要转换为数值类型:
# num_a = int(a)
# num_b = int(b)
```

5. 自变量命名规范 **(Variable Naming Conventions)**

- 规则：
  - 只能以字母（大小写敏感）或下划线 `_` 开头。
  - 其余部分可以是字母、数字或下划线 `_`。
  - 不能包含空格。
  - 不能使用Python的保留字（关键字）。
  - 建议：变量名应具有描述性（例如 `user_age` 而不是 `ua`）。
- 正确示例：

```python
my_variable = 10
_count = 5
userAge = 25
MAX_VALUE = 100
```

- 错误示例：

```python
# 1_variable = 10  # 以数字开头
# my variable = 20 # 包含空格
# for = 5          # 使用保留字
```

- **保留字 (Keywords)**：Python有一些具有特殊含义的词，不能用作变量名。可以通过 `keyword` 模块查看。

```python
import keyword
print(keyword.kwlist)
# 输出: ['False', 'None', 'True', 'and', 'as',
'assert', ...]
```

- **避免覆盖内置函数名**：不要使用Python内置函数名（如 `list`, `str`, `print`）作为变量名，否则会覆盖原有功能。

```
# print("原始print函数")
# original_print = print # 可以先保存原始函数
# print = 1 # 覆盖了print函数
# # print(print) # 此时会报错，因为print现在是整数1
# # original_print(print) # 使用保存的原始函数输出1
# del print # 删除自定义变量，恢复内置函数
# print("恢复后的print函数")
```

6. 运算符 **(Operators)**

- 算术运算符：
  - `+` (加), `-` (减), `*` (乘), `/` (除 - 结果总是浮点数)
  - `%` (取模 - 返回除法的余数)
  - `**` (幂 - 例如 `2**3` 结果是 8)
  - `//` (取整除 - 返回商的整数部分，向下取整)

```
a = 10
b = 3
print(f"{a} + {b} = {a + b}")    # 13
print(f"{a} / {b} = {a / b}")    # 3.333...
print(f"{a} // {b} = {a // b}")  # 3
print(f"{a} % {b} = {a % b}")    # 1
print(f"{a} ** {b} = {a ** b}")  # 1000
```

- 比较运算符：返回布尔值 (`True` 或 `False`)
  - `==` (等于), `!=` (不等于)
  - `>` (大于), `<` (小于)
  - `>=` (大于等于), `<=` (小于等于)

```
x = 5
y = 10
print(f"{x} == {y} is {x == y}") # False
print(f"{x} < {y} is {x < y}")   # True
```

- 逻辑运算符：
  - `and` (逻辑与：两边都为 `True` 时结果为 `True`)
  - `or` (逻辑或：两边至少一个为 `True` 时结果为 `True`)
  - `not` (逻辑非：取反)

```python
p = True
q = False
print(f"{p} and {q} is {p and q}") # False
print(f"{p} or {q} is {p or q}")   # True
print(f"not {p} is {not p}")       # False
```

- 赋值运算符：
  - `=` (基本赋值)
  - `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `//=` (复合赋值，例如 `x += 1` 等价于 `x = x + 1`)

```python
count = 0
count += 5 # count 现在是 5
print(count)
```

- 成员运算符：
  - `in` (如果在序列中找到值则返回 `True`)
  - `not in` (如果在序列中未找到值则返回 `True`)

```python
my_list = [1, 2, 3, 4]
print(3 in my_list)       # True
print(5 not in my_list)   # True
```

- 身份运算符：
  - `is` (如果两个变量引用同一个对象则返回 `True`)
  - `is not` (如果两个变量引用不同对象则返回 `True`)

```python
list1 = [1, 2]
list2 = [1, 2]
list3 = list1
print(list1 == list2) # True (值相等)
print(list1 is list2) # False (不是同一个对象)
print(list1 is list3) # True (是同一个对象)
```

二、Python的数据类型

1. 基本数据类型简介

- Python变量不需要预先声明类型，其类型由赋给它的值决定。
- 每个变量在使用前都必须赋值。

2. 整数 **(int)**

- 表示整数，可正可负，没有大小限制（受限于内存）。
- 示例：`age = 30`, `negative_num = -100`

3. 浮点数 **(float)**

- 表示带有小数点的数字，或使用科学计数法表示的数字。
- 示例：`price = 19.99`, `pi_approx = 3.14`, `scientific_notation = 1.2e5` (表示 1.2 * 10^5)
- 注意：浮点数运算可能存在精度问题。

4. 布尔值 **(bool)**

- 只有两个值：`True` 和 `False`（首字母大写）。
- 通常是比较运算或逻辑运算的结果。
- 在数字上下文中，`True` 被视为 `1`，`False` 被视为 `0`。
- 示例：`is_active = True`, `has_error = False`, `result = (4 > 3)` (result 为 `True`)

5. 字符串 **(str)**

- 表示文本数据，由单引号 `'...'` 或双引号 `"..."` 括起来。
- 多行字符串可以使用三个单引号 `'''...'''` 或三个双引号 `"""..."""`。
- 字符串是不可变的（一旦创建，其内容不能修改）。
- 常用操作：
    - 拼接：`"Hello" + " " + "World"`
    - 重复：`"Ha" * 3` (结果是 `"HaHaHa"`)
    - 索引和切片：`my_string = "Python"`, `my_string[0]` (是 `'P'`), `my_string[1:4]` (是 `'yth'`)
    - 内置方法：`split()`, `find()`, `upper()`, `lower()`, `replace()` 等。

```python
greeting = "Hello"
name = "Alice"
message = greeting + ", " + name + "!" # "Hello, Alice!"
print(message)

s = "I love data structure"
print(s.split())      # ['I', 'love', 'data', 'structure']
print(s.find('love')) # 2 (返回子串首次出现的索引，未找到则返回-1)
print(s.upper())      # "I LOVE DATA STRUCTURE"
```

6. 常量 **(Constants)**

- Python没有严格意义上的常量声明方式，但约定俗成使用全大写字母的变量名表示常量。
- 内置常量：
  - `True`: 布尔真。
  - `False`: 布尔假。
  - `None`: 表示空值或无值，不同于 `0` 或空字符串。

```python
is_valid = True
data_found = False
user_input = None
```

- `math` 模块中的常量：

```python
import math
print(math.pi)  # 圆周率 π
print(math.e)   # 自然对数的底 e
print(math.tau) # 2π
print(math.inf) # 正无穷大
print(-math.inf)# 负无穷大
```

7. 类型影响语义 **(Type Affects Semantics)**

- 运算符的行为（语义）取决于操作数的数据类型。
- 示例：`+` 运算符
  - 对于数字，是加法：`3 + 2` 结果是 `5`。

- 对于字符串，是拼接：`"Data" + "whale"` 结果是 `"Datawhale"`。
- 不同类型之间直接运算通常会导致 `TypeError`：`3 + "p2s"` 会报错。

8. 短路求值 **(Short-Circuit Evaluation)**

- `and` 运算符：如果第一个操作数为 `False`，则整个表达式必为 `False`，Python不会计算第二个操作数。
- `or` 运算符：如果第一个操作数为 `True`，则整个表达式必为 `True`，Python不会计算第二个操作数。
- 示例：

```
def print_and_return_true():
    print("Executing True function")
    return True

def print_and_return_false():
    print("Executing False function")
    return False

print("Testing 'and':")
result_and = print_and_return_false() and print_and_return_true() # "Executing False function" 会打印，第二个函数不会执行
print(f"Result of 'and': {result_and}\n")

print("Testing 'or':")
result_or = print_and_return_true() or print_and_return_false() # "Executing True function" 会打印，第二个函数不会执行
print(f"Result of 'or': {result_or}")
```

9. 判断类型 **(Type Checking)**

- `type()` 函数：返回对象的类型。

```
print(type("p2s") == str) # True
print(type(123) == int)   # True
```

- `isinstance()` 函数：检查一个对象是否是指定类或其子类的实例。通常比 `type()` 更推荐，因为它能正确处理继承关系。

```python
print(isinstance("p2s", str)) # True
print(isinstance(123, int))   # True
print(isinstance(True, bool)) # True
print(isinstance(True, int))  # True, 因为bool是int的子类

import numbers
def is_number(x):
    return isinstance(x, numbers.Number) # numbers.Number 是所有数字类型的基类

print(is_number(1), is_number(1.1), is_number(1+2j), is_number("p2s"))
# 输出: True True True False
```

10. 不同数据类型转换 **(Type Conversion / Casting)**

- **隐式类型转换 (Implicit)**：Python自动进行的类型转换，通常发生在不同数字类型混合运算时，较低精度类型会转换为较高精度类型以避免数据丢失。

```python
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo # int被提升为float
print(f"num_new的值: {num_new}, 数据类型: {type(num_new)}")
# 输出: num_new的值: 124.23, 数据类型: <class 'float'>
```

- **显式类型转换 (Explicit)**：使用内置函数强制转换数据类型。
  - `int(x)`: 将 `x` 转换为整数。浮点数转换会截断小数部分。字符串必须表示整数。
  - `float(x)`: 将 `x` 转换为浮点数。
  - `str(x)`: 将 `x` 转换为字符串。
  - `bool(x)`: 将 `x` 转换为布尔值。对于数字，`0` 为 `False`，其他为 `True`。对于序列（字符串、列表等），空序列为 `False`，非空为 `True`。`None` 为 `False`。

```python
a = -2023.5
print(f"Original: {a}, type: {type(a)}")

b = int(a) # 转换为整型（向下取整）
print(f"int(a): {b}, type: {type(b)}") # -2023

c = bool(a) # 转换为布尔型（非零数为True）
print(f"bool(a): {c}, type: {type(c)}") # True
print(f"bool(0): {bool(0)}, bool(''): {bool('')}, bool(None): {bool(None)}") # False, False, False

d = str(a) # 转换为字符串型
print(f"str(a): {d}, type: {type(d)}") # "-2023.5"
```

- 统一数据类型进行运算：

```python
num_int = 123
num_str = "456"

# 错误示例: print(num_int + num_str) # TypeError

# 转换为字符串拼接
print(str(num_int) + num_str) # "123456"

# 转换为整数相加
print(num_int + int(num_str)) # 579
```

三、逻辑语句与循环

1. `if` 条件语句

- 作用：根据一个或多个条件的真假来决定执行哪个代码块。
- 基本形式 (`if-else`)：

```
# 语法:
# if 判断条件:
#     执行语句...
# else:
#     执行语句...

results = 59
if results >= 60:
    print('及格')
else:
    print('不及格') # 输出: 不及格
```

- **真值判断**：Python中，任何非零数字和非空对象（如非空字符串、列表）被视为 `True`，而 `0`、`None` 和空对象被视为 `False`。

```
num = 6
if num: # num不为0，所以为True
    print('Hello Python') # 输出: Hello Python
else:
    print('It is false')

num = 0
if num:
    print('Hello Python')
else:
    print('It is false') # 输出: It is false
```

- **多条件判断 (`if-elif-else`)**：`elif` 是 `else if` 的缩写。

```
# 语法:
# if 判断条件1:
#     执行语句1...
# elif 判断条件2:
#     执行语句2...
# else:
#     执行语句N...

results = 99
if results > 90:
    print('优秀') # 输出: 优秀
elif results > 80:
    print('良好')
elif results >= 60:
```

```
    print('及格')
else:
    print('不及格')
```

- 逻辑运算符组合条件（`and`, `or`）：

```
java_score = 86
python_score = 68

if java_score > 80 and python_score > 80:
    print('双科优秀')
else:
    print('至少一科未达优秀') # 输出

if (java_score >= 80 and java_score < 90) or \
   (python_score >= 80 and python_score < 90):
    print('至少一科良好') # 输出
```

- 示例应用：一元二次方程求解个数

```
def number_of_roots(a, b, c):
    # 返回 y = a*x**2 + b*x + c 的实数根(零点)数量
    if a == 0: # 不是二次方程
        if b == 0:
            return 0 if c != 0 else float('inf')
# 0x=c, 无解或无穷解
        else:
            return 1 # 一次方程有一个解

    delta = b**2 - 4*a*c
    if delta > 0:
        return 2
    elif delta == 0:
        return 1
    else: # delta < 0
        return 0

print(f"y = 4*x**2 + 5*x + 1 has
{number_of_roots(4, 5, 1)} root(s).") # 2
print(f"y = 4*x**2 + 4*x + 1 has
{number_of_roots(4, 4, 1)} root(s).") # 1
print(f"y = 4*x**2 + 3*x + 1 has
{number_of_roots(4, 3, 1)} root(s).") # 0
```

2. `match...case` 语句 **(Python 3.10+)**

- 作用：提供了一种更结构化的方式来进行多路分支判断，类似于其他语言的 `switch-case`。
- 语法：

```
# match subject:
#     case <pattern_1>:
#         <action_1>
#     case <pattern_2> if <condition>: # 带守卫的case
#         <action_2>
#     case <pattern_3> | <pattern_4>: # 或模式
#         <action_3>
#     case _: # 通配符，匹配任何未被以上case匹配的情况(类似default)
#         <action_wildcard>
```

- 示例：**HTTP状态码处理**

```
def http_status(status):
    match status:
        case 200:
            return "OK"
        case 400:
            return "Bad Request"
        case 404:
            return "Not Found"
        case 418:
            return "I'm a teapot"
        case _: # Default case
            return "Something's wrong with the internet"

print(http_status(200)) # OK
print(http_status(404)) # Not Found
print(http_status(500)) # Something's wrong with the internet
```

- 示例：学生分数等级 **(替代if-elif)**

```
def get_grade_match_case(score):
```

```
    match score:
        case s if s >= 90: # 使用守卫 (guard)
            grade = "A"
        case s if s >= 80:
            grade = "B"
        case s if s >= 70:
            grade = "C"
        case s if s >= 60:
            grade = "D"
        case _:
            grade = "F"
    return grade

# print(f"103 --> {get_grade_match_case(103)}") # A (假设分数上限是100，这里仅为演示)
# print(f"88 --> {get_grade_match_case(88)}")   # B
```

3. `while` 循环语句

- 作用：当给定条件为 `True` 时，重复执行循环体内的代码块。
- 基本形式：

```
# while 判断条件:
#     执行语句...
#     (通常需要有改变判断条件的语句，以避免死循环)
```

- 示例1：打印1到9的奇数

```
a = 1
while a < 10:
    print(a)
    a += 2
# 输出:
# 1
# 3
# 5
# 7
# 9
```

- 示例2：处理列表元素

```python
numbers = [12, 37, 5, 42, 8, 3]
even = []
odd = []
while len(numbers) > 0:
    number = numbers.pop() # 从列表末尾取出一个元素并删除
    if number % 2 == 0:
        even.append(number)
    else:
        odd.append(number)
print(f"Even numbers: {even}") # [8, 42, 12] (顺序与pop有关)
print(f"Odd numbers: {odd}")   # [3, 5, 37]
```

- 示例3：查找第n个质数 (结合函数)

```python
def is_prime_simple(num): # 简单的质数判断函数
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

def nth_prime(n):
    found = 0
    guess = 0
    while found <= n: # 注意PDF中是 found <= n，通常找第n个是 found < n 或 found == n
        guess += 1
        if is_prime_simple(guess):
            found += 1
    return guess

# for i in range(10): # 找第0到第9个质数 (如果从第0个开始计数)
#     print(f"The {i}-th prime is: {nth_prime(i)}")
# print(f"The 5th prime is: {nth_prime(5)}") # 如果n=0是第一个质数，则第5个是13
```

4. `for` 循环语句

- 作用：遍历任何序列（如列表、元组、字符串）或可迭代对象的项目。
- 基本形式：

```
# for iterating_var in sequence:
#     statements(s)
```

- 示例1：遍历字符串

```
for letter in '你好! python':
    print(letter, end=" ") # 你 好 !   p y t h o n
print()
```

- 示例2：遍历列表

```
fruits = ['banana', 'apple', 'mango']
# 方式一：通过索引
for index in range(len(fruits)):
    print(f'当前水果 (索引 {index}):
{fruits[index]}')

# 方式二：直接遍历元素 (更Pythonic)
for fruit in fruits:
    print(f'当前水果: {fruit}')
```

- 示例3：结合 `range()` 和 `if-else`

```
for i in range(5): # range(5) 生成 0, 1, 2, 3, 4
    if i % 2 == 0:
        print(f'{i} 是偶数')
    else:
        print(f'{i} 是奇数')
```

5. `range()` 函数

- 作用：生成一个整数序列，常用于 `for` 循环中控制迭代次数。
- 形式：
  - `range(stop)`：生成从 `0` 到 `stop-1` 的整数。
  - `range(start, stop)`：生成从 `start` 到 `stop-1` 的整数。
  - `range(start, stop, step)`：生成从 `start` 到 `stop-1` 的整数，步长为 `step`。`step` 可以是负数。
- 示例：求和

```
def sum_from_m_to_n(m, n):
    return sum(range(m, n + 1)) # n+1 是因为range不包含stop值
print(sum_from_m_to_n(5, 10)) # 45
```

```
4
5  def sum_to_n(n): # start默认为0
6      total = 0
7      for x in range(n + 1):
8          total += x
9      return total
10 print(sum_to_n(5)) # 15
11
12 def sum_every_kth_from_m_to_n(m, n, k): # 带步长
13     total = 0
14     for x in range(m, n + 1, k):
15         total += x
16     return total
17 print(sum_every_kth_from_m_to_n(5, 20, 7)) # 5 +
   12 + 19 = 36
18
19 # 反向序列
20 for i in range(5, 0, -1): # 5, 4, 3, 2, 1
21     print(i, end=" ")
22 print()
```

6. 条件与循环综合实例

- 示例：打印九九乘法表

```
1 for i in range(1, 10):
2     for j in range(1, i + 1):
3         print(f'{j}x{i}={i*j}\t', end='') # 使用f-
  string，\t是制表符
4     print() # 每行结束后换行
```

- 示例：判断是否是闰年

```
1 # year = int(input("请输入一个年份: "))
2 # if (year % 4 == 0 and year % 100 != 0) or (year
  % 400 == 0):
3 #     print(f'{year}是闰年')
4 # else:
5 #     print(f'{year}不是闰年')
```

7. 控制循环语句

- `break`：立即终止当前所在的整个循环（`for` 或 `while`），并跳出循环体。如果循环有 `else` 子句，`else` 子句也不会执行。

```python
for num in range(10, 20):  # 迭代10到19之间的数字
    for i in range(2, num): # 根据因子迭代
        if num % i == 0:    # 确定第一个因子
            j = num // i    # 计算第二个因子（使用整除）
            print(f'{num} 等于 {i} * {j}')
            break # 跳出内部的for循环（寻找下一个num）
    else: # 内部循环正常结束（没有被break）时执行
        print(f'{num} 是一个质数')
```

- `continue`：立即终止当前迭代，跳过循环体中尚未执行的语句，并开始下一次迭代。

```python
# 统计1到10之间的奇数和
current_sum = 0
for count in range(1, 11):
    if count % 2 == 0: # 如果是偶数
        continue       # 跳过本次循环的剩余部分，直接开始下一次迭代
    current_sum += count
print(f"1到10之间的奇数和为: {current_sum}") # 1+3+5+7+9 = 25
```

- `pass`：空语句，不做任何事情，用作占位符。当语法上需要一条语句但程序不需要执行任何操作时使用。

```python
for letter in 'Python':
    if letter == 'h':
        pass # 遇到'h'时什么也不做，只是占位
        print('这是 pass 块内部，在pass之后')
    print('当前字母:', letter)

# 另一个例子
# def my_function():
#     pass # 以后再来实现这个函数
```

- 循环的 `else` 子句 (补充)：
    - `for` 循环：当循环正常完成所有迭代（即没有被 `break` 语句中断）时，执行 `else` 子句。
    - `while` 循环：当循环条件变为 `False`（即循环正常终止，没有被 `break` 语句中断）时，执行 `else` 子句。

```python
for i in range(5):
    print(i)
    if i == 10: # 这个条件永远不会满足
        break
else:
    print("For循环正常结束") # 会执行

count = 0
while count < 3:
    print(count)
    count += 1
    # if count == 1:
    #     break
else:
    print("While循环正常结束") # 会执行 (如果break未注释掉，则不会执行)
```

8. 循环语句总结

- `for` 循环：
  - **优点**：语法简洁，易于遍历可迭代对象，自动处理索引。
  - **缺点**：对于依赖条件而非确定次数的循环不够直观。
  - **适用场景**：确切知道循环次数或需要遍历集合元素时。
- `while` 循环：
  - **优点**：灵活，适用于基于条件的循环，可创建无限循环（需 `break`）。
  - **缺点**：可能导致死循环，代码可能较 `for` 冗长。
  - **适用场景**：循环次数不确定，取决于特定条件时。
- `range()`：
  - **优点**：简洁明了，适合迭代固定次数，可读性高。
  - **缺点**：仅限数值序列。
  - **适用场景**：需要按顺序生成数字作为循环计数器。

四、函数定义 (初步)

1. 基本形式

- 使用 `def` 关键字定义函数。
- 函数名后跟圆括号 `()`，括号内可以包含参数列表。

- 函数体代码块需要缩进。
- 使用 `return` 语句返回一个值。如果函数没有 `return` 语句，或者 `return` 语句后没有值，则默认返回 `None`。

```
# def 函数名(参数1, 参数2, ...):
#     """可选的文档字符串 (docstring)，描述函数功能"""
#     函数体语句...
#     return 返回值
```

2. 示例1：两数之和

```
def sum_two_numbers(num1, num2):
    """返回两个数字的和。"""
    return num1 + num2

# 调用函数
result = sum_two_numbers(5, 6)
print(f"5 + 6 = {result}") # 输出: 5 + 6 = 11
```

3. 示例2：无参数，无显式返回 (默认返回 `None`)

```
def greet():
    """打印问候语。"""
    print('Hello world')

greet() # 调用函数，输出: Hello world
return_value = greet()
print(f"greet()的返回值是: {return_value}") # greet()的返回值是: None
```

4. 示例3：参数默认值

- 可以在定义函数时为参数指定默认值。如果调用函数时未提供该参数的值，则使用默认值。

```python
def factorial(n=10): # n的默认值是10
    """计算n的阶乘，默认计算10的阶乘。"""
    x = 1
    for i in range(1, n + 1):
        x *= i
    return x

result1 = factorial() # 未传入参数，使用默认n=10
print(f'10的阶乘为: {result1}') # 3628800

result2 = factorial(5) # 传入参数5，覆盖默认值
print(f'5的阶乘为: {result2}') # 120
```

Python与大数据分析 --python函数的使用

1. 函数的含义、定义与调用

- **函数含义** (课件P3)： 函数是一段具有特定功能的、命名的代码序列（或称为流程、过程）。它主要由两部分组成：
    - **头部 (Header)** (课件P4)：定义函数的接口，包括使用 `def` 关键字、函数名和圆括号内的参数列表。参数是函数在被调用时接收的输入值。函数头部以冒号 `:` 结尾。
    - **主体 (Body)** (课件P5)：包含实现函数具体功能的代码语句。函数体内的语句需要一致缩进。
- **定义函数 (Function Definition)** (课件P4-P5)： 使用 `def` 关键字来定义函数。基本语法：

```
def function_name(parameter1, parameter2, ...):
    """可选的函数文档字符串，用于解释函数功能。"""
    # 函数体 (执行的代码)
    # ...
    return # 可选的返回值
```

- `function_name`：函数的名称。
- `parameters`：函数的参数（形参），可以有零个或多个。
- `return` 语句：用于从函数中返回一个值。如果函数没有 `return` 语句或 `return` 后没有值，则默认返回 `None`。函数可以返回多个值，它们会以元组的形式打包返回 (课件P11)。

- **调用函数 (Function Call)** (课件P6-P8)： 定义函数后，通过函数名后跟括号 `()` 来调用它。如果函数定义了参数，调用时需要传入相应数量和类型的实际参数（实参）。

```
def double(x):
    print("我在一个名叫“double”函数里!") # 课件P5示例
    return 2 * x # 课件中为 2 + x，这里按常见乘法示例

result = double(2) # 调用函数
print(result)      # 输出: 我在一个名叫“double”函数里! \n 4

def greet():       # 无参数函数 (参考课件P9 g()函数)
    return "Hello!"
print(greet())     # 输出: Hello!
```

注意：调用时传入的参数数量必须与函数定义时的参数数量匹配，否则会产生 `TypeError` (课件P10)。

2. 参数传递 (Argument Passing) (课件P26-P29)

理解Python中参数如何传递对于正确使用函数至关重要。

- **传递的是对象的引用** (课件P26)： 在Python中，变量存储的是对象的引用（可以理解为内存地址）。当向函数传递参数时，实际上传递的是这些对象的引用。函数内部的形参会成为实参所指向对象的新的引用。
- **不可变类型 (Immutable Types) 作为参数** (课件P27-P28)： 如数字（int, float）、字符串 (str)、元组 (tuple)。 当不可变类型的对象作为参数传递给函数时，如果在函数内部对形参进行重新赋值（例如 `a = 10`），实际上是创建了一个新的对象，并将形参指向这个新对象。这**不会影响**到函数外部原始实参所指向的对象。

```python
# 课件P28 示例
def ChangeInt(a):
    a = 10 # a 在函数内部指向了新的int对象10
    print(f"函数内 a 的值: {a}")

b = 2
ChangeInt(b)
print(f"函数外 b 的值: {b}") # b 的值仍然是 2
# 输出:
# 函数内 a 的值: 10
# 函数外 b 的值: 2
```

- **可变类型 (Mutable Types) 作为参数** (课件P27, P29)： 如列表 (list)、字典 (dict)。 当可变类型的对象作为参数传递给函数时，函数内部的形参和函数外部的实参指向**同一个**对象。如果在函数内部通过形参修改了这个对象的内容（例如修改列表的元素），这些修改会**直接反映**到函数外部原始实参所指向的对象上。

```python
# 课件P29 示例
def changeme(mylist):
    mylist.append([1, 2, 3, 4]) # 直接修改了传入的列表对象
    print(f"函数内取值: {mylist}")
    return

original_list = [10, 20, 30]
changeme(original_list)
print(f"函数外取值: {original_list}")
# 输出:
# 函数内取值: [10, 20, 30, [1, 2, 3, 4]]
# 函数外取值: [10, 20, 30, [1, 2, 3, 4]]
```

易错点：如果函数内部将形参重新赋值给一个全新的可变对象 (例如 `mylist = [9,8,7]`)，那么形参的引用就改变了，后续对这个新对象的修改不会影响外部的原始对象。

3. 函数参数的类型 (课件P30-P37)

Python函数支持多种类型的参数：

- **必备参数/位置参数 (Required/Positional Arguments)** (课件P31-P32)： 调用函数时，必须按照函数定义中形参的顺序和数量传入实参。

```python
def printme(str_val): # 课件P31示例，略作修改以运行
    print(str_val)
    return

printme("Hello Python") # 必须传入一个参数
# printme() # 若不传参数，则会 TypeError (课件P32)
```

- **关键字参数 (Keyword Arguments)** (课件P33-P34)： 调用函数时，可以通过 `参数名=值` 的形式指定传入的实参。使用关键字参数时，实参的顺序可以与形参定义的顺序不一致。

```python
# 课件P33 示例
def printinfo(name, age):
    print(f"Name: {name}")
    print(f"Age: {age}")
    return

printinfo(age=50, name="miki")
```

- **默认参数值 (Default Argument Values)** (课件P35)： 在定义函数时，可以为某些形参指定默认值。如果在调用函数时没有为这些参数提供实参，则它们将使用定义时指定的默认值。

```python
# 课件P35 示例
def printinfo_default(name, age=35):
    print(f"Name: {name}")
    print(f"Age: {age}")
    return

printinfo_default(age=50, name="miki")
printinfo_default(name="miki") # age 将使用默认值 35
```

注意：具有默认值的参数必须放在没有默认值的参数之后。

- **不定长参数 (`\*args`)** (课件P36-P37)： 当不确定函数会接收多少个位置参数时，可以使用 `*args`。它允许函数接收任意数量的位置参数，这些参数在函数内部被收集到一个元组 (tuple) 中。

```python
# 课件P37 示例 (参数名改为常见的 args)
def print_flexible_info(arg1, *other_args):
    print("输出:")
    print(arg1)
    for var in other_args:
        print(var)
    return

print_flexible_info(10)
print_flexible_info(70, 60, 50)
```

(虽然课件未明确提及 `**kwargs`，但它是 `*args` 的对应概念，用于收集任意数量的关键字参数到字典中。)

4. 返回值 (`return` 语句) (课件P5, P11, P21-P24)

- 函数可以使用 `return` 语句将计算结果返回给调用者。
- 一旦执行了 `return` 语句，函数就会立即终止，并将指定的值返回 (课件P22)。

```python
def is_positive(x):
    print("Hello!") # 会运行
    return (x > 0)
    print("Goodbye!") # 不会运行，因为已被return终止

print(is_positive(5)) # 输出: Hello! \n True
```

- 如果函数没有 `return` 语句，或者 `return` 语句后面没有跟任何表达式，则函数默认隐式返回 `None`。
- **`print` vs `return` (课件P23-P24)**：这是一个常见的初学者易混淆点。
    - `print()`：是一个内置函数，用于将信息**显示**到控制台。它本身不“返回”一个可供程序后续使用的值（严格来说，`print()` 返回 `None`）。
    - `return`：是一个关键字，用于从函数中**输出**一个值。这个值可以被赋给变量，或用作其他表达式的一部分。

```python
# 课件P23 错误示例分析
def cubed_print(x):
    print(x**3) # 仅打印结果，不返回值

```

```
result_print = cubed_print(2) # 调用时会打印 8
print(f"cubed_print(2) 的结果是: {result_print}") # 输出: cubed_print(2) 的结果是: None

# 正确使用 return (课件P24)
def cubed_return(x):
    return x**3

result_return = cubed_return(3)
print(f"cubed_return(3) 的结果是: {result_return}") # 输出: cubed_return(3) 的结果是: 27
print(2 * cubed_return(4)) # 可以用于进一步计算, 输出: 128
```

5. 变量作用域 (重点 - 必考) (课件P16-P20) 🌟

变量作用域（Scope）指的是变量在程序中有效的、可被访问的区域。

- **局部变量 (Local Variables)** (课件P16-P17):
  - 在**函数内部定义**的变量（包括函数的参数）拥有**局部作用域**。
  - 它们**只能**在其定义的函数内部被访问和修改。函数外部无法直接访问函数内部的局部变量。
  - 当函数执行完毕后，这些局部变量通常会被销毁。
  - 不同函数中可以有同名的局部变量，它们是相互独立的 (课件P17 `f(x)` 和 `g(x)` 中的 `x`)。

```
# 课件P16 示例
def f_local(x_param):
    print(f"函数内 x_param: {x_param}")
    y_local = 5
    print(f"函数内 y_local: {y_local}")
    return x_param + y_local

f_local(4)
# print(x_param) # 会引发 NameError
# print(y_local) # 会引发 NameError
```

- **全局变量 (Global Variables)** (课件P19):
  - 在**所有函数外部**（通常在模块的顶层）定义的变量拥有**全局作用域**。
  - 全局变量可以在程序的**任何地方**被访问，包括所有函数内部。

```python
# 课件P19 示例 (略作修改)
global_var = 100

def func_access_global(x_local):
    return x_local + global_var # 函数内部可以直接读取全局变量 global_var

print(func_access_global(5))  # 输出: 105
print(func_access_global(6))  # 输出: 106
print(global_var)             # 输出: 100
```

- **在函数内部修改全局变量：`global` 关键字 (课件P20)**
    - 如果只是在函数内部**读取**全局变量的值，可以直接使用。
    - 但如果需要在函数内部**修改**全局变量的值，**必须**在该函数内部使用 `global` 关键字来声明该变量。这会告诉Python，你引用的是全局作用域中的那个变量。

```python
# 课件P20 示例
g_val = 100

def f_modify_global(x_local):
    global g_val # 声明 g_val 是全局变量
    g_val += 1   # 现在修改的是全局 g_val
    return x_local + g_val

print(f_modify_global(5)) # 第一次调用: g_val 变为 101, 返回 5 + 101 = 106
print(f_modify_global(6)) # 第二次调用: g_val 变为 102, 返回 6 + 102 = 108
print(g_val)              # 全局 g_val 已被修改为 102
# 输出:
# 106
# 108
# 102
```

易错点标记 (非常重要)： 如果你在函数内部尝试给一个与全局变量同名的变量赋值，但**没有**使用 `global` 关键字：

a. Python 会默认创建一个**新的局部变量**。

b. 如果在这个新的局部变量被赋值之前就尝试读取它（例如，在 `g_val = g_val + 1` 这样的操作中，右边的 `g_val`），Python会认为你正在引用一个尚未初始化的局部变量，从而引发 `UnboundLocalError`。

```
another_global = 50
def try_modify_global_incorrectly():
    # 错误尝试：没有 global 关键字
    # another_global = another_global + 10 # 如果取消注释这行，会发生 UnboundLocalError

    # 如果是直接赋值，则会创建一个同名的局部变量，不会影响全局变量
    another_global = 1000
    print(f"函数内 (局部) another_global: {another_global}")

try_modify_global_incorrectly()
print(f"函数外 (全局) another_global: {another_global}") # 全局变量的值未改变
# 输出:
# 函数内 (局部) another_global: 1000
# 函数外 (全局) another_global: 50
```

- 作用域查找规则 **(LEGB规则)**： Python解释器查找变量时遵循 LEGB 顺序：
    a. **L (Local)**：当前函数内的局部作用域。
    b. **E (Enclosing function locals)**：外层嵌套函数的局部作用域 (本次不考)。
    c. **G (Global)**：全局/模块作用域。
    d. **B (Built-in)**：Python内置的名称。

6. Lambda 函数 (匿名函数) (课件P39-P47)

- Lambda函数是一种小型的、匿名的（没有正式名称的）函数，通常只有一行。
- 它使用 `lambda` 关键字定义，主要用于需要一个简单函数作为参数的场合。
- 语法 (课件P39)：`lambda argument_list: expression`
    - `argument_list`：参数列表。
    - `expression`：一个单一的表达式，其计算结果作为函数的返回值。

```
# 课件P40 示例
square = lambda x: x * x
print(square(5)) # 输出: 25

# 课件P44 示例
add = lambda a, b: a + b
print(add(10, 20)) # 输出: 30
```

- **Lambda函数的用法** (课件P42, P45-P47)： 常作为高阶函数（如 `map()`, `filter()`, `sorted()`）的参数。

```
# 课件P42/P45 map() 示例
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x*x, numbers))
print(squared) # 输出: [1, 4, 9, 16, 25]

# 课件P42/P46 filter() 示例
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers) # 输出: [2, 4]

# 课件P47 sorted() 示例
names = ['Tiyong', 'Bob', 'Toy', 'Alice']
sorted_names = sorted(names, key=lambda x: len(x))
print(sorted_names) # 输出: ['Bob', 'Toy', 'Alice', 'Tiyong']
```

7. 其他相关概念

- **内置函数 (Built-in Functions)** (课件P14-P15)： Python提供了许多可以直接使用的内置函数，如 `print()`, `len()`, `type()`, `int()`, `float()`, `bool()`, `abs()`, `max()`, `min()`, `pow()`, `round()` 等。它们不需要导入任何模块。
- **辅助函数 (Helper Functions)** (课件P38)： 在编写较复杂的函数时，可以将一些重复的、独立的逻辑提取出来，定义成小的、辅助性的函数。这有助于主函数保持简洁，并提高代码的模块化和可读性。

```
# 课件P38 示例
def onesDigit(n):
    return n % 10

def largerOnesDigit(x, y):
    return max(onesDigit(x), onesDigit(y))

print(largerOnesDigit(134, 672)) # 输出: 4 (比较 4 和 2)
```

Python与大数据分析 --List, Tuple, Set,和Dict

1. 列表 (List) (课件P3-P36)

列表是Python中最常用的数据类型之一，它是一个有序且可变的序列，可以包含不同类型的元素。

- **创建列表 (Creating Lists)** (课件P4-P6)：
  - 使用方括号 `[]`，元素之间用逗号 `,` 分隔。

```
list1 = ['physics', 'chemistry', 1997, 2000] # 包含不同类型的元素 (课件P3)
list2 = [1, 2, 3, 4, 5]
empty_list = [] # 创建空列表 (课件P5)
another_empty_list = list() # 使用list()创建空列表 (课件P5)
```

  - 使用 `list()` 构造函数，可以将其他可迭代对象（如字符串、元组、range对象）转换为列表。

```
char_list = list("hello") # ['h', 'e', 'l', 'l', 'o']
range_list = list(range(5)) # [0, 1, 2, 3, 4] (课件P6)
```

  - 创建包含n个相同元素的列表：

```
zeros_list = [0] * 5 # [0, 0, 0, 0, 0] (课件P6)
```

- **访问列表元素 (Accessing Elements)** (课件P12)：
  - 通过索引访问，索引从0开始。支持正向索引和负向索引（-1表示最后一个元素）。

```
my_list = [10, 20, 30, 40, 50]
print(my_list[0])   # 输出: 10
print(my_list[-1])  # 输出: 50
```

  - 通过切片访问子列表 `list[start:end:step]`。

```
print(my_list[1:3]) # 输出: [20, 30] (不包含索引3的元素)
print(my_list[:2])  # 输出: [10, 20] (从头到索引2之前)
print(my_list[::2]) # 输出: [10, 30, 50] (步长为2)
```

- 修改列表 **(Modifying Lists)** (课件P13-P14)： 列表是可变的，可以直接修改其元素。

```
dna = ['G', 'A', 'T', 'T', 'A', 'C', 'A'] # 课件P13示例
print(f"原始DNA: {dna}, id: {id(dna)}")
dna[0] = 'C' # 修改第一个元素
print(f"修改后DNA: {dna}, id: {id(dna)}") # id不变，列表内容改变
# 输出:
# 原始DNA: ['G', 'A', 'T', 'T', 'A', 'C', 'A'], id: ...
# 修改后DNA: ['C', 'A', 'T', 'T', 'A', 'C', 'A'], id: ...
```

- 列表常用操作/方法：
    - `len(list)`: 返回列表长度 (课件P7)。
    - `min(list)`, `max(list)`, `sum(list)`: 返回列表的最小、最大、总和（元素需为数字）(课件P7)。
    - `item in list`: 判断元素是否存在于列表中 (课件P24)。
    - `list.append(item)`: 在列表末尾添加元素 (破坏性) (课件P14, P26)。
    - `list.insert(index, item)`: 在指定索引处插入元素 (破坏性) (课件P14, P26)。
    - `list.pop(index=-1)`: 移除并返回指定索引处的元素，默认为最后一个 (破坏性) (课件P14, P29)。
    - `list.remove(value)`: 移除列表中第一个出现的指定值的元素 (破坏性) (课件P14, P29)。
    - `list.extend(iterable)` 或 `list1 += list2` (对于列表 `+=` 是破坏性的): 将可迭代对象的元素追加到列表末尾 (破坏性) (课件P26)。
    - `list1 + list2`: 连接两个列表，返回一个新列表 (非破坏性) (课件P27)。
    - `list.count(item)`: 返回元素在列表中出现的次数 (课件P24)。
    - `list.index(item, start=0, end=len(list))`: 返回元素首次出现的索引，可指定查找范围 (课件P25)。
    - `list.sort(key=None, reverse=False)`: 原地对列表进行排序 (破坏性) (课件P36)。

    - `list.reverse()`: 原地反转列表中的元素 (破坏性) (课件P36)。
    - `sorted(iterable, key=None, reverse=False)`: 返回一个新的已排序的列表，不改变原列表 (非破坏性)。
- **列表的别名与复制 (Aliasing and Copying)** (课件P15-P20)：
    - **别名 (Alias)**：当将一个列表赋给另一个变量时（`b = a`），两个变量指向同一个列表对象。修改其中一个会影响另一个 (课件P16)。

```
a = [1, 2, 3]
b = a      # b 是 a 的别名
b[0] = 100
print(a)   # 输出: [100, 2, 3]
print(b)   # 输出: [100, 2, 3]
print(a is b) # 输出: True (它们是同一个对象)
```

    - **复制 (Copying)**：为了创建一个独立的列表副本，而不是别名，可以使用以下方法：
        - 切片：`c = a[:]` (课件P20)
        - `list()` 构造函数：`d = list(a)` (课件P20)
        - `copy` 模块的 `copy.copy()` (浅拷贝) 或 `copy.deepcopy()` (深拷贝，用于嵌套列表) (课件P19)。

```
import copy
a = [1, 2, 3]
c = a[:]
d = copy.copy(a)
c[0] = 200
d[0] = 300
print(a) # 输出: [1, 2, 3] (未被修改)
print(c) # 输出: [200, 2, 3]
print(d) # 输出: [300, 2, 3]
print(a is c) # 输出: False
```

    易错点：函数参数传递列表时，传递的是引用，因此函数内部对列表内容的修改会影响外部的原始列表（除非在函数内部创建了列表的副本或重新赋值给形参）。(课件P17)
- **列表推导式 (List Comprehensions)** (课件P41-P42)： 提供一种简洁的方式来创建列表。 语法：`[expression for item in iterable if condition]`

```
squares = [x**2 for x in range(1, 6)] # [1, 4, 9, 16, 25] (课件P42)
even_numbers = [x for x in range(10) if x % 2 == 0] # [0, 2, 4, 6, 8] (课件P42)
```

2. 元组 (Tuple) (课件P37-P40)

元组与列表类似，是有序序列，但元组是**不可变**的。一旦创建，元组的元素不能被修改、添加或删除。

- **创建元组 (Creating Tuples)** (课件P38)：
    - 使用圆括号 `()`，元素之间用逗号 `,` 分隔。

```
my_tuple = (1, 2, 3, 'a', 'b')
single_element_tuple = (5,) # 注意单个元素的元组末尾必须有逗号
empty_tuple = ()
another_empty_tuple = tuple()
```

    - 使用 `tuple()` 构造函数，可以将其他可迭代对象转换为元组。

```
list_to_tuple = tuple([1, 2, 3]) # (1, 2, 3)
```

- **访问元组元素 (Accessing Elements)**：与列表类似，通过索引和切片访问。

```
my_tuple = (10, 20, 30)
print(my_tuple[0])    # 输出: 10
print(my_tuple[1:]) # 输出: (20, 30)
```

- **元组的不可变性 (Immutability)** (课件P39)：

```
my_tuple = (1, 2, 3)
# my_tuple[0] = 100 # 这会引发 TypeError，因为元组不支持项赋值
```

  由于不可变，元组没有 `append()`, `remove()`, `sort()` 等修改自身的方法。

- **元组的用途**：
    - 当数据不应被修改时，使用元组更安全。
    - 元组可以作为字典的键（因为它们是不可变的），而列表不能。

- 函数返回多个值时，这些值通常以元组的形式返回。
- **并行赋值 (Parallel/Tuple Assignment)** (课件P40)：

```
(x, y) = (10, 20)
print(x) # 输出: 10
print(y) # 输出: 20

# 交换变量值
a = 5
b = 10
(a, b) = (b, a) # a变为10, b变为5
print(f"a: {a}, b: {b}")
```

3. 集合 (Set) (课件P43-P51)

集合是一个无序且不包含重复元素的集合。集合中的元素必须是不可变类型。

- **创建集合 (Creating Sets)** (课件P44)：
  - 使用花括号 `{}`，元素之间用逗号 `,` 分隔，或者使用 `set()` 构造函数。

```
my_set = {1, 2, 3, 2, 1} # 重复元素会被自动去除
print(my_set)           # 输出: {1, 2, 3} (顺序可能不同)

another_set = set([3, 4, 5, 4]) # 从列表创建集合
print(another_set)      # 输出: {3, 4, 5}

empty_set = set()       # 创建空集合必须用 set()
# wrong_empty_set = {} # 这会创建一个空字典，而不是空集合 (课件P44 警告)
```

- **集合的特性** (课件P46-P49)：
  - **无序性 (Unordered)** (课件P47)：集合中的元素没有特定的顺序，不能通过索引访问。
  - **唯一性 (Unique Elements)** (课件P48)：集合自动去除重复元素。
  - **元素必须不可变 (Elements Must Be Immutable)** (课件P49)：集合的元素不能是列表、字典或其他集合等可变类型。可以是数字、字符串、元组。

```
# s = {[1, 2], 3} # 会引发 TypeError: unhashable type: 'list'
s_ok = {(1, 2), 3} # 元组是不可变的，可以作为集合元素
```

- **集合常用操作/方法** (课件P45)：
    - `len(set)`: 返回集合中元素的数量。
    - `item in set`: 判断元素是否存在于集合中。
    - `set.add(item)`: 向集合中添加一个元素 (如果元素已存在则无效果)。
    - `set.remove(item)`: 从集合中移除一个元素，如果元素不存在则引发 `KeyError`。
    - `set.discard(item)`: 从集合中移除一个元素，如果元素不存在则什么也不做。
    - `set.pop()`: 随机移除并返回集合中的一个元素，如果集合为空则引发 `KeyError`。
    - `set.clear()`: 移除集合中的所有元素。
    - **集合运算**：
        - `set1 | set2` 或 `set1.union(set2)`: 并集。
        - `set1 & set2` 或 `set1.intersection(set2)`: 交集。
        - `set1 - set2` 或 `set1.difference(set2)`: 差集 (在set1中但不在set2中)。
        - `set1 ^ set2` 或 `set1.symmetric_difference(set2)`: 对称差集 (只在其中一个集合中出现的元素)。
        - `set1.issubset(set2)`: 判断set1是否为set2的子集。
        - `set1.issuperset(set2)`: 判断set1是否为set2的超集。
- **集合的效率** (课件P50-P51)： 集合对于成员资格检查（`in` 操作）、添加和删除元素等操作非常高效，通常具有 O(1) 的平均时间复杂度。

4. 字典 (Dict) (课件P52-P58)

字典是Python中另一种非常重要的数据结构，它存储键值对 (key-value pairs)。字典是无序的（在Python 3.7+版本中，字典保持插入顺序，但通常不应依赖此特性进行编程），键必须是唯一的且不可变的，值可以是任意类型。

- **创建字典 (Creating Dictionaries)** (课件P53)：
    - 使用花括号 `{}`，键值对之间用逗号 `,` 分隔，键和值之间用冒号 `:` 分隔。

```
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}
empty_dict = {} # 创建空字典
another_empty_dict = dict() # 使用dict()创建空字典
```

- 使用 `dict()` 构造函数，可以从键值对序列创建字典。

```
pairs = [('cow', 5), ('dog', 98), ('cat', 1)] # 课件P53示例
dict_from_pairs = dict(pairs)
print(dict_from_pairs) # 输出: {'cow': 5, 'dog': 98, 'cat': 1} (顺序可能不同)
```

- 访问字典元素 **(Accessing Elements)** (课件P54)：
  - 通过键来访问对应的值：`my_dict[key]`。如果键不存在，会引发 `KeyError`。
  - 使用 `get()` 方法：`my_dict.get(key, default_value)`。如果键不存在，返回 `default_value`（默认为 `None`），而不会报错。

```
student = {'name': 'Bob', 'score': 85}
print(student['name'])         # 输出: Bob
print(student.get('grade'))    # 输出: None (因为 'grade' 键不存在)
print(student.get('grade', 'N/A')) # 输出: N/A
# print(student['grade'])      # 会引发 KeyError
```

- 修改和添加字典元素 (课件P54)：
  - 通过键赋值来添加新的键值对或修改已存在的键的值。

```
student['score'] = 90        # 修改已存在的键的值
student['subject'] = 'Math' # 添加新的键值对
print(student)               # 输出: {'name': 'Bob', 'score': 90, 'subject': 'Math'}
```

- 删除字典元素 (课件P54)：
  - `del my_dict[key]`: 删除指定键的键值对。如果键不存在，引发 `KeyError`。
  - `my_dict.pop(key, default_value)`: 移除并返回指定键的值。如果键不存在，返回 `default_value`（如果提供），否则引发 `KeyError`。

    - `my_dict.popitem()`: 移除并返回字典中最后一个插入的键值对（Python 3.7+）或一个任意键值对（Python 3.6及更早版本）。如果字典为空，引发 `KeyError`。
    - `my_dict.clear()`: 清空字典中的所有键值对。
- **字典的特性** (课件P55-P58)：
    - **键是唯一的 (Keys are Unique)** (课件P56)：如果尝试用相同的键添加新的值，旧的值会被覆盖。
    - **键必须是不可变的 (Keys Must Be Immutable)** (课件P56, P58)：键通常是字符串、数字或元组（如果元组只包含不可变元素）。列表、字典等可变类型不能作为键。

```
# d = {[1, 2]: 'value'} # 会引发 TypeError: unhashable type: 'list'
d_ok = {(1, 2): 'value'} # 元组可以作为键
```

    - **值可以是任意类型 (Values Can Be Anything)** (课件P57)：字典的值可以是任何Python对象，包括数字、字符串、列表、字典等。
    - **无序性（传统上）/插入顺序（现代Python）** (课件P55)：在Python 3.7及更高版本中，字典会记住元素的插入顺序。但在之前的版本中，字典是无序的。编程时不应过于依赖字典的顺序，除非明确知道Python版本。
- **字典常用操作/方法** (课件P54)：
    - `len(dict)`: 返回字典中键值对的数量。
    - `key in dict`: 判断键是否存在于字典中。
    - `dict.keys()`: 返回一个包含所有键的视图对象。
    - `dict.values()`: 返回一个包含所有值的视图对象。
    - `dict.items()`: 返回一个包含所有 (键, 值) 元组的视图对象。
    - **遍历字典**：

```python
my_dict = {'a': 1, 'b': 2, 'c': 3}
# 遍历键 (课件P54)
for key in my_dict:
    print(key, my_dict[key])

# 遍历值
for value in my_dict.values():
    print(value)

# 遍历键值对
for key, value in my_dict.items():
    print(f"Key: {key}, Value: {value}")
```

5. 破坏性 (Destructive) 与非破坏性 (Non-Destructive) 操作比较

这个概念主要针对可变数据类型，尤其是列表。

- 破坏性操作 **(Destructive Operations / In-place Operations)** (参考课件P14, P21-P23, P26, P29, P36)：
    - 直接修改原始数据对象本身。
    - 函数或方法通常不返回新的对象（或者返回 `None`，或者返回被修改的对象自身）。
    - **优点**：如果不需要保留原始数据，可以节省内存，因为不需要创建新的数据副本。
    - **缺点**：如果后续代码还需要使用原始数据，或者有多个变量引用同一个对象（别名），可能会导致意外的副作用。
    - **列表的破坏性方法示例**：
        - `list.append(item)`
        - `list.extend(iterable)`
        - `list.insert(index, item)`
        - `list.remove(value)`
        - `list.pop(index)`
        - `list.sort()`
        - `list.reverse()`
        - 通过索引直接赋值修改元素：`my_list[0] = new_value`

```python
# 破坏性示例
my_list = [1, 2, 3]
print(f"原始列表: {my_list}, id: {id(my_list)}")
my_list.append(4) # append 是破坏性的
print(f"修改后列表: {my_list}, id: {id(my_list)}") # id 相同，内容改变
```

- 非破坏性操作 **(Non-Destructive Operations / Operations that Return a New Object)** (参考课件P20, P23, P27)：
    - 不修改原始数据对象，而是创建一个新的数据对象作为操作结果返回。
    - 原始数据保持不变。
    - **优点**：更安全，因为原始数据不会被意外修改，便于追踪数据状态。
    - **缺点**：如果频繁进行且数据量大，可能会消耗更多内存和时间来创建新对象。
    - **列表的非破坏性操作示例**：
        - 列表连接：`new_list = list1 + list2`
        - 列表切片：`sub_list = my_list[1:3]` (切片本身返回新列表)
        - `sorted(iterable)` 函数 (返回新的已排序列表)
        - 列表推导式 (总是创建新列表)
        - 使用 `copy.copy()` 或 `copy.deepcopy()` 创建副本后的操作。

```python
# 非破坏性示例
my_list = [1, 2, 3]
print(f"原始列表: {my_list}, id: {id(my_list)}")
new_list = my_list + [4, 5] # + 操作是非破坏性的
print(f"原始列表仍然是: {my_list}, id: {id(my_list)}")
print(f"新列表: {new_list}, id: {id(new_list)}") # new_list 是一个新对象
```

易错点 (课件P28)： `a = a + [4]` 看起来像是修改了 `a`，但实际上是先执行 `a + [4]` 创建了一个新列表，然后将变量 `a` 的引用指向了这个新列表。如果之前有其他变量是 `a` 的别名，那么那些别名仍然指向旧的列表。 而 `a += [4]` (对于列表，等价于 `a.extend([4])`) 是破坏性的，会直接修改 `a` 指向的那个列表对象。

Python与大数据分析 --模块和包的简单运用

1. 模块 (Module) (课件P43)

- 什么是模块？
    - 在Python中，一个 `.py` 文件就可以被称为一个模块。
    - 模块是Python组织代码的基本方式，它允许你将相关的代码（如函数、类、变量定义）分割成独立的、可重用的单元。
    - 使用模块可以避免命名冲突，并使得代码结构更清晰。
- **创建模块 (Creating Modules)** (课件P44-P45)： 创建模块非常简单：

    a. 创建一个新的 `.py` 文件（例如，`my_module.py`）。

    b. 在该文件中编写Python代码（例如，定义函数、类或变量）。

    c. 保存文件。

    示例 (参考课件P45)： 创建一个名为 `my_module.py` 的文件，内容如下：

```python
# my_module.py
def say_hello(name):
    print(f"Hello, {name} from my_module!")

my_variable = "This is a module variable."
```

    这个 `my_module.py` 文件就是一个模块。

- **导入模块 (Importing Modules)** (课件P46)： 要使用一个模块中定义的函数、类或变量，你需要先将其导入到当前的代码文件中。Python提供了多种导入模块的方式：

    a. `import module_name`： 导入整个模块。使用时需要通过 `module_name.member_name` 的方式来访问模块中的成员。

```python
import math
print(math.pi)       # 输出圆周率
print(math.sqrt(16)) # 输出 4.0 (课件P46示例)

import my_module
my_module.say_hello("Python User") # 调用 my_module 中的函数
print(my_module.my_variable)
```

b. `from module_name import member_name1, member_name2`：从模块中导入指定的成员（函数、类、变量）到当前命名空间。可以直接使用成员名，无需通过模块名。

```
from math import sqrt, pi
print(pi)           # 直接使用 pi
print(sqrt(16))     # 直接使用 sqrt (课件P46示例)

from my_module import say_hello
say_hello("Developer") # 直接调用
```

c. `from module_name import \*`：导入模块中的所有公共成员（不以 `_` 开头的名称）到当前命名空间。**通常不推荐**这种方式，因为它可能导致命名冲突，降低代码可读性。

```
# 不推荐使用
# from math import *
# print(sqrt(25))
```

d. `import module_name as alias_name`：导入模块并为其指定一个别名，以简化模块名的使用，特别是在模块名较长或容易冲突时。

```
import math as m
print(m.sqrt(16)) # 使用别名 m (课件P46示例)

import my_module as mm
mm.say_hello("Student")
```

e. `from module_name import member_name as alias_name`：从模块导入指定成员，并为该成员指定一个别名。

```
from math import sqrt as square_root
print(square_root(16)) # 使用别名 square_root (课件P46示例)

from my_module import say_hello as greet
greet("World")
```

2. 包 (Package) (课件P47)

- 什么是包？
    - 当模块数量很多，或者需要按功能组织模块时，可以使用包。
    - 包是一种组织Python模块的方式，它将相关的模块组织在一个目录层级结构中。
    - 从文件系统的角度看，一个包就是一个包含了一个特殊文件 `__init__.py` 的文件夹。这个文件夹下可以包含其他模块（`.py` 文件）或子包（同样包含 `__init__.py` 的子文件夹）。
    - `__init__.py` 文件是包的标识，它可以是空文件，也可以包含包的初始化代码或定义 `__all__` 变量来控制 `from package import *` 时导入的内容。
- 包的结构示例 (参考课件P47)： 假设有如下的目录结构：

```
my_project/
    main_program.py
    my_package/               # 这是一个包
        __init__.py           # 标记 my_package 是一个包
        module1.py
        module2.py
        sub_package/          # 这是一个子包
            __init__.py       # 标记 sub_package 是一个包
            sub_module1.py
```

    - `my_package` 是一个包。
    - `module1.py` 和 `module2.py` 是 `my_package` 包中的模块。
    - `sub_package` 是 `my_package` 包中的一个子包。
    - `sub_module1.py` 是 `sub_package` 子包中的模块。

3. 导入包中的模块 (课件P48-P50) (必考点) 🌟

导入包中的模块与导入普通模块类似，但需要使用点号 `.` 来表示包的层级关系。主要有两种导入方式：绝对导入和相对导入。

- **绝对导入 (Absolute Imports)** (课件P50)： 绝对导入总是从项目的根路径（或者说，Python解释器可以找到的顶层包）开始指定模块的完整路径。 假设 `my_project` 目录在Python的搜索路径中（例如，当前工作目录是 `my_project`，或者 `my_project` 的父目录在 `PYTHONPATH` 中）。

  在 `main_program.py` (与 `my_package` 同级) 中导入：

  ```
  # 在 main_program.py 中
  import my_package.module1
  my_package.module1.some_function_in_module1()

  from my_package import module2
  module2.another_function_in_module2()

  from my_package.sub_package import sub_module1
  sub_module1.function_in_sub_module1()

  from my_package.sub_package.sub_module1 import specific_function
  specific_function()
  ```

  课件P50示例：如果在 `sub_package/sub_module1.py` 中想导入 `my_package/module1.py`，可以使用绝对导入：

  ```
  # 在 my_package/sub_package/sub_module1.py 中
  from my_package import module1 # 假设 my_package 是可识别的顶层包
  # module1.say_hello("...") # 假设 module1 有 say_hello 函数
  ```

  绝对导入路径清晰，易于理解，是推荐的导入方式。

- **相对导入 (Relative Imports)** (课件P49)： 相对导入是**相对于当前模块的位置**来指定要导入的模块的路径。它使用点号 `.` 来表示相对位置：
  - 一个点 `.` 表示当前包（模块所在的包）。
  - 两个点 `..` 表示当前包的父包。
  - 三个点 `...` 表示父包的父包，以此类推。

  相对导入只能用于包内的模块之间，不能用于顶层脚本（即直接被执行的 `.py` 文件，它没有包的上下文）。

  课件P49示例分析与校正/详细解释： 目录结构如下：

```
my_package/
    __init__.py
    module1.py
    module2.py
    sub_package/
        __init__.py
        sub_module1.py
```

课件P49中提到：“如果在 `sub_package/sub_module1.py` 中想导入 `module1.py`，可以使用相对导入: `from .. import module1`”。 这个示例是正确的。 解释如下：

- 当前文件是 `sub_module1.py`，它位于 `sub_package` 这个包内。
- `..` 对于 `sub_module1.py` 来说，指的是 `sub_package` 的父包，即 `my_package`。
- 因此，`from .. import module1` 的含义是从 `my_package` 包中导入 `module1` 模块。这是正确的，因为 `module1.py` 确实直接位于 `my_package` 目录下。

更多相对导入示例：

a. 在 `sub_package/sub_module1.py` 中导入同级（`sub_package`内）的另一个模块 `sub_module2.py` (假设存在)：

```
# 在 my_package/sub_package/sub_module1.py 中
from . import sub_module2 # . 代表当前包 sub_package
# 或者 from .sub_module2 import some_function
```

b. 在 `my_package/module1.py` 中导入同级（`my_package`内）的 `module2.py`：

```
# 在 my_package/module1.py 中
from . import module2 # . 代表当前包 my_package
```

c. 在 `my_package/module1.py` 中导入子包 `sub_package` 中的 `sub_module1.py`：

```
# 在 my_package/module1.py 中
from .sub_package import sub_module1 # . 代表当前包 my_package
```

易错点与注意事项：

- 相对导入的写法（如 `from . import name` 或 `from ..name import subname`）只能在包内的模块中使用。如果你尝试在一个不被Python视为包一部分的顶层脚本中使用相对导入，会收到 `ImportError: attempted relative import with no known parent package` 的错误。
- 选择绝对导入还是相对导入：
    - **绝对导入**更明确，易于理解模块的来源，通常是首选，尤其是在大型项目中。
    - **相对导入**在包内部引用时可以更简洁，并且如果包的顶层名称可能改变（例如，包被重命名或移动），相对导入可以保持包内部引用的稳定性。然而，过度使用深层嵌套的相对导入（如 `from .... import something`）可能会降低代码的可读性。

4. `__init__.py` 文件的作用

- **标记目录为包**：如前所述，`__init__.py` 文件的存在告诉Python解释器该目录应该被视为一个包。
- **包初始化代码**：可以在 `__init__.py` 文件中编写包级别的初始化代码。当包或包中的任何模块被导入时，`__init__.py` 文件会被首先执行。
- **简化导入**：可以在 `__init__.py` 中使用 `from .module import name` 的形式，将子模块中的特定名称提升到包的命名空间下，使得用户可以直接从包导入这些名称，而无需指定子模块。 例如，在 `my_package/__init__.py` 中写入：

```
# my_package/__init__.py
from .module1 import say_hello
print("my_package is being initialized!")
```

那么其他地方就可以这样导入：

```
from my_package import say_hello # 而不是 from my_package.module1 import say_hello
say_hello("User")
# 同时会打印 "my_package is being initialized!"
```

- 控制 `from package import \*` 的行为：可以在 `__init__.py` 中定义一个名为 `__all__` 的列表，该列表包含了当执行 `from package import *` 时应该被导入的模块名或子包名（字符串形式）。

```
# my_package/__init__.py
__all__ = ["module1", "module2"]
```

##

Python与大数据分析 --NumPy介绍、安装及其操作

1. NumPy 数组 (ndarray)

NumPy 最核心的数据结构是 `ndarray` 对象。

- 数组的属性 (课件P6, P8, P11-P12)：
  - `ndim`: 数组的维数（轴的个数）。例如，一维数组的 `ndim` 为1，二维数组为2。
  - `shape`: 数组的维度。返回一个元组，表示数组在每个维度上的大小。例如，一个2行3列的二维数组，其 `shape` 为 `(2, 3)`。
  - `size`: 数组中元素的总个数。等于 `shape` 属性中各元素之积。
  - `dtype`: 数组中元素的数据类型。NumPy 支持多种数据类型 (课件P7-P9)，如 `int32`, `float64`, `bool_`, `string_` 等。
  - `itemsize`: 数组中每个元素占用的字节大小。
  - `data`: 包含实际数组元素的缓冲区。

```
import numpy as np

arr1d = np.array([1, 2, 3, 4])
print(f"一维数组: {arr1d}")
print(f"维数 (ndim): {arr1d.ndim}")       # 输出: 1
print(f"形状 (shape): {arr1d.shape}")     # 输出: (4,)
print(f"总元素数 (size): {arr1d.size}")   # 输出: 4
print(f"数据类型 (dtype): {arr1d.dtype}") # 输出: int32 (或根据系统)

arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(f"\n二维数组:\n{arr2d}")
print(f"维数 (ndim): {arr2d.ndim}")       # 输出: 2
print(f"形状 (shape): {arr2d.shape}")     # 输出: (2, 3)
```

```
print(f"总元素数 (size): {arr2d.size}")   # 输出: 6
print(f"数据类型 (dtype): {arr2d.dtype}") # 输出: int32 (或根据系统)
```

- **数据类型 (dtype) 及转换** (课件P7-P9)：
  - NumPy 支持多种数值类型，如 `np.int8`, `np.int16`, `np.int32`, `np.int64`, `np.float32`, `np.float64`, `np.bool_`, `np.string_` 等。
  - 创建数组时可以指定 `dtype`，否则 NumPy 会尝试推断合适的数据类型。
  - 可以使用 `astype()` 方法显式转换数组的数据类型。`astype()` **总是创建一个新数组（副本）**，即使 `dtype` 与原数组相同。

```
arr_float = np.array([1.1, 2.2, 3.3])
print(f"浮点数组: {arr_float}, dtype: {arr_float.dtype}")

arr_int = arr_float.astype(np.int32) # 转换为整数，小数部分被截断
print(f"转换为整数后: {arr_int}, dtype: {arr_int.dtype}") # 输出: [1 2 3], dtype: int32
print(f"arr_int is arr_float: {arr_int is arr_float}") # False, astype创建了新数组

numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
arr_from_str = numeric_strings.astype(float) # 字符串转浮点数
print(f"字符串转浮点数: {arr_from_str}, dtype: {arr_from_str.dtype}") # 输出: [  1.25  -9.6   42.  ], dtype: float64
```

2. 创建 NumPy 数组 (课件P10-P14)

- **从 Python 列表或元组创建** (课件P10-P12)： 使用 `np.array()` 函数。

```
list_data = [1, 2, 3, 4, 5]
arr_from_list = np.array(list_data)
print(f"从列表创建: {arr_from_list}")

tuple_data = ((1, 2, 3), (4, 5, 6))
arr_from_tuple = np.array(tuple_data) # 创建二维数组
print(f"从元组创建二维数组:\n{arr_from_tuple}")
```

- 使用 `arange()` 创建等差序列数组 (课件P13)： 类似于 Python 内置的 `range()` 函数，但返回的是 NumPy 数组。 `np.arange(start, stop, step, dtype=None)`

```
arr_range1 = np.arange(5)          # [0 1 2 3 4]
arr_range2 = np.arange(1, 10, 2)  # [1 3 5 7 9] (课件P13示例)
print(f"arange(5): {arr_range1}")
print(f"arange(1, 10, 2): {arr_range2}")
```

- 使用 `linspace()` 创建等间隔序列数组： `np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)` 在指定的间隔内返回 `num` 个均匀间隔的样本。 `endpoint` 决定是否包含 `stop` 值。

```
arr_linspace = np.linspace(0, 10, 5) # 在0到10之间生成5个等间隔数
print(f"linspace(0, 10, 5): {arr_linspace}") # 输出: [ 0.   2.5  5.   7.5 10. ]
```

- 创建特殊数组 (课件P14)：
    - `np.zeros(shape, dtype=float)`: 创建指定形状的全零数组。
    - `np.ones(shape, dtype=float)`: 创建指定形状的全一数组。
    - `np.empty(shape, dtype=float)`: 创建指定形状的未初始化数组（其值是任意的）。
    - `np.eye(N, M=None, k=0, dtype=float)`: 创建一个 N x M 的单位矩阵（对角线为1，其余为0）。如果 M 为 None，则默认为 N x N。k 控制对角线的位置。
    - `np.full(shape, fill_value, dtype=None)`: 创建指定形状并以 `fill_value` 填充的数组。
    - `np.random` 模块：用于创建各种随机数数组，例如：
        - `np.random.rand(d0, d1, ..., dn)`: 创建[0, 1)之间均匀分布的随机数数组。
        - `np.random.randn(d0, d1, ..., dn)`: 创建标准正态分布（均值为0，方差为1）的随机数数组。
        - `np.random.randint(low, high=None, size=None, dtype=int)`: 创建指定范围内的随机整数数组。

```

```

```
arr_zeros = np.zeros((2, 3))
print(f"全零数组:\n{arr_zeros}")

arr_ones = np.ones(3)
print(f"全一数组: {arr_ones}")

arr_eye = np.eye(3) # 3x3 单位矩阵
print(f"单位矩阵:\n{arr_eye}")

arr_full = np.full((2, 2), 7)
print(f"全7数组:\n{arr_full}")

arr_rand = np.random.rand(2, 2)
print(f"随机数组 (0-1均匀分布):\n{arr_rand}")
```

3. 数组的维度操作 (课件P15-P18)

- `reshape()`: 改变数组的形状，但不改变其数据。新形状的元素总数必须与原形状一致。**通常返回原数组的视图 (view)**，如果可能的话；否则返回副本。如果希望确保是副本，可以使用 `arr.reshape(...).copy()`。(课件P15-P16)

```
arr = np.arange(1, 7) # [1 2 3 4 5 6]
reshaped_arr_view = arr.reshape((2, 3))
print(f"原始数组: {arr}")
print(f"reshape((2,3)) 后 (通常是视
图):\n{reshaped_arr_view}")
reshaped_arr_view[0,0] = 99 # 修改视图
print(f"修改视图后，原数组 arr: {arr}") # 原数组也被修改

# 如果某个维度设为 -1，NumPy会自动计算该维度的大小
reshaped_auto = arr.reshape((3, -1)) # 自动计算为 (3, 2)
print(f"reshape((3,-1)) 后:\n{reshaped_auto}")
```

- `resize()`:
    - `ndarray.resize(new_shape, refcheck=True)`: **直接修改原数组 (破坏性操作)**。如果新大小与原大小不同，可能会报错（如果其他变量引用该数组且`refcheck=True`）。如果新大小大于原大小，则用0填充；如果小于，则舍弃数据。(课件P17)

- `np.resize(a, new_shape)`: **返回新数组 (非破坏性操作)**，不改变原数组。如果新大小大于原大小，则循环使用原数组数据填充；如果小于，则舍弃数据。(课件P18)

```
arr_to_resize_inplace = np.array([[1,2],[3,4],[5,6]])
print(f"resize前: shape={arr_to_resize_inplace.shape}\n{arr_to_resize_inplace}")
arr_to_resize_inplace.resize((2,3)) # 直接修改原数组 (破坏性)
print(f"arr.resize((2,3))后: shape={arr_to_resize_inplace.shape}\n{arr_to_resize_inplace}")

arr_orig = np.array([1,2,3])
resized_np_new_array = np.resize(arr_orig, (2,3)) # 返回新数组 (非破坏性)
print(f"原数组 arr_orig: {arr_orig}")
print(f"np.resize(arr_orig, (2,3))后 (新数组):\n{resized_np_new_array}")
```

- `flatten()` 与 `ravel()`: 将多维数组转换为一维数组。
  - `flatten()`: **总是返回一个数组的副本 (非破坏性)**。
  - `ravel()`: **通常返回原始数组的视图 (view)**（如果可能），修改视图会影响原数组。如果无法返回视图，则返回副本。

4. 数组索引和切片 (课件P19-P31, P37)

NumPy 数组的索引和切片功能非常强大。

- **一维数组索引与切片** (课件P19-P23)： 与 Python 列表类似。

```
arr = np.arange(0, 90, 10) # [ 0 10 20 30 40 50 60 70 80] (课件P19)
print(f"arr[5]: {arr[5]}")              # 输出: 50
print(f"arr[5:8]: {arr[5:8]}")          # 输出: [50 60 70]
print(f"arr[[1,2,-3]]: {arr[[1,2,-3]]}") # 花式索引，输出: [10 20 60]

# 使用 slice 对象 (课件P20)
s = slice(2, 9, 3) # 从索引2到9（不含），步长3
print(f"arr[slice(2,9,3)]: {arr[s]}") # 输出: [20 50 80]

# 切片是视图 (View)，不是副本 (课件P21)
# 对视图的修改会影响原数组 (破坏性影响原数组)
```

```
a_slice_view = arr[5:8]
a_slice_view[1] = 12345 # 修改切片（视图）
print(f"修改切片后，原数组 arr: {arr}") # 原数组也被修改

# 若要副本，使用 .copy() (课件P22) (非破坏性)
arr_copy_slice = arr[5:8].copy()
arr_copy_slice[:] = 99 # 修改副本
print(f"修改副本切片后，原数组 arr: {arr}") # 原数组不受影响
```

- **多维数组索引与切片** (课件P24-P31)：
  - 可以通过逗号分隔的索引元组访问元素，如 `arr2d[row_index, col_index]`。
  - 可以对每个维度进行切片。
  - 如果省略了后面的索引，返回的是一个维度更低的子数组（视图）。
  - 对切片（视图）的赋值会直接修改原数组的数据 (破坏性影响原数组)。

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) (课件P24, P28)
print(f"arr2d[1, 0]: {arr2d[1, 0]}") # 第2行第1列元素，输出: 4 (课件P27)

# 切片
view1 = arr2d[:2, 1:] # 前两行，从第二列开始 (课件P28)
view1[0,0] = 99 # 修改视图
print(f"修改视图后，原数组 arr2d:\n{arr2d}")
# [[ 1 99  3]
#  [ 4  5  6]
#  [ 7  8  9]]
```

- **布尔型索引 (Boolean Indexing)** (课件P32-P36)： 可以使用布尔数组来选择数组中的元素。布尔数组的长度必须与被索引的轴长度一致。 布尔索引**总是创建一个副本 (非破坏性)**。

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
data = np.arange(28).reshape(7, 4) # 示例数据

is_bob = (names == 'Bob')
selected_data = data[is_bob] # selected_data 是一个副本
print(f"data[is_bob]:\n{selected_data}")
selected_data[0,0] = 1000 # 修改副本
print(f"修改副本后，原data中Bob的第一行首元素: {data[0,0]}") # 原data不变

# 也可以直接用在赋值操作中，这时是修改原数组中满足条件的部分 (破坏性)
data_copy_for_bool_assign = data.copy()
data_copy_for_bool_assign[data_copy_for_bool_assign < 10] = -1 # 将小于10的元素设为-1
print(f"布尔赋值修改后:\n{data_copy_for_bool_assign}")
```

- **花式索引 (Fancy Indexing)**：使用整数数组（或列表）进行索引。花式索引**总是创建一个副本 (非破坏性)**。

```
arr_fancy = np.arange(10, 20)
indices = [3, 1, 5]
selected_fancy = arr_fancy[indices] # selected_fancy 是一个副本
print(f"花式索引结果: {selected_fancy}")
selected_fancy[0] = 999 # 修改副本
print(f"修改副本后，原arr_fancy[3]: {arr_fancy[3]}") # 原arr_fancy不变
```

- `np.where(condition, [x, y])` (课件P37)：`np.where` 本身是**非破坏性**的，它返回一个新的数组或索引。 如果只提供 `condition`，返回索引，不改变原数组。如果提供 `condition, x, y`，则根据条件从 `x` 或 `y` (或标量) 中选取元素构建一个新数组。

```
arr_where = np.arange(0, 100, 10)
indices = np.where(arr_where < 50) # 返回索引，非破坏性
print(f"arr_where < 50 的索引: {indices}")

result_where = np.where(arr_where > 30, arr_where * 2, arr_where / 2) # 构建新数组，非破坏性
print(f"np.where 条件赋值结果 (新数组): {result_where}")
print(f"原数组 arr_where: {arr_where}") # 原数组不变
```

5. 数组运算 (课件P39-P41, P54)

NumPy 的数组运算（如加减乘除、比较等）通常是**非破坏性**的，它们会返回一个新的数组作为运算结果，而不会修改原始输入数组。

- **数组与标量的运算** (课件P41)：返回新数组。
- **数组间的算术运算** (课件P39-P40)：返回新数组。
- **广播 (Broadcasting)**：在运算过程中，如果发生广播，NumPy 也是在概念上扩展数组，实际操作结果会存放在新数组中。

```
a = np.array([1, 2, 3])
b_orig = a # b_orig 是 a 的别名
c_new = a + 5 # 非破坏性，c_new 是新数组
print(f"a: {a}, id(a): {id(a)}")
print(f"c_new: {c_new}, id(c_new): {id(c_new)}")
print(f"b_orig: {b_orig}, id(b_orig): {id(b_orig)}") # a 和 b_orig 仍然是同一个

# 如果想原地修改，可以使用增量赋值操作符，如 +=, -=, *=, /=
# 这些对于NumPy数组通常是破坏性的
a_inplace = np.array([1.0, 2.0, 3.0])
b_alias_inplace = a_inplace
print(f"修改前 a_inplace: {a_inplace}, id: {id(a_inplace)}")
a_inplace += 5 # 破坏性操作
print(f"修改后 a_inplace: {a_inplace}, id: {id(a_inplace)}") # id 可能不变，内容改变
print(f"别名 b_alias_inplace: {b_alias_inplace}") # b_alias_inplace 也改变了
```

6. 通用函数 (Universal Functions - ufuncs) (课件P42-P43)

一元 **ufuncs** (对单个数组操作，课件P42-P43 表4-3)：

- `np.abs(arr)` 或 `np.fabs(arr)`: 计算绝对值。
- `np.sqrt(arr)`: 计算平方根。
- `np.square(arr)`: 计算平方。
- `np.exp(arr)`: 计算指数 ex。
- `np.log(arr)`, `np.log10(arr)`, `np.log2(arr)`: 计算自然对数、底为10的对数、底为2的对数。

- `np.sign(arr)`: 计算各元素的正负号 (1, 0, -1)。
- `np.ceil(arr)`, `np.floor(arr)`: 向上取整、向下取整。
- `np.rint(arr)`: 四舍五入到最近整数。
- `np.isnan(arr)`: 判断是否为 NaN (Not a Number)。
- `np.isfinite(arr)`, `np.isinf(arr)`: 判断是否有穷或无穷。
- 三角函数: `np.sin(arr)`, `np.cos(arr)`, `np.tan(arr)` 及其反函数和双曲函数。
- `np.logical_not(arr)`: 计算各元素的逻辑非。

二元 **ufuncs** (对两个数组操作，课件P43 表4-4):

- `np.add(arr1, arr2)` (等同于 `arr1 + arr2`)
- `np.subtract(arr1, arr2)` (等同于 `arr1 - arr2`)
- `np.multiply(arr1, arr2)` (等同于 `arr1 * arr2`)
- `np.divide(arr1, arr2)` (等同于 `arr1 / arr2`)
- `np.floor_divide(arr1, arr2)` (等同于 `arr1 // arr2`)
- `np.power(arr1, arr2)` (等同于 `arr1 ** arr2`)
- `np.maximum(arr1, arr2)`, `np.minimum(arr1, arr2)`: 元素级的最大/最小值。
- `np.mod(arr1, arr2)` (等同于 `arr1 % arr2`)
- 比较函数: `np.greater(arr1, arr2)`, `np.less(arr1, arr2)`, `np.equal(arr1, arr2)` 等。
- 逻辑函数: `np.logical_and(arr1, arr2)`, `np.logical_or(arr1, arr2)`, `np.logical_xor(arr1, arr2)`。

ufuncs 通常是**非破坏性**的，它们对输入数组的每个元素执行操作，并返回一个包含结果的新数组。

```
arr_ufunc = np.array([1, 4, 9])
sqrt_arr = np.sqrt(arr_ufunc) # 非破坏性
print(f"原数组 arr_ufunc: {arr_ufunc}")
print(f"平方根结果 (新数组) sqrt_arr: {sqrt_arr}")
```

一些ufuncs有 `out` 参数，可以指定将结果存储在已存在的数组中，这种情况下可以实现原地操作的效果，但需要用户显式提供输出数组。

7. 聚合函数 (Statistical Methods) (课件P54 `axis` 参数)

NumPy 提供了一些常用的聚合函数，可以对整个数组或沿某个轴计算统计值。

- `arr.sum()` 或 `np.sum(arr)`: 计算所有元素的和。
- `arr.mean()` 或 `np.mean(arr)`: 计算平均值。
- `arr.std()` 或 `np.std(arr)`: 计算标准差。
- `arr.var()` 或 `np.var(arr)`: 计算方差。
- `arr.min()` 或 `np.min(arr)`: 找到最小值。
- `arr.max()` 或 `np.max(arr)`: 找到最大值。
- `arr.argmin()` 或 `np.argmin(arr)`: 找到最小值的索引。
- `arr.argmax()` 或 `np.argmax(arr)`: 找到最大值的索引。
- `arr.cumsum()` 或 `np.cumsum(arr)`: 计算累积和。
- `arr.cumprod()` 或 `np.cumprod(arr)`: 计算累积积。

这些函数都可以接受一个 `axis` 参数，用于指定沿哪个轴进行计算 (课件P54图示)：

- `axis=0`: 沿列操作 (对每一列进行计算)。
- `axis=1`: 沿行操作 (对每一行进行计算)。
- 如果不指定 `axis`，则对整个数组进行计算。

聚合函数如 `sum()`, `mean()`, `std()` 等，本身是计算并返回一个结果（标量或新数组），因此它们是**非破坏性**的，不修改输入数组。

8. 数组的排序 (课件P49-P51)

- `np.sort(a, ...)`: **非破坏性**，返回数组的排序副本。
- `ndarray.sort(...)`: **破坏性**，原地对数组进行排序，不返回值 (返回 `None`)。
- `np.argsort(a, ...)`: **非破坏性**，返回索引数组。

```
arr_to_sort = np.array([3, 1, 2])
sorted_copy = np.sort(arr_to_sort) # 非破坏性
print(f"np.sort() 后原数组: {arr_to_sort}")
print(f"排序后的副本: {sorted_copy}")

arr_to_sort_inplace = np.array([5, 0, 8])
arr_to_sort_inplace.sort() # 破坏性
print(f"ndarray.sort() 后原数组: {arr_to_sort_inplace}")
```

对比列表的 `sort()` 和 `sorted()`:

- Python列表的 `list.sort()` 方法是破坏性的，原地排序。
- Python内置的 `sorted(list)` 函数是非破坏性的，返回新的排序列表。 NumPy 的 `ndarray.sort()` 和 `np.sort()` 与此类似。

9. 数组的转置和轴对换 (课件P38, P65)

- `arr.T`: 返回数组的**视图 (view)**，不是副本。因此，修改转置后的视图会影响原数组（如果原数组允许这种修改）。可以认为对原数据的结构性影响是破坏性的，但它不创建新数据。
- `arr.transpose(*axes)`: 通常返回原数组的**视图 (view)**。

```
arr_tr = np.array([[1,2],[3,4]])
transposed_view = arr_tr.T
transposed_view[0,0] = 100
print(f"修改转置视图后，原数组 arr_tr:\n{arr_tr}")
# [[100   2]
#  [  3   4]]
```

10. 数组的合并与分割 (课件P55-P62)

合并数组：

- `np.concatenate((a1, a2, ...), axis=0)`: 沿指定轴连接一系列数组。数组必须具有相同的形状（除了连接轴）。(课件P57)
- `np.vstack((tup))` 或 `np.row_stack((tup))`: 沿垂直方向（行）堆叠数组。(课件P58, P61)

- `np.hstack((tup))` 或 `np.column_stack((tup))`: 沿水平方向（列）堆叠数组。(课件P58, P60-P61)
- `np.append(arr, values, axis=None)`: 将值附加到数组的末尾。如果指定了 `axis`，则沿该轴附加。注意：`append` 返回的是新数组，不是原地修改。(课件P55-P56)
- `np.r_[]` 和 `np.c_[]`: 一种更简洁的堆叠方式。

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
print(f"concatenate ((a,b), axis=0):\n{np.concatenate((a, b), axis=0)}")
# [[1 2]
#  [3 4]
#  [5 6]]

c = np.array([[7],[8]])
print(f"hstack((a,c)):\n{np.hstack((a,c))}")
# [[1 2 7]
#  [3 4 8]]
```

分割数组：

- `np.split(ary, indices_or_sections, axis=0)`: 将数组沿指定轴分割成多个子数组。
- `np.hsplit(ary, indices_or_sections)`: 水平分割（按列）。
- `np.vsplit(ary, indices_or_sections)`: 垂直分割（按行）。

删除数组元素/行/列 (课件P62)：

- `np.delete(arr, obj, axis=None)`: 返回删除了指定子数组的新数组。`obj` 可以是整数、切片或整数数组，表示要删除的索引。

```
arr_del = np.arange(1, 10).reshape(3,3)
print(f"删除前:\n{arr_del}")
# 删除第1行 (索引0)
print(f"删除第1行后:\n{np.delete(arr_del, 0, axis=0)}")
# 删除第2列 (索引1)
print(f"删除第2列后:\n{np.delete(arr_del, 1, axis=1)}")
# 删除第1和第3列 (索引0和2)
print(f"删除第1和3列后:\n{np.delete(arr_del, [0,2], axis=1)}")
```

- **合并数组** (如 `np.concatenate`, `np.vstack`, `np.hstack`, `np.append`): 这些函数都是**非破坏性**的，它们创建一个新的数组来存储合并后的结果。
    - 特别注意 `np.append()`，它与Python列表的 `append()` 方法不同。列表的 `append()` 是破坏性的，而 `np.append()` 是非破坏性的，返回新数组。
- **分割数组** (如 `np.split`, `np.hsplit`, `np.vsplit`): 这些函数是**非破坏性**的，它们返回一个包含子数组（通常是视图）的列表。
- **删除数组元素/行/列** (`np.delete`): **非破坏性**，返回删除了指定部分的新数组。(课件P62)

11. 字符串操作 (`np.char`) (课件P44-P48)

NumPy 的 `np.char` 模块提供了一系列对字符串数组进行矢量化操作的函数。这些函数通常与 Python 内置字符串方法同名。

- `np.char.add(x1, x2)`: 元素级字符串连接。(课件P44)
- `np.char.multiply(a, i)`: 元素级字符串重复。(课件P45)
- `np.char.capitalize(a)`: 首字母大写。
- `np.char.lower(a)`, `np.char.upper(a)`: 转小写/大写。(课件P46)
- `np.char.split(a, sep=None, maxsplit=None)`: 元素级字符串分割。(课件P45)
- `np.char.join(sep, a)`: 使用指定分隔符连接字符串序列。
- `np.char.strip(a, chars=None)`: 去除首尾字符。
- `np.char.replace(a, old, new, count=None)`: 替换子串。
- `np.char.equal(x1, x2)`, `np.char.not_equal(x1, x2)` 等比较函数。(课件P46)

```
s1 = np.array(['I', 'finance', 'Python'])
s2 = np.array([' love', ' and', ' !'])
print(f"np.char.add(s1, s2): {np.char.add(s1, s2)}")
# ['I love' 'finance and' 'Python !']

s_case = np.array(['NumPy', 'PYTHON'])
print(f"np.char.lower(s_case): {np.char.lower(s_case)}")
# ['numpy' 'python']
```

`np.char` 模块中的函数通常是**非破坏性**的，它们对输入的字符串数组进行操作并返回新的字符串数组。

13. 线性代数 (`np.linalg`) (课件P63-P70)

NumPy 的 `linalg` 模块包含了一些线性代数函数。

- `np.dot(a, b)`: 两个数组的点积。对于二维数组，它是矩阵乘法。(课件P64, P66)
- `a @ b`: 矩阵乘法的另一种写法 (Python 3.5+)。(课件P66)
- `np.linalg.det(a)`: 计算矩阵的行列式。(课件P68)
- `np.linalg.inv(a)`: 计算矩阵的逆。(课件P68)
- `np.linalg.eig(a)`: 计算方阵的特征值和特征向量。(课件P69)
- `np.linalg.solve(a, b)`: 求解线性方程组 Ax = b。(课件P70)
- `arr.T`: 数组转置。(课件P65)

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print(f"A 点积 B (矩阵乘法):\n{np.dot(A, B)}")
# [[19 22]
#  [43 50]]

print(f"A 的行列式: {np.linalg.det(A)}") # 输出: -2.0

# 求解 Ax = v
v = np.array([10, 20])
x_solution = np.linalg.solve(A, v)
print(f"方程组解 x: {x_solution}")
```

14. 数组的副本 (Copy) 与视图 (View) (课件P52-P53) - 核心概念

这是理解NumPy中破坏性与非破坏性行为的关键。

- **无复制 (No Copy / Alias)**: `b = a`。`b` 是 `a` 的别名，指向同一块内存。修改 `b` 会影响 `a`。

  简单的赋值操作不会创建数组的副本，而是让新变量指向同一个数组对象（别名）。

```
a = np.array([1, 2, 3])
b = a
b[0] = 100
print(f"a: {a}") # a 也会被修改，输出: [100   2   3]
print(f"b is a: {b is a}") # True
```

- 视图 **(View / Shallow Copy)** (课件P52)：
    - 不同的数组对象可以共享相同的数据。视图是原始数组数据的一个新视角。
    - **对视图的修改会影响原始数组，反之亦然 (破坏性影响原数组数据)**。
    - **切片操作通常返回视图**。
    - `ndarray.view()` 方法可以创建一个视图。
- 副本 **(Copy / Deep Copy)** (课件P53)：
    - 副本是一个全新的数组，拥有自己的数据。
    - **对副本的修改不会影响原始数组 (非破坏性)**。
    - `ndarray.copy()` 方法可以创建一个深拷贝。
    - **花式索引和布尔索引通常返回副本**。

与Python列表的对比和易混淆点：

1. 切片：
    - **Python列表切片：** `list_slice = my_list[1:3]`，`list_slice` 是一个新的列表对象（副本）。修改 `list_slice` 不会影响 `my_list`。
    - **NumPy数组切片：** `arr_slice = my_arr[1:3]`，`arr_slice` 通常是 `my_arr` 的一个**视图**。修改 `arr_slice` 会影响 `my_arr`。这是非常重要的区别！如果需要副本，应使用 `my_arr[1:3].copy()`。
2. `append` 方法：
    - **Python列表** `list.append(item)`：**破坏性**操作，直接修改原列表，返回 `None`。
    - **NumPy** `np.append(arr, values, axis=None)`：**非破坏性**操作，返回一个新的数组，原数组不变。
3. 排序：
    - **Python列表** `list.sort()`：**破坏性**，原地排序。
    - **Python内置** `sorted(list)`：**非破坏性**，返回新列表。
    - **NumPy数组** `ndarray.sort()`：**破坏性**，原地排序。

- **NumPy** `np.sort(ndarray)`：**非破坏性**，返回新数组。 两者在方法和函数的设计上有一致性。

4. `resize`：
    - Python列表没有直接对应的 `resize` 方法来改变大小并可能用默认值填充/截断。可以通过切片赋值或 `del` 来改变大小。
    - NumPy的 `ndarray.resize()` 是**破坏性**的，而 `np.resize()` 是**非破坏性**的。

Python与大数据分析 --pandas的使用

1. Pandas 简介与数据结构 (课件P2, P4-P5)

- **Pandas 的用途** [cite: 476]：
    - 处理带异构列的表格数据，类似于 SQL 表或 Excel 电子表格 [cite: 476]。
    - 处理有序和无序的时间序列数据 [cite: 476]。
    - 处理带行列标签的矩阵数据 [cite: 476]。
    - 其他任意形式的观测/统计数据集 [cite: 476]。
- **核心数据结构**：
    - **Series (序列)** [cite: 479, 480]：
        - 一维标记数组，能够保存任何类型的数据 (整数、字符串、浮点数、Python对象等) [cite: 480]。
        - 轴标签统称为**索引 (index)** [cite: 480]。
        - 类似于带标签的一维数组或字典的列。
    - **DataFrame (数据帧)** [cite: 479, 480]：
        - 二维、大小可变的、具有潜在异构类型的表格数据结构，带有标记的轴（行和列） [cite: 480]。
        - 可以看作是一个 Excel 电子表格、SQL 表，或者一个 Series 对象的字典 [cite: 480]。
        - 是 Pandas 中最常用的数据结构 [cite: 480]。
        - DataFrame 有行索引 (index) 和列索引 (columns) [cite: 481]。

2. Series

- 创建 **Series (Creating Series)** (课件P6-P7)：
    - `pd.Series(data=None, index=None, dtype=None, name=None, copy=False)`
    - 从列表创建 [cite: 488]：

```
import pandas as pd
import numpy as np
s1 = pd.Series([47, 66, 48, 77, 16, 91])
print(s1)
# 0    47
# 1    66
# ...
# dtype: int64
```

      如果没有指定索引，会自动创建从0开始的整数索引 [cite: 489]。
    - 从 **NumPy** 数组创建 [cite: 486]：

```
data_np = np.array(['a', 'b', 'c', 'd'])
s_np = pd.Series(data_np)
print(s_np)
```

    - 指定索引创建 [cite: 493]：

```
s2 = pd.Series([47, 66, 48], index=['a', 'b', 'c'])
print(s2)
# a    47
# b    66
# c    48
# dtype: int64
```

    - 从字典创建 [cite: 499]：字典的键作为索引，值为 Series 的数据。

```
d3 = {'Name': 'Zhang San', 'Gender': 'Male',
'Age': 19}
s3 = pd.Series(d3)
print(s3)
# Name      Zhang San
# Gender         Male
# Age              19
# dtype: object
```

- 访问 **Series** 元素 **(Accessing Elements)** (课件P8-P10)：
    - 通过索引标签 [cite: 493]：

```
print(s2['a'])  # 输出: 47
print(s2[['a', 'c']]) # 输出: a 47, c 48
```

    - 通过位置下标 **(整数索引)** [cite: 494]： 即使有自定义标签索引，仍然可以通过整数位置访问。

```
print(s2[0]) # 输出: 47
print(s2[:2]) # 切片，输出: a 47, b 66
```

    - `.values` 和 `.index` 属性 [cite: 491]： 分别获取 Series 的数据（作为 NumPy 数组）和索引对象。

```
print(s1.values) # 输出: [47 66 48 77 16 91]
print(s1.index)  # 输出: RangeIndex(start=0,
stop=6, step=1)
```

- **Series** 常用操作与函数 (课件P11-P21)：
    - 向量化运算：支持 NumPy 风格的元素级运算，运算时会根据索引自动对齐数据 [cite: 498]。

```
print(s2 * 2)
# a     94
# b    132
# c     96
# dtype: int64
print(np.exp(s2.astype(float))) # 应用NumPy函数 (课
件P12)
```

    - 布尔索引 [cite: 495]：

```
print(s1[s1 > 50])
```

- **检查成员 (`in`)** [cite: 501]：判断索引标签是否存在。

```
print('Age' in s3)    # True
print('ID' in s3)     # False
```

- **处理缺失值 (NaN)**：
    - `pd.isnull(series)` 或 `series.isnull()` [cite: 510, 517]：检查哪些值是缺失的。
    - `pd.notnull(series)` 或 `series.notnull()` [cite: 514, 521]：检查哪些值不是缺失的。
    - 可以通过布尔索引筛选缺失或非缺失值 [cite: 526, 527]。
- **添加数据 (`.append()`)** [cite: 530, 531]：返回新的 Series，原 Series 不变（注意：Pandas Series 的 append 与 Python 列表的 append 行为不同，列表是原地修改）。
- **删除数据 (`.drop(label)`)** [cite: 530, 533]：返回新的 Series，原 Series 不变。
- **查看数据 (`.head(n=5)`, `.tail(n=5)`)** [cite: 538]：查看前n行或后n行数据。

3. DataFrame

- 创建 **DataFrame (Creating DataFrame)** (课件P24-P25)：
    - `pd.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)`
    - **从字典创建 (常用)** [cite: 550]：
        - **字典的键**成为 DataFrame 的**列名**。
        - **字典的值**（列表、NumPy数组或Series）成为 DataFrame 的**列数据**。所有列数据长度应一致。

```
data_dict = {'Name': ['Tom', 'Jack', 'Steve', 'Ricky'],
             'Age': [28, 34, 29, 42],
             'BMI': [17.5, 25.0, 21.0, 22.3]}
df_dict = pd.DataFrame(data_dict)
print(df_dict)
#     Name  Age   BMI
# 0    Tom   28  17.5
# 1   Jack   34  25.0
# 2  Steve   29  21.0
# 3  Ricky   42  22.3
```

可以指定 `index` 和 `columns` 参数来控制行索引和列顺序。

- **从嵌套列表创建** [cite: 553]：需要提供 `columns` 参数。

```
data_list = [['Tom', 28, 17.5], ['Jack', 34, 25.0]]
df_list = pd.DataFrame(data_list, columns=['Name', 'Age', 'BMI'])
print(df_list)
```

- **从嵌套字典创建** [cite: 556]：外层字典的键作为列名，内层字典的键作为行索引。

```
pop_data = {'Nevada': {2001: 2.4, 2002: 2.9},
            'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
df_pop = pd.DataFrame(pop_data)
print(df_pop) # 列是 Nevada, Ohio; 行是 2000, 2001, 2002
```

- **DataFrame 的基本属性与操作** (课件P29-P30)：
  - `.T`: 转置 DataFrame（行列互换）[cite: 578]。
  - `.shape`: 返回一个表示维度的元组 (行数, 列数) [cite: 582]。
  - `.size`: 返回元素的总数 [cite: 582]。
  - `.columns`: 返回列索引 [cite: 585]。
  - `.index`: 返回行索引 [cite: 585]。
  - `.dtypes`: 返回每列的数据类型。
  - `.head(n=5)`, `.tail(n=5)`: 查看前n行或后n行数据 [cite: 611]。

- `.info()`: 提供 DataFrame 的简洁摘要，包括数据类型、非空值数量和内存使用情况 [cite: 611, 612]。
- `.describe()`: 生成描述性统计数据，如计数、均值、标准差、最小值、最大值和分位数（默认只对数值列）[cite: 611, 615]。

4. 数据导入与导出 (Data IO) (课件P26-P28)

Pandas 提供了方便的函数来读写多种格式的数据文件。

- 读取数据 **(Reading Data)**:
    - **CSV** 文件: `pd.read_csv('filepath_or_buffer', sep=',', header='infer', index_col=None, usecols=None, parse_dates=False, nrows=None, ...)` [cite: 559, 565]
        - `sep`: 指定分隔符，默认为逗号。
        - `header`: 指定哪一行作为列名，默认为0（第一行）。若无列名则设为 `None`。
        - `index_col`: 指定某一列或几列作为行索引。
        - `usecols`: 选择要读取的列。
        - `parse_dates`: 指定要解析为日期的列。
        - `nrows`: 读取的文件行数。
    - **Excel** 文件: `pd.read_excel('io', sheet_name=0, header=0, index_col=None, usecols=None, ...)` [cite: 567, 569]
        - `sheet_name`: 指定工作表名称或索引。
    - 文本文件 **(如 .txt)**: `pd.read_table('filepath_or_buffer', sep='\t', ...)` [cite: 561, 563, 572, 574]
        - `read_table` 默认分隔符是制表符 `\t`。可以使用 `sep` 参数指定其他分隔符，可能需要设置 `engine='python'` [cite: 574]。
- 写入数据 **(Writing Data)** [cite: 577]:
    - `df.to_csv('path_or_buf', sep=',', index=True, header=True, ...)`
        - `index=False`: 通常用于在写入时不保存行索引。
    - `df.to_excel('excel_writer', sheet_name='Sheet1', index=True, header=True, ...)`

- `index=False`: 不写入行索引。

5. DataFrame 常用操作

这些操作通常涉及选择数据、清洗数据、转换数据和聚合数据。

- **查看数据 (Viewing Data)**：
    - `.head(n)`、`.tail(n)`：查看首尾数据 [cite: 611]。
    - `.info()`：查看 DataFrame 的概览信息 [cite: 611, 612]。
    - `.describe()`：获取数值列的描述性统计 [cite: 611, 615]。
- **选择数据 (Selection/Indexing)**：
    - **选择列** (课件P43)：
        - `df['column_name']`: 选择单列，返回 Series。
        - `df.column_name`: 另一种选择单列的方式（列名需符合变量命名规则且不与 DataFrame 方法冲突）。
        - `df[['col1', 'col2']]`: 选择多列，返回 DataFrame。
    - **使用 `.loc[]` (基于标签的索引)** (课件P44-P49)：
        - `df.loc[row_label]`: 选择单行。
        - `df.loc[[row_label1, row_label2]]`: 选择多行。
        - `df.loc[start_label:end_label]`: 行切片 (包含 `end_label`)。
        - `df.loc[:, column_label]`: 选择单列。
        - `df.loc[:, [col_label1, col_label2]]`: 选择多列。
        - `df.loc[row_label, col_label]`: 选择特定单元格。
        - `df.loc[boolean_array_rows, boolean_array_cols]`: 使用布尔数组进行选择 [cite: 650]。
        - `df.loc[callable_function]`: 使用可调用函数进行选择 [cite: 653]。
    - **使用 `.iloc[]` (基于整数位置的索引)** (课件P50)：
        - 用法与 `.loc[]` 类似，但使用的是整数位置而不是标签。
        - `df.iloc[row_position]`, `df.iloc[[0, 2]]`, `df.iloc[0:3]` (不包含位置3), `df.iloc[:, 0]`, `df.iloc[0, 0]`。
    - **条件选择 (Boolean Indexing)**：

```
df = pd.DataFrame({'Age': [22, 25, 30], 'Score': [80, 90, 85]})
print(df[df['Age'] > 25]) # 选择 Age 大于 25 的行
print(df.loc[df['Score'] >= 85, ['Age', 'Score']]) # 使用 .loc 进行条件选择和列选择
```

- `.query(expr)` 方法 (课件P51)： 使用字符串表达式进行查询。

```
df_query = pd.DataFrame({'Name': ['Tom', 'Steve'], 'BMI': [17.5, 21.0], 'Age': [28, 29]})
print(df_query.query('Age > 17 & Age < 29')) # 课件P61示例略作修改
```

- 数据处理与清洗：
  - 唯一值与计数 (课件P31, P37)：
    - `df['column'].unique()`: 返回该列中唯一值组成的 NumPy 数组 [cite: 589, 617]。
    - `df['column'].nunique()`: 返回唯一值的数量 [cite: 590, 617]。
    - `df['column'].value_counts()`: 返回每个唯一值及其出现的频率 [cite: 591, 618]。
  - 处理重复数据 (课件P32)：
    - `df.duplicated(subset=None, keep='first')`: 返回标记重复行的布尔 Series。
    - `df.drop_duplicates(subset=None, keep='first', inplace=False)`: 删除重复行 [cite: 593]。`keep` 参数可以是 'first' (保留第一个), 'last' (保留最后一个), 或 `False` (删除所有重复项) [cite: 593, 595]。
  - 数据替换 (课件P33-P35)：
    - `.replace(to_replace, value, ...)`: 替换值。`to_replace` 可以是标量、列表、字典等 [cite: 598]。
    - `.where(cond, other=np.nan, ...)`: 满足条件 `cond` 的地方保留原值，否则替换为 `other` (默认为NaN) [cite: 599, 600]。
    - `.mask(cond, other=np.nan, ...)`: 不满足条件 `cond` 的地方保留原值，否则替换为 `other` (默认为NaN) [cite: 599, 602]。
    - 数值操作：`.round(decimals)`, `.abs()`, `.clip(lower=None, upper=None)` [cite: 604, 605, 607, 609]。
  - 处理缺失值 **(NaN)** (常用方法，课件中Series部分有提及 `isnull`):

    - `df.isnull()` / `df.isna()`: 检测缺失值，返回布尔型 DataFrame。
    - `df.notnull()` / `df.notna()`: 检测非缺失值。
    - `df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)`: 删除包含缺失值的行或列。
    - `df.fillna(value=None, method=None, axis=None, inplace=False, limit=None, ...)`: 填充缺失值。`value` 可以是标量、字典、Series、DataFrame。`method` 可以是 `'ffill'` (向前填充) 或 `'bfill'` (向后填充)。
- 应用函数 **(Applying Functions)** (课件P39)：
    - `df.apply(func, axis=0, ...)`: 将函数 `func` 应用于 DataFrame 的行 (`axis=1`) 或列 (`axis=0`) [cite: 623]。
    - `df.applymap(func)` (DataFrame-only): 将函数 `func` 应用于 DataFrame 的每个元素。
    - `series.map(arg, na_action=None)` (Series-only): 根据输入对应关系映射 Series 的值。`arg` 可以是函数、字典或 Series。
- 排序 **(Sorting)** (课件P38)：
    - `.sort_values(by, axis=0, ascending=True, inplace=False, ...)`: 按一个或多个列的值排序 [cite: 620]。
    - `.sort_index(axis=0, level=None, ascending=True, inplace=False, ...)`: 按行或列的索引排序。
- 统计计算： DataFrame 的许多统计方法（如 `sum()`, `mean()`, `std()`, `min()`, `max()`, `count()`, `median()`, `nunique()`）可以按行或列进行计算，通过 `axis` 参数控制 (`axis=0` 对列操作， `axis=1` 对行操作)。`.describe()` 提供了数值列的汇总统计 [cite: 615]。
- 窗口函数 **(Window Functions)** (课件P40-P42)：
    - `.rolling(window, min_periods=None, center=False, ...).<aggregation_func>()`: 提供滚动窗口计算，如滚动均值、滚动和等 [cite: 626]。
    - `.expanding(min_periods=1, center=False, ...).<aggregation_func>()`: 提供扩展窗口计算，从序列开始处到当前位置 [cite: 632]。
    - `.shift(periods=1, freq=None, axis=0, ...)`: 将数据向前或向后移动指定的期数 [cite: 629]。

        - `.diff(periods=1, axis=0)`: 计算元素与前一个元素的差值（一阶差分）[cite: 629]。
        - `.pct_change(periods=1, fill_method='pad', limit=None, freq=None, ...)`: 计算百分比变化 [cite: 629]。
    - 索引操作 **(Index Manipulation)** (课件P52-P54)：
        - `df.set_index(keys, drop=True, append=False, inplace=False, ...)`: 将一个或多个现有列设为 DataFrame 的行索引 [cite: 665]。
        - `df.reset_index(level=None, drop=False, inplace=False, col_level=0, col_fill='')`: 重置索引，将当前的行索引转换为普通列。
        - `df.reindex(index=None, columns=None, ...)`: 根据新的索引重新调整 DataFrame，可以添加/删除/重排行和列，并可选地填充缺失值 [cite: 670, 671]。
    - 随机抽样 **(Random Sampling)** (课件P55, P59):
        - `df.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None)`: 从DataFrame中随机抽取行或列。

##

pandas高级操作-数据处理

1. 分组 (Grouping) (课件P2-P13)

分组操作是数据分析中非常常见的步骤，它允许我们根据一个或多个标准将数据分割成组，然后对每个组独立地应用函数。

- 分组的基本模式 (课件P2)： `df.groupby(分组依据)[数据来源].聚合操作()`
    - **分组依据 (`by`)**：可以是一个或多个列名、一个Series、一个与DataFrame行数相同的列表或数组，甚至是一个函数。
    - **数据来源**：指定要进行聚合操作的列（一个或多个）。如果省略，则聚合操作会尝试作用于所有有效的列。
    - **聚合操作**：对分组后的数据执行的计算，如 `mean()`, `sum()`, `count()`, `median()`, `max()`, `min()` 等。

```
import pandas as pd
# 假设 df 是一个包含学生信息的DataFrame，有 'Gender' 和 'Height' 列
# df = pd.read_csv('../data/learn_pandas.csv') # 课件示例数据加载
# 按性别统计身高的中位数 (课件P2)
# result = df.groupby('Gender')['Height'].median()
# print(result)
```

- 分组依据的本质 (课件P3-P4)：
    - 多列分组：在 `groupby()` 中传入列名列表。

```
# 按学校和性别分组，统计身高的均值 (课件P3)
# result_multi_group = df.groupby(['School', 'Gender'])['Height'].mean()
# print(result_multi_group)
```

    - 基于条件表达式分组：可以将一个布尔Series作为分组依据。

```
# 根据体重是否超过总体均值来分组，计算身高的均值 (课件P4)
# condition = df['Weight'] > df['Weight'].mean()
# result_condition_group = df.groupby(condition)['Height'].mean()
# print(result_condition_group)
```

- `GroupBy` 对象 (课件P5-P6)： `df.groupby()` 操作返回的是一个 `GroupBy` 对象。这个对象本身并不直接显示数据，但它包含了分组后的信息，并提供了许多方法来进行后续计算。
    - `.ngroups`: 返回分组的数量。
    - `.groups`: 返回一个字典，键是组名（元组形式，如果多重分组），值是该组对应的行索引列表。
    - `.get_group(name)`: 获取指定名称的组，返回一个DataFrame。
- 分组的三大操作：聚合 **(Aggregation)**、变换 **(Transformation)**、过滤 **(Filtering)** (课件P7)： Pandas 提供了专门的函数 `.agg()`, `.transform()`, `.filter()` 来执行这三种核心的分组操作。
    - 聚合 **(Aggregation) (`.agg()` 或直接调用聚合函数)** (课件P8-P13)： 聚合操作将每个组的数据缩减为一个标量值（或少数几个值）。

- **内置聚合函数** (课件P8)：可以直接在 `GroupBy` 对象上调用，如 `mean()`, `sum()`, `size()`, `count()`, `max()`, `min()`, `median()`, `std()`, `var()`, `quantile()`, `nunique()` 等。

```
# gb = df.groupby('Gender')['Height']
# print(gb.mean())
# print(gb.quantile(0.95))
```

- `.agg()` **方法** (课件P9-P13)：提供了更大的灵活性。
  - **使用多个函数**：传入函数名字符串的列表。结果列会形成多级索引。(课件P10)

```
# gb_multi_col = df.groupby('Gender')[['Height', 'Weight']]
# result_agg_multi_func = gb_multi_col.agg(['sum', 'idxmax', 'skew'])
# print(result_agg_multi_func)
```

  - **对特定列使用特定聚合函数**：传入一个字典，键为列名，值为函数（或函数列表）。(课件P11)

```
# result_agg_specific_col = gb_multi_col.agg({'Height': ['mean', 'max'], 'Weight': 'count'})
# print(result_agg_specific_col)
```

  - **使用自定义函数**：可以直接传入lambda函数或已定义的函数。(课件P12)

```
# result_agg_custom_func = gb_multi_col.agg(lambda x: x.max() - x.min()) # 计算极差
# print(result_agg_custom_func)
```

  - **聚合结果重命名**：将函数位置改写成元组 `(new_name, function)`。(课件P13)

```
# result_agg_rename = gb_multi_col.agg([('range', lambda x: x.max() - x.min()), ('my_sum', 'sum')])
# print(result_agg_rename)
```

- **变换 (Transformation) (`.transform()`)**: 变换操作对每个组应用一个函数，但返回的结果与原始DataFrame具有相同的形状（相同的索引）。函数应用于每个组后，结果会广播回原始的形状。常用于组内标准化、填充缺失值等。

```
# 示例：组内标准化 (z-score)
# zscore = lambda x: (x - x.mean()) / x.std()
# transformed_height = df.groupby('Gender')['Height'].transform(zscore)
# print(transformed_height.head())
```

- **过滤 (Filtering) (`.filter()`)**: 过滤操作根据对每个组计算得到的布尔值来决定是否保留该组的所有行。函数应用于每个组，如果函数返回 `True`，则该组的所有行被保留；如果返回 `False`，则该组的所有行被丢弃。

```
# 示例：只保留平均身高大于160的性别组
# filtered_df = df.groupby('Gender').filter(lambda x: x['Height'].mean() > 160)
# print(filtered_df['Gender'].unique())
```

2. 数据连接 (Joining and Merging) (课件P14-P20)

Pandas 提供了多种方式来合并或连接不同的 DataFrame。

- **关系型连接 (`pd.merge()` 和 `df.join()`)** (课件P14-P17)： 类似于 SQL 中的 JOIN 操作。
    - `pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, suffixes=('_x', '_y'), ...)` (课件P15-P16):
        - `left`, `right`: 要合并的两个 DataFrame。
        - `how`: 连接方式，可以是：
            - `'inner'` (内连接)：只保留两个表中连接键都存在的部分 (课件P14图示)。

        - `'left'` (左连接)：保留左表所有行，右表中匹配不上的用 NaN 填充 (课件P14图示, P15示例)。
        - `'right'` (右连接)：保留右表所有行，左表中匹配不上的用 NaN 填充 (课件P14图示)。
        - `'outer'` (外连接)：保留两个表的所有行，匹配不上的用 NaN 填充 (课件P14图示)。
    - `on`: 用于连接的列名（键）。如果左右两表中的键列名相同，则使用此参数。可以是单个列名或列名列表。
    - `left_on`, `right_on`: 当左右两表中用于连接的键列名不同时，分别指定左右表的列名。
    - `left_index`, `right_index`: 布尔值，如果为 `True`，则使用对应表的行索引作为连接键。
    - `suffixes`: 如果合并后存在同名列（非连接键），则为它们添加后缀以区分。

```
# 值连接示例 (课件P15)
# df1 = pd.DataFrame({'Name': ['San Zhang', 'Si Li'], 'Age': [20, 30]})
# df2 = pd.DataFrame({'Name': ['Si Li', 'Wu Wang'], 'Gender': ['F', 'M']})
# merged_left = df1.merge(df2, on='Name', how='left')
# print(merged_left)

# 使用 left_on 和 right_on (课件P16)
# df1_diff_key = pd.DataFrame({'df1_name': ['San Zhang', 'Si Li'], 'Age': [20, 30]})
# df2_diff_key = pd.DataFrame({'df2_name': ['Si Li', 'Wu Wang'], 'Gender': ['F', 'M']})
# merged_diff_keys = df1_diff_key.merge(df2_diff_key, left_on='df1_name', right_on='df2_name', how='left')
# print(merged_diff_keys)
```

- `DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', ...)` (课件P17)： 主要用于基于索引的连接，也可以基于列连接。
    - `other`: 另一个 DataFrame。

- `on`: `self` DataFrame 中用于连接的列名（`other` DataFrame 必须以索引连接）。
- `how`: 连接方式，同 `merge`。
- `lsuffix`, `rsuffix`: 用于处理重叠列名的后缀。 默认情况下，`join` 按索引连接。

```
# 索引连接示例 (课件P17)
# df1_idx = pd.DataFrame({'Age': [20, 30]}, index=pd.Series(['San Zhang', 'Si Li'], name='Name'))
# df2_idx = pd.DataFrame({'Gender': ['F', 'M']}, index=pd.Series(['Si Li', 'Wu Wang'], name='Name'))
# joined_df = df1_idx.join(df2_idx, how='left')
# print(joined_df)
```

- **方向连接/拼接 (`pd.concat()`)** (课件P18-P20)： 用于沿某个轴将多个 Series 或 DataFrame 对象堆叠在一起。 `pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None, ...)`
  - `objs`: 一个序列或 Series/DataFrame 对象的映射。
  - `axis`: 拼接方向。`0` 表示按行拼接（垂直方向，默认），`1` 表示按列拼接（水平方向）。
  - `join`: 处理非连接轴上的索引对齐方式。
    - `'outer'` (默认)：取所有索引的并集，缺失处用 NaN 填充。
    - `'inner'`：取索引的交集，只保留共有的部分。
  - `ignore_index`: 如果为 `True`，则不使用原始索引，而是生成新的从0开始的整数索引。
  - `keys`: 用于在连接轴上创建一个层次化索引，以区分原始对象。

```
# 纵向拼接 (课件P18)
# df1_concat = pd.DataFrame({'Name': ['San Zhang', 'Si Li'], 'Age': [20, 30]})
# df2_concat = pd.DataFrame({'Name': ['Wu Wang'], 'Age': [40]})
# concat_vertical = pd.concat([df1_concat, df2_concat], ignore_index=True) # ignore_index 避免索引重复
# print(concat_vertical)

# 横向拼接 (课件P19)
# df_grade = pd.DataFrame({'Grade': [80, 90]})
```

```
# df_gender = pd.DataFrame({'Gender': ['M', 'F']})
# concat_horizontal = pd.concat([df1_concat, df_grade, df_gender], axis=1)
# print(concat_horizontal)

# join 参数示例 (课件P20)
# df_concat_inner_join = pd.concat([df1_concat, df_grade_diff_index], axis=1, join='inner')
# print(df_concat_inner_join)
```

3. 数据变形 (Reshaping Data) (课件P21-P30)

数据变形是指改变 DataFrame 的结构，如长宽表之间的转换，或索引与列之间的转换。

- 长宽表的变形 (`pivot`, `pivot_table`, `melt`) (课件P21-P27)：
  - 长表 (Long Format)：每一行代表一个观测，变量和值在不同的列中。
  - 宽表 (Wide Format)：将某个变量的不同取值作为列名，形成更宽的表。
  - `DataFrame.pivot(index=None, columns=None, values=None)` (课件P22-P24)： 用于将长表转换为宽表。
    - `index`: 新 DataFrame 的行索引来源列。
    - `columns`: 新 DataFrame 的列索引来源列。
    - `values`: 新 DataFrame 的值来源列。 **注意：** `pivot` 要求 `index` 和 `columns` 的组合必须唯一，否则会报错。如果存在重复组合，需要使用 `pivot_table` 进行聚合。

```
# 长表转宽表示例 (课件P22-P23)
# data_long = {'Class': [1, 1, 2, 2],
#              'Name': ['San Zhang', 'San Zhang', 'Si Li', 'Si Li'],
#              'Subject': ['Chinese', 'Math', 'Chinese', 'Math'],
#              'Grade': [80, 75, 90, 85]}
# df_long = pd.DataFrame(data_long)
# df_wide = df_long.pivot(index='Name', columns='Subject', values='Grade')
# print(df_wide)
```
```pivot` 也支持多级 `index` 和 `columns`，以及多个 `values` 列 (课件P24)。
```

- `pd.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, ...)` (课件P25-P26)： 功能更强大的透视表工具，类似于 Excel 中的数据透视表。
    - 当 `index` 和 `columns` 的组合不唯一时， `pivot_table` 可以通过 `aggfunc` 参数指定的聚合函数（如 `mean`, `sum`, `count`）来处理重复值。
    - `margins=True`: 添加行/列的边际汇总（总计/平均等）。

```
# pivot_table 聚合示例 (课件P25)
# data_scores = {'Name': ['San Zhang', 'San Zhang', 'San Zhang', 'San Zhang', 'Si Li', 'Si Li', 'Si Li', 'Si Li'],
#                'Subject': ['Chinese', 'Chinese', 'Math', 'Math', 'Chinese', 'Chinese', 'Math', 'Math'],
#                'Grade': [80, 90, 100, 90, 70, 80, 85, 95]}
# df_scores = pd.DataFrame(data_scores)
# df_pivot_agg = df_scores.pivot_table(index='Name', columns='Subject', values='Grade', aggfunc='mean')
# print(df_pivot_agg)
# df_pivot_margins = df_scores.pivot_table(index='Name', columns='Subject', values='Grade', aggfunc='mean', margins=True)
# print(df_pivot_margins)
```

- `pd.melt(frame, id_vars=None, value_vars=None, var_name=None, value_name='value', ...)` (课件P27)： `pivot` 的逆操作，用于将宽表转换为长表，将列名“熔化”到行中。
    - `id_vars`: 不需要被转换（保持为标识列）的列名。
    - `value_vars`: 需要被转换到行中的列名。如果未指定，则除 `id_vars` 外的所有列都会被转换。
    - `var_name`: 用于存储原列名的新列的名称。
    - `value_name`: 用于存储原单元格值的新列的名称。

```
# 宽表转长表示例 (课件P27)
# df_wide_to_melt = pd.DataFrame({'Class': [1, 2],
#                                 'Name': ['San Zhang', 'Si Li'],
#                                 'Chinese': [80, 90],
#                                 'Math': [80, 75]})
# df_melted = df_wide_to_melt.melt(id_vars=['Class', 'Name'],
#                                  value_vars=['Chinese', 'Math'],
#                                  var_name='Subject',
#                                  value_name='Grade')
# print(df_melted)
```

- **索引的变形 (`stack`, `unstack`)** (课件P28-P30)： 这两个函数主要用于处理具有层次化索引 (MultiIndex) 的 Series 或 DataFrame，在行索引和列索引之间移动层级。
    - `DataFrame.unstack(level=-1, fill_value=None)` (课件P28-P29)：将行索引的指定层级“解堆”到列索引的最内层。`level` 参数指定要移动的行索引层级（可以是名称或整数位置，默认最内层）。
    - `DataFrame.stack(level=-1, dropna=True)` (课件P30)： `unstack` 的逆操作，将列索引的指定层级“堆叠”到行索引的最内层。

  这些操作在处理多维数据或重塑数据以便于分析时非常有用。

4. 文本数据处理 (`.str` 访问器) (课件P31-P42)

Pandas Series 提供了一个特殊的 `.str` 访问器，可以方便地对 Series 中的每个字符串元素应用 Python 内置的字符串方法或正则表达式操作。

- **`.str` 对象的设计意图** (课件P31)： 对 Series 中的文本数据进行向量化（元素级）操作。

```
s_text = pd.Series(['abcd', 'efg', 'hi'])
print(s_text.str.upper()) # 将所有字符串转为大写
# 0    ABCD
# 1     EFG
# 2      HI
# dtype: object
```

- `.str[]` **索引器** (课件P32)： 类似于字符串的索引和切片，作用于 Series 中的每个字符串。

```
print(s_text.str[0])     # 取每个字符串的第一个字符
print(s_text.str[-1::-2]) # 对每个字符串进行反向步长切片 (课件示例 'db')
```

- **文本数据的五类主要操作：**

  a. **拆分 (`.str.split()`)** (课件P33)： 将每个字符串按指定的分隔符（可以是正则表达式）拆分成列表。

  - `n`: 最大拆分次数。
  - `expand=True`: 将拆分后的列表展开为 DataFrame 的多列。

```
s_address = pd.Series(['上海市黄浦区方浜中路249号', '上海市宝山区密山路5号'])
print(s_address.str.split('[市区路]', expand=True, n=2))
```

  b. **合并 (`.str.join()`, `.str.cat()`)** (课件P34-P35)：

  - `.str.join(sep)`: 对于 Series 中每个元素是字符串列表的情况，用指定分隔符 `sep` 连接列表内的字符串。
  - `.str.cat(others=None, sep=None, na_rep=None, join='left')`: 将 Series 中的字符串与另一个 Series（或列表）中的字符串按元素连接起来。
    - `sep`: 连接符。
    - `na_rep`: 缺失值的替代表示。
    - `join`: 连接方式（基于索引）。

```
s1_cat = pd.Series(['a', 'b'])
s2_cat = pd.Series(['cat', 'dog'])
print(s1_cat.str.cat(s2_cat, sep='-'))
# 0    a-cat
# 1    b-dog
# dtype: object
```

c. 匹配 (`.str.contains()`, `.str.startswith()`, `.str.endswith()`, `.str.match()`, `.str.find()`, `.str.rfind()`) (课件P36-P38):

- `.str.contains(pat, case=True, flags=0, na=nan, regex=True)`: 检查每个字符串是否包含指定的模式（`pat` 可以是正则表达式）。返回布尔 Series。
- `.str.startswith(pat)`, `.str.endswith(pat)`: 检查字符串是否以指定模式开始/结束（不支持正则）。返回布尔 Series。
- `.str.match(pat, case=True, flags=0, na=nan)`: 检查字符串的开头是否匹配正则表达式。返回布尔 Series。
- `.str.find(sub, start=0, end=None)`, `.str.rfind(sub, start=0, end=None)`: 返回子字符串 `sub` 首次/末次出现的位置索引，未找到则返回 -1（不支持正则）。

```
s_match = pd.Series(['my cat', 'he is fat', 'railway station'])
print(s_match.str.contains(r'\s\w+at')) # 匹配一个空格后跟多个字母数字下划线再跟at
print(s_match.str.startswith('my'))
```

d. 替换 (`.str.replace()`) (课件P39): `Series.str.replace(pat, repl, n=-1, case=None, flags=0, regex=True)` 替换字符串中匹配 `pat`（可以是字符串或正则表达式）的部分为 `repl`（可以是字符串或可调用函数）。

```
s_replace = pd.Series(['a_1_b', 'c_?'])
print(s_replace.str.replace(r'\d|\?', 'new', regex=True)) # 将数字或问号替换为new
# 0    a_new_b
# 1      c_new
# dtype: object
```

可以使用正则表达式的捕获组和自定义函数进行更复杂的替换。

e. **提取 (`.str.extract()`, `.str.extractall()`)** (课件P40-P42)： 使用正则表达式从每个字符串中提取匹配的组。

- `.str.extract(pat, flags=0, expand=True)`: 对每个字符串只提取第一个匹配项。如果正则表达式包含捕获组，`expand=True` (默认) 会将每个组作为 DataFrame 的一列返回。
- `.str.extractall(pat, flags=0)`: 提取所有匹配项。如果一个字符串有多个匹配，会产生多级索引。

```
s_extract = pd.Series(['A135T15, A26S5', 'B674S2, B25T6'])
pat_extract = r'[AB](\d+)[TS](\d+)' # 提取数字部分
print(s_extract.str.extract(pat_extract)) # 只提取第一个匹配
print(s_extract.str.extractall(pat_extract)) # 提取所有匹配
```

可以通过命名捕获组 `(?P<name>...)` 来直接为提取结果的列命名。

5. Kaggle 项目示例中的 Pandas 数据整理 (课件P43-P54)

这部分课件通过一个宝可梦数据集的实际案例，演示了前面学到的多种 Pandas 操作的综合运用，包括：

- **数据概览**：`.head()`, `.info()`, `.corr()` (计算相关系数矩阵)。
- **数据可视化配合**：如使用 `seaborn.heatmap` 可视化相关系数，使用 `boxplot` 进行分组箱线图分析。
- **数据变形**：使用 `.melt()` 将宽表转换为长表，便于某些类型的分析或可视化。
- **数据类型转换**：使用 `.astype()` 转换列的数据类型（如 object 转 category，int 转 float）。
- **处理缺失数据**：
  - 检查缺失值：`.info()`, `df['col'].value_counts(dropna=False)`。
  - 删除缺失值：`.dropna(inplace=True)` (课件中提到但随后用 `fillna` 覆盖了)。
  - 填充缺失值：`df['col'].fillna('replacement_value', inplace=True)`。
  - 使用 `assert` 语句进行数据校验，确保操作符合预期。

- **条件过滤**：使用布尔序列和逻辑操作符 (`&`, `|`) 进行多条件行筛选。
- **数据转换**：使用 `.apply()` 配合自定义函数或 lambda 函数对列数据进行转换。

Python与大数据分析 --数据可视化基础

1. 可视化流程 (一般概念) (课件P4)： 一个典型的可视化流程可能包括以下步骤：

- a. **定义问题 (Define Content)**：明确要通过可视化解决的问题，想要探索的假设，以及需要寻找的关键指标。
  b. **获取数据 (Acquire Data)**：收集与问题相关的数据。
  c. **数据预处理 (Parse/Filter/Mine Data)**：清洗数据，处理缺失值、异常值，转换数据格式，提取有用特征。
  d. **数据分析与模型构建 (Data Analysis & Model Building)**：进行统计分析，或构建预测模型。
  e. **选择可视化方法/映射 (Choose Visual Mapping)**：根据数据类型和要表达的信息，选择合适的图表类型和视觉编码（如颜色、形状、大小）。
  f. **可视化呈现 (Visualize/Represent)**：使用工具生成图表。
  g. **用户交互与改进 (User Interaction & Refine)**：可能需要根据反馈调整可视化设计，或者允许用户与图表交互以探索更多细节。
  h. **归纳信息/形成报告 (Induct Information & Report)**：从可视化结果中提炼见解，形成结论或报告。 这是一个迭代的过程，可能需要多次循环和调整。

2. Matplotlib 简介与使用 (课件P6-P11)

- 基本使用流程 (课件P11)：
  a. 导入 `pyplot` 模块：通常将其导入为 `plt`。

```
import matplotlib.pyplot as plt
import numpy as np # 通常与matplotlib一起使用
```

  b. 准备数据：通常使用 NumPy 数组或 Pandas Series/DataFrame 作为绘图数据。
  c. 创建图形 **(Figure)** 和坐标轴 **(Axes)**：

- `plt.figure(figsize=(width, height), dpi=value)`: 创建一个新的图形窗口。`figsize` 指定宽度和高度（英寸），`dpi` 指定分辨率。
- 可以直接调用绘图函数（如 `plt.plot()`），Matplotlib 会自动创建图形和坐标轴。
- 对于更复杂的布局或多个子图，通常会显式创建 Figure 和 Axes 对象 (例如使用 `plt.subplots()`)。

d. **绘图**：调用具体的绘图函数，如 `plt.plot()`, `plt.scatter()`, `plt.bar()` 等。

e. **自定义图表**：添加标题、坐标轴标签、图例、网格，调整颜色、线型、标记样式等。

f. **显示图表**：`plt.show()`。

g. **保存图表** (可选)：`plt.savefig('filename.png', dpi=value)`。

3. 常用图表类型及其绘制

3.1 折线图 (`plt.plot()`) (课件P12-P14, P21-P28)

折线图用于显示数据随某个连续变量（通常是时间或顺序）变化的趋势。

- **基本绘制：**

```
# 绘制正弦函数曲线 (课件P12)
x = np.arange(0, 2 * np.pi, 0.1)
y_sin = np.sin(x)
plt.plot(x, y_sin)
plt.title("正弦函数曲线")
plt.xlabel("x轴")
plt.ylabel("sin(x)")
plt.show()
```

- **自定义样式** (课件P13-P14, P24, P33)： `plt.plot(x, y, color='red', linewidth=2, linestyle='--', marker='o', label='数据1')`
  - `color`: 线条颜色 (如 'red', 'blue', 'g', 'k', '#FFDD44')。
  - `linewidth`: 线条宽度。
  - `linestyle` 或 `ls`: 线条样式 (如 '-', '--', '-.', ':')。

    - `marker`: 数据点标记样式 (如 'o', '.', ',', 'x', '+', '*', 's', 'D', '^')。
    - `label`: 图例中显示的标签。
- **绘制多条折线** (课件P23-P26)： 在同一个坐标轴上多次调用 `plt.plot()`。

```
# 同时绘制正弦和余弦函数 (课件P23)
x = np.arange(0, 2 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

plt.plot(x, y_sin, 'ro--', label='正弦') # 红色圆点虚线 (课件P24, P25)
plt.plot(x, y_cos, 'b*-.', label='余弦') # 蓝色星点点划线
plt.title("正弦与余弦函数")
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc='lower right') # 显示图例，loc指定位置 (课件P25-P26)
plt.grid(True) # 显示网格 (课件P28)
plt.show()
```

- **设置坐标轴刻度** (`plt.xticks()`, `plt.yticks()`) (课件P27)： 可以指定刻度显示的位置和标签。

```
# X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
# C, S = np.cos(X), np.sin(X)
# plt.plot(X, C)
# plt.plot(X, S)
# plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
#            [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$\pi/2$', r'$\pi$']) # 使用LaTeX格式化标签
# plt.yticks([-1, 0, +1])
# plt.show()
```

3.2 散点图 (`plt.scatter()`) (课件P16-P20)

散点图用于显示两个变量之间的关系，每个数据点在图上表示为一个点。

- **基本绘制**： `plt.scatter(x, y, s=None, c=None, marker='o', cmap=None, alpha=None, label=None)`
    - `s`: 点的大小，可以是标量或数组。

- `c`: 点的颜色，可以是颜色字符串、颜色序列或用于颜色映射的数值序列。
- `marker`: 点的形状。
- `cmap`: 当 `c` 是数值序列时使用的颜色映射表。
- `alpha`: 透明度。

```
# 简单散点图 (课件P16)
x_scatter = [1, 2]
y_scatter = [2, 4]
plt.scatter(x_scatter, y_scatter, s=200) # s设置点的大小
plt.title("简单散点图")
plt.show()

# 随机散点图，自定义大小和颜色 (课件P18)
# n = 50
# xr = np.random.rand(n)
# yr = np.random.rand(n)
# area = np.pi * (15 * np.random.rand(n))**2 # 点大小随机
# colors = np.random.rand(n) # 颜色随机
# plt.scatter(xr, yr, s=area, c=colors, alpha=0.5, cmap='viridis')
# plt.colorbar() # 显示颜色条
# plt.show()
```

3.3 柱状图/条形图 (`plt.bar()`, `plt.barh()`) (课件P35-P38, P42)

柱状图用于比较不同类别的数据量或频率。`plt.barh()` 用于绘制水平条形图。

- 基本绘制： `plt.bar(x, height, width=0.8, bottom=None, align='center', color=None, edgecolor=None, label=None)`
    - `x`: 条形的x坐标（类别位置）。
    - `height`: 条形的高度（数值）。
    - `width`: 条形的宽度。
    - `bottom`: 堆叠柱状图的起始高度。

```
# 简单柱状图 (课件P35)
categories = ['A', 'B', 'C']
values = [10, 25, 15]
plt.bar(categories, values, color=['red', 'green', 'blue'], width=0.5) # (课件P38颜色示例)
plt.title("简单柱状图")
plt.xlabel("类别")
plt.ylabel("数量")
# plt.xticks([0,1,2], ['男','女','未知']) # 设置x轴刻度标签 (课件P37)
plt.show()
```

- 堆叠柱状图 (课件P42)：通过设置 `bottom` 参数实现。

```
# N = 5
# menMeans = (20, 35, 30, 35, 27)
# womenMeans = (25, 32, 34, 20, 25)
# ind = np.arange(N)
# width = 0.45
# p1 = plt.bar(ind, menMeans, width, color='grey')
# p2 = plt.bar(ind, womenMeans, width, color='lightblue', bottom=menMeans) # bottom参数
# plt.ylabel('Scores')
# plt.title('Scores by group and gender')
# plt.xticks(ind, ('G1', 'G2', 'G3', 'G4', 'G5'))
# plt.yticks(np.arange(0, 81, 10))
# plt.legend((p1[0], p2[0]), ('Men', 'Women'))
# plt.show()
```

3.4 直方图 (`plt.hist()`) (课件P40-P41)

直方图用于显示连续数据或离散数据（大量数据点时）的分布情况。

- 基本绘制：`plt.hist(x, bins=None, range=None, density=False, weights=None, cumulative=False, color=None, label=None)`
    - `x`: 输入数据（一维数组或序列）。
    - `bins`: 定义箱子的数量或边界。
    - `density`: 如果为 `True`，则归一化直方图，使其面积为1。

```
# 假设 housing_df 是包含房价数据的 Pandas DataFrame (课件P39-P41)
# data_hist = housing_df['Avg. Area Number of Rooms']
# plt.hist(data_hist, bins=10, edgecolor='black')
# plt.title('房间数量分布直方图')
# plt.xlabel('平均房间数')
# plt.ylabel('频数')
# plt.show()
```

3.5 饼图 (`plt.pie()`) (课件P43-P44)

饼图用于显示各部分占总体的百分比。

- 基本绘制： `plt.pie(x, explode=None, labels=None, colors=None, autopct=None, shadow=False, startangle=0)`
    - `x`: 各部分的值。
    - `explode`: 指定某几块“爆炸”出来，突出显示。
    - `labels`: 各部分的标签。
    - `colors`: 各部分的颜色。
    - `autopct`: 控制百分比显示的格式，如 `'%1.1f%%'`。
    - `startangle`: 起始角度。

```
# 简单饼图 (课件P43)
edu_levels = [0.2515, 0.3724, 0.3336, 0.0368, 0.0057]
labels = ['中专', '大专', '本科', '硕士', '其他']
explode = [0, 0.1, 0, 0, 0] # 第二块“大专”突出显示 (课件P44)
colors = ['#9999ff', '#ff9999', '#7777aa', '#2442aa', '#dd5555'] # (课件P44)

plt.rcParams['font.sans-serif'] = ['SimHei'] # 解决中文显示问题 (课件P32, P43)
plt.rcParams['axes.unicode_minus'] = False # 解决负号显示问题

plt.pie(edu_levels, explode=explode, labels=labels, colors=colors,
        autopct='%.1f%%', shadow=True, startangle=90,
        wedgeprops={'linewidth': 1, 'edgecolor': 'green'}, # (课件P44)
```

```
        textprops={'fontsize': 10, 'color': 'black'})
    # (课件P44)
plt.title('用户教育水平分布')
plt.axis('equal') # 保证饼图是正圆形
plt.show()
```

4. 图表元素自定义 (课件多处涉及)

- **标题：** `plt.title('图表标题', fontsize=16)` (课件P17, P22)
- **坐标轴标签：** `plt.xlabel('X轴标签', fontsize=14)`, `plt.ylabel('Y轴标签', fontsize=14)` (课件P13, P17, P22)
- **图例：** `plt.legend(loc='best', frameon=False, bbox_to_anchor=None)` (课件P13, P25-P26)
    - `loc`: 图例位置 (如 'upper right', 'lower left', 或数字代码)。
    - `frameon`: 是否显示图例边框。
    - `bbox_to_anchor`: 自定义图例位置。
- **刻度：**
    - `plt.xticks()`, `plt.yticks()`: 设置刻度位置和标签 (课件P27)。
    - `plt.tick_params(axis='both', which='major', labelsize=14)`: 自定义刻度外观 (课件P17)。
- **网格：** `plt.grid(True)` (课件P28)
- **图像大小和分辨率：** `plt.figure(figsize=(w, h), dpi=val)` (课件P11, P28 `fig.set_size_inches()`)
- **保存图像：** `plt.savefig('filename.png', dpi=300)` (课件P15)
- **解决中文乱码** (课件P32)：

```
plt.rcParams['font.sans-serif'] = ['SimHei'] # 指定默认字体
plt.rcParams['axes.unicode_minus'] = False   # 解决保存图像是负号'-'显示为方块的问题
```

5. 子图 (Subplots) (课件P29-P30)

在一个图形窗口中绘制多个独立的图表。

- `plt.subplot(nrows, ncols, index)`: 将当前图形划分为 `nrows` 行 `ncols` 列的网格，并选择第 `index` 个子图（从1开始计数）作为当前绘图区域。

```
# X1 = np.linspace(-np.pi, np.pi, 256, endpoint=True)
# C1, S1 = np.cos(X1), np.sin(X1)
# X2 = np.linspace(-10, 10, 256, endpoint=True)
# C2, S2 = np.cos(X2), np.sin(X2)

# plt.figure(figsize=(10, 8)) # 创建一个大一点的图形窗口

# plt.subplot(2, 2, 1) # 2行2列的第1个
# plt.plot(X1, C1)
# plt.title("fig1")

# plt.subplot(2, 2, 2) # 2行2列的第2个
# plt.plot(X1, S1, 'r')
# plt.title("fig2")

# plt.subplot(2, 2, 3) # 2行2列的第3个
# plt.plot(X2, C2, 'k')
# plt.title("fig3")

# plt.subplot(2, 2, 4) # 2行2列的第4个
# plt.plot(X2, S2, 'c')
# plt.title("fig4")

# plt.subplots_adjust(hspace=0.4, wspace=0.3) # 调整子图间距
# plt.show()
```

- `plt.subplots(nrows=1, ncols=1, sharex=False, sharey=False, figsize=None)`: 更推荐的方式，一次性创建图形和所有子图的 Axes 对象。返回一个 Figure 对象和一个包含 Axes 对象的数组（或单个 Axes 对象）。

```
# fig, axes = plt.subplots(2, 2, figsize=(10, 8)) # axes是一个2x2的NumPy数组
# axes[0, 0].plot(X1, C1)
# axes[0, 0].set_title("fig1 (面向对象)")
# axes[0, 1].plot(X1, S1, 'r')
# axes[0, 1].set_title("fig2")
# # ...以此类推
# plt.tight_layout() # 自动调整子图参数，使其填充整个图像区域
# plt.show()
```

6. Seaborn 简介 (课件P46-P48)

Seaborn 是一个基于 Matplotlib 的 Python 数据可视化库，它提供了更高级的接口，用于绘制引人入胜且内容丰富的统计图形。

- **特点** (课件P46)：
    - 绘图接口更为集成，少量参数可实现复杂绘图。
    - 多数图表具有统计学含义（如分布图、关系图、回归图）。
    - 对 Pandas DataFrame 和 NumPy 数组支持友好。
    - 风格设置更为多样（预设主题、颜色配置）。
- **导入**：

```
import seaborn as sns
```

- **常用绘图函数**：
    - **关系图**: `sns.relplot()` (散点图 `scatterplot`, 折线图 `lineplot`)
    - **分布图**: `sns.displot()` (直方图 `histplot`, 核密度估计 `kdeplot`, 经验累积分布 `ecdfplot`)
    - **分类图**:
        - 分类散点图: `sns.stripplot()`, `sns.swarmplot()`
        - 分类分布图: `sns.boxplot()`, `sns.violinplot()`, `sns.boxenplot()`
        - 分类统计图: `sns.barplot()`, `sns.countplot()`, `sns.pointplot()`
    - **回归图**: `sns.lmplot()`, `sns.regplot()`
    - **矩阵图**: `sns.heatmap()` (热力图，课件P47-P48), `sns.clustermap()`
- **风格设置** (课件P48)： `sns.set_theme(style="darkgrid")` 或 `sns.set_style("whitegrid")` `sns.heatmap(data, cmap='YlGnBu')` (cmap 设置颜色映射表)

```
# 热力图示例 (课件P47)
# np.random.seed(0)
# uniform_data = np.random.rand(10, 12)
# sns.heatmap(uniform_data, cmap='viridis') # cmap可以改变颜色主题
# plt.show()
```

7. 可视化选择 (课件P34)

课件P34 提供了一个根据“你想展示什么？”来选择图表类型的决策树，这是一个很好的参考：

- **比较 (Comparison)**: 条形图, 柱状图, 折线图, 雷达图。
- **趋势 (Trend)**: 折线图, 面积图。
- **联系 (Relationship/Correlation)**: 散点图, 气泡图。
- **分布 (Distribution)**: 直方图, 散点图, 箱线图 (Seaborn)。
- **构成 (Composition)**: 饼图, 堆叠条形图/柱状图, 100%堆叠图, 面积图, 漏斗图, 金字塔图。
- **地理 (Geographical)**: 地图。

Python与大数据分析 --大数据分析-预处理

1. 大数据的概念 (课件P2, P4)

- **定义**：大数据通常指无法在可容忍的时间内用传统信息技术和软硬件工具对其进行获取、管理和处理的巨量数据集。它需要可伸缩的计算体系结构来处理。
- **数据量增长**：根据IDC报告，人类社会数据每年以50%的速度增长，每两年就翻一番，这被称为“大数据摩尔定律” (课件P4)。

2. 大数据的核心特征 (4V/5V - 必考点) 🌟 (课件P2-P3, P7)

大数据的典型特征通常用多个以“V”开头的词来描述，最核心的是“4V”，有时也会扩展到“5V”或其他。

- **Volume (大量化/数据量大)** (课件P3-P4)：
  - **含义**：指的是数据集合的规模巨大，远超传统数据处理能力。数据量可以从TB级别到PB、EB甚至ZB级别。
  - **示例**：社交媒体每天产生海量帖子和互动，科学研究（如基因组学、高能物理LHC加速器、地球与空间探测）产生庞大的数据集，企业应用中的交易记录、日志文件等。
- **Velocity (快速化/处理速度快)** (课件P3, P6)：
  - **含义**：指数据的产生速度极快，并且需要快速处理和分析以获取即时价值。数据流是持续不断的，要求系统具备实时或近乎实时的处理能力。

        - 示例：“1秒定律”——从数据生成到决策响应仅需1秒。例如，电商网站（如淘宝）每秒处理大量交易和用户行为数据，社交媒体（如新浪微博）信息流的快速更新。
    - **Variety (多样化/数据类型繁多)** (课件P3, P5)：
        - 含义：大数据来源广泛，类型多样，远不止传统的关系型数据库中的结构化数据。
        - 类型：
            - **结构化数据 (Structured Data)**：如数据库中的表格数据。
            - **半结构化数据 (Semi-structured Data)**：如XML、JSON文档，带有一定的结构标记。
            - **非结构化数据 (Unstructured Data)**：如文本（Email、文档、社交媒体帖子、博客）、图像、音频、视频、日志文件、传感器数据等。这是大数据中最主要的部分。
        - 示例：Web 1.0数据、Web 2.0数据（Twitter, Blog, SNS, Wiki）、基因组数据、应用日志、查询日志等。
    - **Value (价值密度低/价值)** (课件P3, P7)：
        - 含义：尽管数据总量巨大，但有价值的信息可能只占其中一小部分。需要通过有效的分析手段从海量数据中提炼出有价值的洞察。
        - 挑战：如何从低价值密度的数据中挖掘出高价值的信息是大数据分析的核心挑战之一。
        - 示例：监控视频数据量庞大，但可能只有少数几帧包含关键事件信息。
    - **Veracity (真实性/准确性)** (课件P3 - 图示中提及)：
        - 含义：指数据的准确性、可信度和质量。大数据中可能包含不确定、不精确、有偏差甚至错误的数据。
        - 挑战：确保数据的真实性和质量是进行有效分析的前提。需要数据清洗和验证过程。

(除了上述4V/5V，有时还会提及 Variability - 易变性，指数据含义随时间或上下文变化的特性；以及 Complexity - 复杂性，指数据来源多、关联复杂等。但课件主要强调的是前述几个V。)

4. 大数据分析的流程 (课件P11)

一个典型的大数据分析流程可以概括为：

1. 数据收集 **(Data Collection)**
2. 数据处理 **(Data Processing / Preprocessing)**

3. 数据分析 **(Data Analysis)**

4. **(数据可视化与结果呈现)** (虽然课件P11未直接列出，但通常是重要一环)

6. 数据处理/预处理 (课件P43-P58)

数据处理（或预处理）是大数据分析中至关重要的一步，旨在提高数据质量，使其适用于后续的分析和建模。

- 主要任务 (课件P43, P47)：
    - 数据清洗 **(Data Cleaning)**：处理“脏数据”，包括：
        - 缺失值 **(Missing Values)** (课件P48-P52)：
            - 产生原因：信息无法获取、遗漏、属性不存在等。
            - 处理方法：赋默认值（如空字符串、均值、中位数）、删除含缺失值的行或列。
        - 异常值 **(Outliers)** (课件P56-P58)：明显偏离其余观测值的数值。
            - 检测方法：简单统计量分析（最大/最小值）、3σ原则（适用于正态分布）、箱型图分析。
        - 不一致的值 **(Inconsistent Values)**：如单位不统一、格式错误。
        - 重复数据 **(Duplicate Data)** (课件P55)：使用如Pandas的`drop_duplicates()`处理。
        - 含有特殊符号的数据。
    - 数据转换 **(Data Transformation)** (课件P43, P53-P54)：
        - 规范化数据类型（如将字符串数字转为数值类型）。
        - 标准化/归一化。
        - 编码分类变量。
        - 必要的文本处理（如大小写统一、去除多余空格）。
    - 数据集成 **(Data Integration)** (课件P43)：合并来自不同数据源的数据，解决冲突，创建统一视图。
    - **(数据规约 Data Reduction)**：降低数据维度或数量，如特征选择、主成分分析(PCA)等（课件未在此部分详述，但属于预处理范畴）。
- 数据质量分析 (课件P47)：检查原始数据中是否存在脏数据是预处理的首要步骤。

7. 数据分析流程 (后续步骤概览) (课件P59)

数据预处理之后，通常会进入更深入的数据分析阶段：

- **数据探索分析 (Exploratory Data Analysis - EDA)**：通过统计摘要、可视化图表了解数据特性、分布和模式。
- **特征工程 (Feature Engineering)**：选择、构建或转换特征以增强模型表现。
- **模型选择与评估 (Model Selection and Evaluation)**：对比多种机器学习算法，选择最适合任务的模型，并进行评估。

Python与大数据分析 --线性回归

一、线性回归原理

1. 概述

线性回归是一种统计方法，用于确定两个或多个变量之间的线性关系。它通过找出一条**最佳拟合直线**来描述因变量（Y）如何随一个或多个自变量（X）的变化而变化。

- **简单线性回归公式：** $Y=a+b\cdot X$
    - Y: 因变量的预测值
    - a: 回归线在Y轴上的截距（当X=0时，Y的期望值）
    - b: 回归系数（X每增加一个单位，Y预期的平均变化量）
    - X: 自变量
- **多元线性回归公式：** $y=\beta0+\beta1x1+\beta2x2+\cdots+\beta pxp+\epsilon$
    - 扩展到多个自变量，βi 为对应自变量的系数，ϵ 为误差项。
- **矩阵表示法：** $Y=X\beta+\epsilon$

2. 核心假设

为确保线性回归的有效性和结果可靠性，需满足以下假设：

- **线性关系：** 自变量和因变量之间存在线性关系。
- **独立性：** 观测值之间相互独立。

- **同方差性 (Homoscedasticity)：** 误差项的方差在所有自变量水平上是相同的。
- **正态性：** 误差项 ϵ 服从均值为0、标准差固定的正态分布。
- **无多重共线性：** 在多元回归中，自变量间不存在严格的线性关系。

3. 最小二乘法 (Least Squares Method)

- **定义：** 一种通过最小化预测值与实际值之间的**残差平方和 (RSS)** 来确定模型参数的最优解的数学方法。
- **核心思想：** 找到一组参数，使得模型预测结果与真实数据之间的误差尽可能小。
- **残差平方和 (RSS)：** $RSS=\sum_{i=1}^{n}(y_i-(\beta_0+\beta_1 x_i))^2$
- **求解：** 通过对RSS求偏导并令其为0，解得 β_0 和 β_1 的最优解。
    - $\beta_1=\frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=1}^{n}(x_i-\bar{x})^2}$
    - $\beta_0=\bar{y}-\beta_1\bar{x}$
- **多元线性回归的矩阵解：** $\beta=(X^TX)^{-1}X^TY$

二、线性回归的实现 (基于Sklearn)

1. Sklearn 简介

- **定义：** Scikit-learn (Sklearn) 是一个广泛使用的Python机器学习库，提供简单高效的工具。
- **特点：** 易于使用、开源、文档丰富、社区支持。
- **常用模块：** 分类、回归、聚类、降维、模型选择、预处理等。

2. 核心函数

- **导入模型：** `from sklearn.linear_model import LinearRegression`
- **创建模型实例：** `model = LinearRegression()`
- **训练模型：** `model.fit(X, y)`
- **进行预测：** `model.predict(X_test)`
- **获取系数和截距：** `model.coef_`, `model.intercept_`
- **数据划分：** `from sklearn.model_selection import train_test_split`

        - `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)`
        - 参数： `*arrays` (输入数据), `test_size` (测试集比例), `train_size` (训练集比例), `random_state` (随机种子), `shuffle` (是否打乱), `stratify` (分层采样)。
- 数据标准化： `from sklearn.preprocessing import StandardScaler`
    - `scaler = StandardScaler()`
    - `X_scaled = scaler.fit_transform(X)`
    - 标准化公式： $z=\sigma x-\mu$

三、模型的评估指标

1. 回归任务常用指标

- **MSE (均方误差)**： $MSE=n1\sum i=1n(yi-y^i)2$
    - 优点： 对大误差惩罚大，数学性质好。
    - 缺点： 对异常值敏感。
- **RMSE (均方根误差)**： $RMSE=MSE$
    - 优点： 与因变量量纲一致，更直观。
    - 缺点： 同样对异常值敏感。
- **MAE (平均绝对误差)**： $MAE=n1\sum i=1n|yi-y^i|$
    - 优点： 对异常值鲁棒性强。
    - 缺点： 对大误差惩罚较弱。
- **R^2系数 (决定系数)**： $R2=1-TSSRSS$ (TSS为总平方和)
    - 优点： 反映模型拟合度，越接近1越好。
    - 缺点： 增加不相关特征也可能提高R^2。
- **调整R^2 (Adjusted R^2):** 解决了R^2因特征数量膨胀的问题。
    - 优点： 考虑样本量和特征数量影响，更适合多特征模型。
    - 缺点： 解释性较弱，需配合其他指标。

2. 其他常用评估方法

- **残差分析：** 检查残差是否符合正态分布，判断模型拟合情况。
- **K折交叉验证 (K-Fold Cross-Validation)：** `from sklearn.model_selection import cross_val_score`
    - 将数据集分为K个子集，轮流作为验证集，其余作为训练集，取平均结果。
    - **优点：** 避免数据划分导致的过拟合，尤其适合小数据集。
- **学习曲线 (Learning Curve)：** 诊断模型过拟合或欠拟合。
    - **过拟合：** 训练误差低，验证误差高，且不随样本增加而下降。
    - **欠拟合：** 训练误差和验证误差都高，且不随样本增加而下降。
- **模型复杂度 vs 泛化误差：** 通过图表选择合适的模型复杂度。
- **早停法 (Early Stopping)：** 训练过程中监测验证集性能，当不再改进时停止训练，防止过拟合。

四、Python库应用

- **数据处理：** `pandas`, `numpy`
- **模型构建：** `sklearn.linear_model` (如 `LinearRegression`)
- **数据预处理：** `sklearn.preprocessing` (如 `StandardScaler`)
- **模型选择与评估：** `sklearn.model_selection` (如 `train_test_split`, `cross_val_score`), `sklearn.metrics` (用于计算MSE, MAE等)
- **可视化：** `seaborn`, `matplotlib.pyplot`

五.6. 模型评估基础：欠拟合、过拟合与模型复杂度 (参考“第十节.pdf” P44-P49)

在机器学习模型训练过程中，理解和处理欠拟合与过拟合问题，以及模型复杂度的影响至关重要。

- **基本概念：**
    - **训练误差 (Training Error)：**模型在训练数据集上的误差。
    - **验证误差 (Validation Error) / 测试误差 (Test Error)：**模型在未见过的新数据（验证集或测试集）上的误差，也称为泛化误差。我们更关心这个误差，因为它反映了模型的泛化能力。

- 欠拟合 **(Underfitting)** (参考课件“第十节.pdf” P44, P46)：
    - **现象**：模型在训练集和验证集（测试集）上的误差都比较高。这意味着模型过于简单，未能充分学习到数据中的基本模式和规律。
    - **原因**：
        - 模型复杂度过低（例如，线性模型试图拟合非线性数据，或者决策树深度太浅）。
        - 特征数量过少，未能包含足够的信息。
        - 训练不充分（例如，迭代次数过少）。
    - **学习曲线表现** (参考课件“第十节.pdf” P44-P45)：训练误差和验证误差都很高，并且随着训练样本数量的增加，两者可能都变化不大或缓慢下降，但始终维持在较高水平。两条曲线可能靠得很近。
    - **如何解决**：
        - 增加模型复杂度（例如，使用更复杂的模型，增加网络层数或神经元数量，增加多项式特征）。
        - 添加更多有用的特征。
        - 减少正则化强度。
        - 训练更长时间（如果是因为训练不充分）。
- 过拟合 **(Overfitting)** (参考课件“第十节.pdf” P44, P46)：
    - **现象**：模型在训练集上表现非常好（训练误差很低），但在验证集（测试集）上表现较差（验证误差很高）。这意味着模型过度学习了训练数据中的噪声和特定细节，导致其泛化能力下降，无法很好地适应新数据。
    - **原因**：
        - 模型复杂度过高（例如，决策树深度过深，神经网络参数过多）。
        - 特征数量过多，特别是存在噪声特征或不相关特征。
        - 训练数据量相对于模型复杂度来说太少。
    - **学习曲线表现** (参考课件“第十节.pdf” P44-P45)：训练误差很低并持续下降，而验证误差在某个点之后开始上升或维持在较高水平，训练误差和验证误差之间存在较大差距。
    - **如何解决**：
        - **获取更多训练数据**：这是最有效的方法之一，但往往成本较高。
        - **降低模型复杂度**：使用更简单的模型，减少特征数量（特征选择或降维如PCA），减少网络层数或神经元数量，剪枝决策树等。

        - **正则化 (Regularization)**：在模型的损失函数中加入惩罚项（如L1、L2正则化），限制模型参数的大小，防止模型过于复杂。
        - **Dropout** (主要用于神经网络)：在训练过程中随机丢弃一部分神经元，减少神经元之间的共同适应。
        - **早停法 (Early Stopping)** (参考课件“第十节.pdf” P48-P49)：在训练过程中监控验证集上的性能，当验证集性能不再提升甚至开始下降时，提前停止训练，以防止模型在训练集上过度拟合。
        - **数据增强 (Data Augmentation)**：通过对现有训练数据进行变换（如图像旋转、裁剪）来生成更多训练样本。
        - **交叉验证 (Cross-Validation)** (参考课件“第十节.pdf” P42-P43)：更可靠地评估模型性能，帮助选择合适的模型和超参数。
- **模型复杂度与泛化误差的关系** (参考课件“第十节.pdf” P46-P47)：
    - 通常存在一个“最佳”的模型复杂度点。
    - **模型过于简单**：容易导致欠拟合，训练误差和验证误差都较高。
    - **模型过于复杂**：容易导致过拟合，训练误差低，但验证误差高。
    - **目标**：找到一个平衡点，使得模型在验证集（测试集）上的误差最小，即具有良好的泛化能力。
    - **诊断工具**：可以通过绘制模型复杂度（例如，决策树的深度，多项式回归的阶数，SVM的C值）与训练误差、验证误差的关系图（验证曲线）来观察这种关系，帮助选择合适的模型复杂度。

Python与大数据分析 --聚类

一、聚类概述

1. 定义

聚类是一种**无监督学习**技术，旨在将数据集划分为多个“簇”（clusters），使得：

- **簇内相似度高**： 同一簇内的元素彼此相似。
- **簇间相似度低**： 不同簇之间的元素差异较大。

形式化地，给定样本集 D=x1,x2,...,xm 包含 m 个无标记样本，每个样本 xi=xi1xi2,...,xin 是一个 n 维特征向量。聚类算法将样本集划分为 k 个不相交的簇 Cl∣l=1,2,...,k，其中 Cl∩Cl′\=l=ϕ 且 D=Ul=1kCl。

2. 目标与重要性

- **发现潜在结构：** 在无标签数据中揭示内在模式。
- **分组相似数据：** 便于后续分析、处理或预测。
- **简化数据复杂度：** 将大量数据压缩为较少簇。
- **帮助决策：** 为市场细分、产品推荐提供洞察。
- **提高数据处理效率：** 合理分组降低计算复杂度。
- **数据预处理：** 作为其他机器学习算法的前处理步骤。

3. 应用领域

- **数据分析：** 探索性分析，如客户数据分析、社交网络分析。
- **图像识别：** 图像分割（人脸识别、物体识别）。
- **市场细分：** 根据消费者行为划分客户群体。
- **生物信息学：** 基因表达数据分析，疾病模式识别。
- **文本挖掘：** 文档聚类（信息检索、推荐系统）。

二、聚类算法介绍与比较

聚类算法有多种类型，每种方法都有不同的优缺点和适用场景。

1. K-Means 算法 (K-Means) - 重点

- **原理：** 一种**基于划分**的聚类算法，通过最小化簇内数据点到簇中心的距离（平方误差）进行聚类。
    - **目标：** 最小化平方误差 E=∑i=1k∑x∈Ci∣x−μi∣22
        - μi 是簇 Ci 的均值向量（质心）。
    - **迭代过程：** 随机选择初始质心 → 将样本分配到最近的质心 → 重新计算簇的质心 → 重复直到质心不再变化。
- **特点：** 简单、高效、易于理解。
- **优点：** 简单，效率高，适用于大规模数据集。

- 缺点：
    - **需要预设 K 值**（簇的数量）。
    - **对初始质心敏感**，容易陷入局部最优。
    - **只能发现球形簇**（均匀密度）。
    - 对噪声和离群点敏感。
- **K-Means++ 初始化：** K-Means 的一种优化初始化方法，通过选择距离当前已选质心较远的点作为下一个质心，减少对初始化的敏感性，提高聚类质量和收敛速度。
    - **选择质心的概率：** 样本 x 被选中的概率与其到已选质心的最小距离的平方成正比。
- 核心函数 **(Sklearn)：** `sklearn.cluster.KMeans`
    - `kmeans = KMeans(n_clusters=3, init='k-means++', max_iter=300, n_init=10, random_state=0)`
    - `kmeans.fit(X)`
    - `kmeans.labels_` (获取每个点的簇标签)
    - `kmeans.cluster_centers_` (获取簇中心坐标)
    - `kmeans.inertia_` (获取内部误差平方和，即SSE)

2. 层次聚类 (Hierarchical Clustering)

- **原理：** 通过构建树形结构（树状图）来逐步合并（凝聚型，自底向上）或分割（分裂型，自顶向下）簇。
- **特点：** 不需要预设簇数（但可以指定），能展示聚类过程的层次结构。
- **优点：** 可以处理任意形状的簇，提供层次结构。
- 缺点：
    - 计算复杂度高（尤其在大数据集上，时间复杂度为 O(n2)）。
    - 对噪声和离群点敏感。
    - 一旦合并无法撤销。
- 核心函数 **(Sklearn)：** `sklearn.cluster.AgglomerativeClustering`
    - `agg_clustering = AgglomerativeClustering(n_clusters=3)`
    - `agg_clustering.fit(X)`
    - `agg_clustering.labels_`

3. DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

- **原理：** 一种**基于密度**的聚类方法，根据数据点的密度连接性来发现任意形状的簇，并能自动识别噪声数据。
- **特点：** 不需要预设 K 值，能发现任意形状的簇，对噪声鲁棒。
- **优点：** 能发现任意形状的簇，能有效识别噪声点和边界点。
- **缺点：**
    - 对参数 `eps` (邻域半径) 和 `min_samples` (最小样本数) 敏感。
    - 不适用于密度差异较大的数据集。
- **核心函数 (Sklearn)：** `sklearn.cluster.DBSCAN`
    - `dbscan = DBSCAN(eps=0.5, min_samples=5)`
    - `dbscan.fit(X)`
    - `dbscan.labels_`

4. 高斯混合模型 (Gaussian Mixture Model, GMM)

- **原理：** 一种**基于概率模型**的聚类方法，假设数据来自多个高斯分布的混合。
- **特点：** 能够处理不同密度的簇，适应性强，在高维数据中表现较好。
- **优点：** 能够处理不同密度的簇，适应性强。
- **缺点：** 对初始化和参数选择敏感，容易收敛到局部最优解。
- **核心函数 (Sklearn)：** `sklearn.mixture.GaussianMixture`
    - `gmm = GaussianMixture(n_components=3)`
    - `gmm.fit(X)`
    - `gmm.predict(X)`

5. 不同聚类算法对比总结

特性	**K-Means算法**	层次聚类	**DBSCAN**	高斯混合模型 **(GMM)**
算法类型	划分型聚类	层次聚类	基于密度的聚类	基于概率的聚类

簇形状假设	簇为球形 (均匀密度)	无假设 (可处理任意形状的簇)	簇形状不固定，能够发现任意形状的簇	簇为高斯分布 (圆形或椭圆形)
簇数	**需要预设 K 值**	不需要预设簇数	不需要预设簇数	需要预设簇数
对异常值鲁棒性	不鲁棒，对噪声和离群点敏感	不鲁棒，对噪声和离群点敏感	**对噪声和离群点鲁棒**	对噪声不太鲁棒
算法复杂度	O(nKd)	O(n2)	O(nlogn) (或 O(n2) 最坏情况)	O(nKd)
初始化要求	需要初始化 K 个簇中心 (K-Means++)	无初始化要求	无初始化要求	需要初始化 (通过 EM 算法估算高斯分布参数)
处理高维数据能力	对高维数据表现较差，可能会受到“维度灾难”影响	对高维数据表现较差，计算复杂度高	对高维数据效果较差，可能需要降维	在高维数据中表现较好，但对初始化较为敏感
适用场景	聚类数已知，数据分布均匀，簇为球形	用于探索数据的层次结构或多层次聚类	适用于有噪声数据及具有不规则簇形的数据	适用于数据集包含多个高斯分布的情况，特别是密度不同的簇
优点	简单，效率高，适用于大规模数据集	可以处理任意形状的簇，提供层次结构	能发现任意形状的簇，能有效识别噪声点和边界点	能够处理不同密度的簇，适应性强
缺点	对 K 值敏感，对噪声敏感，容易陷入局部最优	计算复杂度高，时间消耗大	对参数选择敏感 (如邻域半径)，难以处理密度差异较大的数据	对初始化和参数选择敏感，容易收敛到局部最优解
常见应用	图像压缩、市场细分、客户分群	生物学中的物种分类、图像分割、基因数据分析	异常检测、地理信息系统 (GIS)、图像处理等	图像分割、语音识别、异常检测等

三、距离计算

聚类算法通常依赖于样本之间的距离来衡量相似度。

1. 距离度量的基本性质

- 非负性： $dist(xi,xj)\geq 0$
- 同一性： $dist(xi,xj)=0$ 当且仅当 $xi=xj$
- 对称性： $dist(xi,xj)=dist(xj,xi)$
- 三角不等式： $dist(xi,xj)\leq dist(xi,xk)+dist(xk,xj)$

2. 闵可夫斯基距离 (Minkowski Distance)

给定样本 $xi=(xi1;xi2;\ldots;xin)$ 与 $xj=(xj1;xj2;\ldots;xjn)$：

$$distmk(xi,xj)=(\sum u=1n|xiu-xju|p)p1$$

其中 $p\geq 1$。

- 欧式距离 (Euclidean Distance)： 当 $p=2$ 时。
    - $disted(xi,xj)=|xi-xj|2=\sum u=1n|xiu-xju|2$
    - 特点： 计算精确。
    - 缺点： 可能受到维度灾难影响。
- 曼哈顿距离 (Manhattan Distance)： 当 $p=1$ 时。
    - $distman(xi,xj)=|xi-xj|1=\sum u=1n|xiu-xju|$
    - 特点： 计算简单，适合高维数据。

四、聚类性能度量

评估聚类结果的好坏。

1. 目标

- “簇内相似度” (intra-cluster similarity) 高
- “簇间相似度” (inter-cluster similarity) 低

2. 内部指标 (Internal Index)

直接考察聚类结果，不依赖参考模型。

- **DBI (Davies-Bouldin Index)：**
    - DBI=k1∑i=1kmaxj\=i(dcen(μi,μj)avg(Ci)+avg(Cj))
    - 值越小越好。
- **DI (Dunn Index)：**
    - DI = \min*{1 \le i \le k}* \left{ *min*{j \ne i} \left(\frac{d*{min}(C_i, C_j)} {*max*{1 \le l \le k} diam(C_l)} \right) \right}
    - 值越大越好。
- 轮廓系数 **(Silhouette Score)：** `sklearn.metrics.silhouette_score`
    - 原理： 对每个样本计算其到同簇其他样本的平均距离 a(i)（内聚度）和到最近其他簇所有样本的平均距离 b(i)（分离度）。
    - 公式： s(i)=maxa(i),b(i)b(i)−a(i)
    - 范围： s(i)∈[−1,1]，值越大表示聚类效果越好。
    - 用途： 寻找最佳 K 值（肘部法之外的另一种方法）。

3. 外部指标 (External Index)

将聚类结果与某个“参考模型”（真实标签）进行比较。

- **Jaccard 系数 (Jaccard Coefficient, JC)：** JC=a+b+ca
- **FM 指数 (Fowlkes and Mallows Index, FMI)：** FMI=a+ba·a+ca
- **Rand 指数 (Rand Index, RI)：** RI=m(m−1)2(a+d)
    - 其中 a,b,c,d 分别表示在聚类结果和参考模型中样本对的匹配情况。
    - 上述指标结果值都在 [0,1] 区间，值越大越好。

五、K-Means 实现与应用 (Sklearn)

1. 数据预处理

- 删除额外属性： `df.drop(columns, axis=1, inplace=True)`
- 处理分类变量： `status_id` 和 `status_published` 唯一标签过多，删除。

- `LabelEncoder` 转换分类变量： `from sklearn.preprocessing import LabelEncoder`
    - `le = LabelEncoder()`
    - `df['status_type'] = le.fit_transform(df['status_type'])`
- 特征缩放 (`MinMaxScaler`)： `from sklearn.preprocessing import MinMaxScaler`
    - `ms = MinMaxScaler()`
    - `X_scaled = ms.fit_transform(X)`
    - `X = pd.DataFrame(X_scaled, columns=[cols])`

2. K-Means 聚类

- 创建 **KMeans** 对象： `kmeans = KMeans(n_clusters=K, random_state=0, init='k-means++', n_init=10, max_iter=300)`
- 训练模型： `kmeans.fit(X)`
- 获取结果：
    - `kmeans.labels_` (簇标签)
    - `kmeans.cluster_centers_` (簇中心)
    - `kmeans.inertia_` (SSE)

3. 确定最佳簇数 K

- 肘部法 **(Elbow Method)**：
    - 计算不同 K 值下的 `SSE` (`kmeans.inertia_`)。
    - 绘制 `K-SSE` 曲线，寻找 SSE 下降幅度突然变缓的“肘部”拐点。
- 轮廓系数法 **(Silhouette Method)**：
    - 计算不同 K 值下的**轮廓系数均值** (`silhouette_score(X, kmeans.labels_)`)。
    - 绘制 `K-轮廓系数` 曲线，选择轮廓系数最大的 K。

Sklearn 常用模块及函数复习笔记

一、Sklearn 概述

Scikit-learn (Sklearn) 是一个基于 Python 的开源机器学习库，提供了大量用于数据挖掘和数据分析的简单高效工具。它构建在 NumPy、SciPy 和 Matplotlib 之上，是进行机器学习任务的强大平台。

- **特点：** 易于使用、高效、功能全面、文档丰富。
- **主要功能：** 分类、回归、聚类、降维、模型选择、预处理。

二、常用模块及核心函数

1. 数据预处理 (Preprocessing)

用于数据清洗、转换和特征工程。

- `sklearn.preprocessing`:
    - `StandardScaler()`: 标准化数据（均值0，方差1）。
        - `scaler.fit_transform(X)`
        - 公式： $z=\sigma x-\mu$
    - `MinMaxScaler()`: 归一化数据到指定范围（通常是 [0,1]）。
        - `scaler.fit_transform(X)`
        - 公式： $Xnorm=Xmax-XminX-Xmin$
    - `OneHotEncoder()`: 独热编码分类特征。
        - `encoder.fit_transform(categorical_data)`
    - `LabelEncoder()`: 标签编码目标变量或分类特征（将类别映射为整数）。
        - `encoder.fit_transform(labels)`
    - `PolynomialFeatures()`: 生成多项式特征，用于捕捉非线性关系。
        - `poly = PolynomialFeatures(degree=2)`
        - `X_poly = poly.fit_transform(X)`
- `sklearn.impute.SimpleImputer()`: 填充缺失值（均值、中位数、众数等）。

- `imputer = SimpleImputer(strategy='mean')`
- `X_imputed = imputer.fit_transform(X)`

2. 数据划分与模型选择 (Model Selection)

用于数据集的划分、交叉验证和模型参数调优。

- `sklearn.model_selection`:
  - `train_test_split()`: **划分数据集**为训练集和测试集。
    - `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)`
  - `cross_val_score()`: 进行**K折交叉验证**，评估模型性能。
    - `scores = cross_val_score(model, X, y, cv=5)`
  - `GridSearchCV()`: **网格搜索**，用于自动调优模型超参数。
    - `grid_search = GridSearchCV(estimator, param_grid, cv=5)`
    - `grid_search.fit(X, y)`
    - `grid_search.best_params_`, `grid_search.best_score_`

3. 回归模型 (Regression Models) - 重点

用于预测连续值的模型。

- `sklearn.linear_model`:
  - `LinearRegression()`: **普通最小二乘线性回归**。
    - **特点：** 简单、可解释性强、计算快。
    - **缺点：** 对异常值敏感，要求数据呈线性关系。
  - `Ridge()`: **岭回归**（L2正则化），用于处理多重共线性和过拟合。
    - **特点：** 惩罚大系数，使模型更稳定。
    - **缺点：** 会将系数缩小到接近零，但不会完全为零。
  - `Lasso()`: **Lasso回归**（L1正则化），用于特征选择和处理过拟合。
    - **特点：** 可以使部分系数**完全为零**，实现特征选择。

        - **缺点：** 对共线性特征的处理可能不稳定。
    - `ElasticNet()`: **弹性网络回归**（L1和L2正则化结合）。
        - **特点：** 结合Lasso和Ridge的优点，既能特征选择又能处理共线性。
- `sklearn.tree.DecisionTreeRegressor()`: **决策树回归**。
    - **特点：** 能处理非线性关系，可解释性较好。
    - **缺点：** 容易过拟合，对数据波动敏感。
- `sklearn.ensemble.RandomForestRegressor()`: **随机森林回归**。
    - **特点：** 集成学习，多棵决策树的平均结果，鲁棒性强，不易过拟合，能处理高维数据和非线性。
    - **缺点：** 模型解释性不如单棵决策树直观。
- `sklearn.svm.SVR()`: **支持向量回归**。
    - **特点：** 通过核函数处理非线性，对异常值不敏感（通过容忍带）。
    - **缺点：** 大数据集上计算量大，参数调优复杂。

4. 聚类模型 (Clustering Models) - 重点

用于发现数据中内在结构或分组（无监督学习）。

- `sklearn.cluster`:
    - `KMeans()`: **K均值聚类**。
        - **原理：** 将数据点分配到最近的K个质心，并迭代更新质心位置。
        - **特点：** 简单、高效、易于理解。
        - **缺点：** 需要预设K值，对初始质心和异常值敏感，只能发现球形簇。
        - `kmeans = KMeans(n_clusters=3, random_state=0)`
        - `kmeans.fit(X)`
        - `kmeans.labels_` (获取每个点的簇标签)
    - `DBSCAN()`: **基于密度的空间聚类**。
        - **原理：** 根据数据点的密度连接性来发现任意形状的簇，能识别噪声点。
        - **特点：** 不需要预设K值，能发现任意形状的簇，对噪声鲁棒。

        - **缺点：** 对参数 `eps` (邻域半径) 和 `min_samples` (最小样本数) 敏感，不适用于密度差异大的数据集。
        - `dbscan = DBSCAN(eps=0.5, min_samples=5)`
        - `dbscan.fit(X)`
        - `dbscan.labels_`
    - `AgglomerativeClustering()`: 层次聚类（凝聚型）。
        - **原理：** 从每个数据点为一个簇开始，逐步合并最近的簇，直到达到指定数量的簇或满足停止条件。
        - **特点：** 不需要预设K值（但可以指定），可以生成树状图（dendrogram）展示聚类过程，能发现不同尺度的簇。
        - **缺点：** 计算复杂度高（尤其在大数据集上），对噪声和异常值敏感，一旦合并无法撤销。
        - `agg_clustering = AgglomerativeClustering(n_clusters=3)`
        - `agg_clustering.fit(X)`
        - `agg_clustering.labels_`

5. 分类模型 (Classification Models) - 常用

用于预测离散类别的模型。

- `sklearn.linear_model.LogisticRegression()`: 逻辑回归。
    - **特点：** 线性分类器，输出概率，简单高效。
- `sklearn.svm.SVC()`: 支持向量分类。
    - **特点：** 通过核函数处理非线性，寻找最优超平面。
- `sklearn.tree.DecisionTreeClassifier()`: 决策树分类。
    - **特点：** 树状结构，易于理解。
- `sklearn.ensemble.RandomForestClassifier()`: 随机森林分类。
    - **特点：** 集成多棵决策树，鲁棒性好，不易过拟合。

6. 模型评估 (Metrics)

用于衡量模型性能。

- `sklearn.metrics`:
    - 回归： `mean_squared_error`, `mean_absolute_error`, `r2_score`。
    - 分类： `accuracy_score`, `precision_score`, `recall_score`, `f1_score`, `roc_auc_score`, `confusion_matrix`。
    - 聚类： `silhouette_score` (轮廓系数，衡量聚类效果好坏)。

三、总结

`sklearn` 是一个功能强大且易于使用的机器学习库，通过其模块化的设计，可以高效地完成从数据预处理到模型训练、评估和调优的整个机器学习流程。熟练掌握其核心模块和函数是进行数据科学项目的基础。

1. 维度灾难与降维概述 (课件P2-P6)

- **维度灾难 (Curse of Dimensionality)** (课件P3-P4)：
    - **定义**：当数据的维度（即特征数量）非常高时，数据分析和机器学习任务会遇到一系列问题，这种现象被称为维度灾难。
    - **影响**：
        i. **距离计算失效**：在高维空间中，数据点之间的距离趋于相似，使得基于距离的算法（如K近邻）效果下降。
        ii. **数据稀疏性**：维度越高，数据点在空间中分布越稀疏，难以进行有效的统计学习（如密度估计、聚类）。
        iii. **计算复杂度增加**：许多算法的计算量随维度增加呈指数级增长，导致训练时间过长，需要更多存储。
        iv. **过拟合风险增加**：高维数据更容易导致模型过度学习训练数据中的噪声和细节，从而在新数据上表现不佳（泛化能力差）。需要更多数据来缓解，但高质量数据往往难以获取。
    - **如何解决维度灾难？** (课件P3)
        - **降维 (Dimensionality Reduction)**：如 PCA、LDA、t-SNE 等。

        - **特征选择 (Feature Selection)**：选择最重要的特征，去除冗余或不相关的特征。
        - **正则化方法 (Regularization)**：在模型训练中加入惩罚项，避免过拟合。
- **降维 (Dimensionality Reduction)** (课件P5-P6)：
    - **定义**：将高维数据映射到低维空间的过程，目标是尽可能保留原始数据中的重要信息。
    - **主要目标**：
        - 减少计算复杂度，降低存储和计算成本。
        - 去除冗余信息和噪声，提高模型的泛化能力。
        - 避免维度灾难。
        - 便于数据可视化（将高维数据降至2D或3D）。
    - **常见方法分类** (课件P6)：
        i. **特征选择**：直接从原始特征中挑选子集。
            - 过滤法 (Filter)
            - 包装法 (Wrapper)
            - 嵌入法 (Embedded)
        ii. **特征提取 (Feature Extraction)**：通过对原始特征进行变换，生成一组新的、数量更少的特征。PCA 属于此类。
            - **PCA (主成分分析)**：通过线性变换找到数据方差最大的方向。
            - LDA (线性判别分析)：适用于分类问题，旨在最大化类间差异。
            - t-SNE (t-分布随机邻域嵌入)：常用于高维数据的可视化。
            - UMAP (统一流形逼近与投影)：另一种高效的降维和可视化方法。

2. 主成分分析 (PCA) 概述 (课件P7-P12)

- **定义与目的** (课件P7)：
    - PCA 是一种广泛应用的无监督学习方法，主要用于降维、特征提取和数据压缩。
    - 其核心目的是在**尽可能保留原始数据信息的前提下，降低数据的维度**。
    - 它通过将数据投影到方差最大的方向（即主成分）来实现这一目标。

- **为什么需要PCA？/ PCA的应用场景** (课件P7-P8)：
    - 原始数据维度过高，影响分析效率和模型性能。
    - 数据冗余，存在高度相关的特征。PCA 可以消除这种相关性。
    - 数据可视化需求，将高维数据降至二维或三维进行展示。
    - 数据降噪，去除方差较小（信息量少）的维度，保留主要变化。
    - 图像压缩（如人脸识别中的特征脸提取）。
- **PCA 的基本思想 (不涉及数学公式)** (课件P9-P12)：
    a. **找到数据变化最大的方向**：想象数据点在空间中的分布。PCA 首先寻找一个方向（一个轴），使得数据点投影到这个方向上后，散布得最开，也就是说，数据在这个方向上的方差最大。这个方向就是第一个主成分。
        - 课件P10的例子：如果数据点主要沿x轴分布，y轴变化很小，那么x轴方向就是变化最大的方向。降维时可以考虑保留x轴信息，舍弃y轴信息。

    b. **找到与前一个方向正交且变化次大的方向**：在与第一个主成分正交（垂直）的所有方向中，找到数据方差次大的方向，作为第二个主成分。

    c. **以此类推**：继续寻找与已选主成分都正交且方差尽可能大的后续主成分，直到达到预定的维度数，或者主成分数量等于原始数据维度。

    d. **坐标轴的旋转/正交变换**：PCA的过程可以理解为对原始坐标系进行旋转，使得新的坐标轴（即主成分）能够更好地捕捉数据的变异性，并且这些新的坐标轴之间是相互正交（不相关）的。
        - 课件P11-P12的例子：如果数据点斜向分布，直接看x轴或y轴都不能很好地概括其主要变化。PCA会找到一个新的坐标系（旋转后的X轴和Y轴），使得数据在新X轴上的方差最大，在新Y轴上的方差较小。降维时就可以保留新的X轴，舍弃新的Y轴。

    e. **降维的本质**：选择方差最大的前 k 个主成分来代表原始数据，从而丢弃方差较小（信息量较少或视为噪声）的主成分，达到降低维度的目的，同时保留了数据中的主要变化信息。

3. PCA 的算法步骤 (概念层面) (课件P13-P14)

（课件P13-P14给出了包含数学符号的步骤，这里将其简化为概念描述，不涉及具体计算公式）

1. **数据预处理 - 中心化 (零均值化)**：对原始数据的每一个特征（每一列），减去该特征的均值，使得每个特征的均值为0。这是PCA的必要步骤。
2. **计算数据的“散布程度”矩阵**：计算一个能反映不同特征之间相关性以及各个特征自身变化程度的矩阵（即协方差矩阵）。

3. **找到主方向和变化程度**：通过数学方法（特征值分解或奇异值分解）从上述矩阵中找到数据变化的主要方向（特征向量/主成分方向）以及这些方向上数据变化的程度（特征值/方差大小）。
4. **选择主成分**：将找到的主方向按照其对应的数据变化程度（方差）从大到小排序。选择变化程度最大的前 k 个方向作为新的 k 个主成分。
5. **数据投影/转换**：将中心化后的原始数据投影到选定的 k 个主成分方向上，得到降维后的新数据集。新的数据集有 k 个特征（维度），这些新特征是原始特征的线性组合。

4. 如何选择降维后的维度数 (k值选择) (课件P22)

选择保留多少个主成分 (k值) 是一个重要问题。

- **方差贡献率 (Explained Variance Ratio)**：
    - 每个主成分都对应一个特征值，该特征值表示数据在该主成分方向上的方差大小。
    - 某个主成分的方差贡献率 = (该主成分的特征值) / (所有特征值的总和)。
    - 它表示该主成分解释了原始数据总方差的百分比。
- **选择方法**：
    a. **经验法则/设定阈值**：选择足够数量的主成分，使得它们的累计方差贡献率达到一个预设的阈值，例如85%-95%。这意味着降维后的数据保留了原始数据85%-95%的信息。
    b. **碎石图 (Scree Plot)**：
        - 将主成分按其对应的特征值大小降序排列。
        - 绘制一个图表，横轴是主成分的序号，纵轴是对应的特征值（或方差贡献率）。
        - 观察曲线的“拐点 (Elbow Point)”，即斜率急剧下降的点。通常选择拐点之前的主成分数量。这个点之后的主成分对应的特征值较小，贡献的方差也较小。

5. PCA 的实现

PCA可以通过多种方式实现：

- **使用 NumPy 手动实现** (课件P23-P24)： 课件中提供了一个使用NumPy实现PCA算法的函数 `pca_numpy`。其步骤与前面描述的算法步骤一致：数据标准化、计算协方差矩阵、计算特征值和特征向量、选择主成分、数据投影。 并给出了一个对随机生成的二维数据降至一维并进行可视化的例子。

- 使用 **scikit-learn** 库中的 `PCA` 类 (课件P25-P48)：

  `sklearn.decomposition.PCA` 是一个非常方便易用的PCA实现。

  - 导入：`from sklearn.decomposition import PCA`
  - 主要参数：
    - `n_components`: 指定降维后的维度数。
      - 可以是整数：直接指定保留的主成分个数。
      - 可以是 `0.0` 到 `1.0` 之间的浮点数：表示希望保留的方差百分比，PCA会自动选择相应数量的主成分。
      - 可以是 `'mle'`：使用Minka的MLE算法自动选择维度。
      - 如果为 `None`：保留所有主成分，`n_components == min(n_samples, n_features)`。
    - `whiten`: 布尔值，默认为 `False`。如果为 `True`，会对降维后的数据进行白化处理，使得每个主成分的方差为1。
    - `svd_solver`: 奇异值分解 (SVD) 的求解器选择，如 'auto', 'full', 'arpack', 'randomized'。
  - 常用属性：
    - `explained_variance_ratio_`: 一个数组，表示每个选定主成分的方差贡献率。
    - `components_`: 主成分方向（特征向量）。
  - 常用方法：
    - `fit(X)`: 用数据 X 训练PCA模型（计算主成分）。
    - `transform(X)`: 将数据 X 投影到已训练的主成分空间。
    - `fit_transform(X)`: 先用 X 训练模型，然后对 X 进行降维。

  案例应用 (课件P26-P48)：

  a. 鸢尾花数据集 **(Iris Dataset)** (课件P26-P32)：
    - 原始数据有4个特征。
    - 通过PCA降至2维，可以方便地进行可视化，观察不同类别鸢尾花在二维主成分空间中的分布。
    - 课件还展示了PCA降维后，使用决策树分类器的准确率有所提升的例子，说明PCA有时可以改善模型性能（通过去除噪声或冗余信息）。

  b. 手写数字数据集 **(Digits Dataset)** (课件P33-P41)：
    - 原始数据是8x8像素的图像，即64个特征。

- 通过PCA降至2维进行可视化，可以看到不同数字的聚类情况。
- 与t-SNE降维方法进行了对比，t-SNE通常能更好地展现非线性结构，但计算成本更高。
- 通过绘制累计方差贡献率图，可以选择保留例如90%方差的主成分数量（示例中约为21个）。

c. 图像压缩 **(Image Compression)** (课件P42-P48)：

- 将图像（彩色或灰度）的像素数据视为高维数据。
- 对图像的每个颜色通道（或灰度图像本身）应用PCA，选择少量主成分来重构图像，从而达到压缩的目的。
- 通过改变保留主成分的数量 (k值)，可以看到不同压缩程度下图像质量的变化。k值越小，压缩率越高，但图像失真也越大。