第 3 章

Python基本语法

3

编程语言的逻辑规则既决定了表达的边界，也塑造了思维的路径。作为现代编程体系中最具表达力与灵活性的语言之一，Python在语法设计上兼顾了简洁性与结构性，使其成为人工智能、数据分析、自动化运维等领域的首选语言。

深入理解Python的语法基础，不仅是掌握AI辅助开发的前提，更是实现高效协作与构建高可读性代码的关键。从变量定义到流程控制，从函数构造到模块引用，每一项语言特性都关系着编程意图的清晰传达。熟练掌握这些规则，将为后续的智能化编程奠定坚实的语言根基。

3.1 数据类型与变量机制

编程的本质在于对数据的表达与处理，而数据类型的划分与变量的组织方式，直接决定了程序的运行逻辑与内存模型。Python作为一种动态类型语言，在变量定义、类型转换以及内存引用等方面具有显著特征。

理解不同数据类型的行为模式，以及变量在命名空间中的生命周期管理，是构建高质量代码的基础起点。明确数据结构与变量机制的内在联系，能够为后续的函数设计、数据操作乃至复杂结构建模提供坚实的语义基础。

3.1.1 数值类型及算术运算

在Python语言中，数值类型主要包括整数、浮点数与复数，是进行数学计算与逻辑判断的基础。不同数值类型之间既可独立运算，也可通过类型转换实现混合运算。Python支持常见的算术运算符，如加、减、乘、除、幂运算与取余运算，且具有较强的表达能力。

掌握数值类型的精度特性、运算规则以及隐式转换机制，对于理解程序运行结果、避免精度陷阱以及构建精确的数值逻辑具有重要意义。下面通过具体的代码示例，展示不同数值类型及算术运算的核心用法。

【例3-1】实现整数、浮点数与复数的混合运算，演示基本算术运算符的使用效果，验证不同数据类型在表达式中的交互行为。

```
# 定义整数类型
a = 12
b = 5

# 整数运算
sum_ab = a + b              # 加法
diff_ab = a - b             # 减法
prod_ab = a * b             # 乘法
div_ab = a / b              # 浮点除法
floor_div_ab = a // b       # 整除
mod_ab = a % b              # 取余
exp_ab = a ** b             # 幂运算

# 定义浮点数类型
x = 3.5
y = 2.0

# 浮点运算
sum_xy = x + y
div_xy = x / y

# 定义复数类型
z1 = complex(2, 3)
z2 = complex(1, -1)
z_sum = z1 + z2
z_product = z1 * z2

# 输出所有结果
print("整数加法:", sum_ab)
print("整数减法:", diff_ab)
print("整数乘法:", prod_ab)
print("浮点除法:", div_ab)
print("整除结果:", floor_div_ab)
print("取余结果:", mod_ab)
print("幂运算结果:", exp_ab)
print("浮点加法:", sum_xy)
print("浮点除法:", div_xy)
print("复数加法:", z_sum)
```

```
print("复数乘法:", z_product)
```

测试结果：

```
整数加法: 17
整数减法: 7
整数乘法: 60
浮点除法: 2.4
整除结果: 2
取余结果: 2
幂运算结果: 248832
浮点加法: 5.5
浮点除法: 1.75
复数加法: (3+2j)
复数乘法: (5+1j)
```

在实际开发中，数值表达不仅用于逻辑控制与状态管理，更是模型参数、物理仿真与数据计算的核心载体。深入理解数据类型的精度范围、运算优先级以及类型转换机制，对于构建稳定、可控的程序逻辑具有重要意义。

3.1.2　字符串操作与编码解码

字符串是Python中最常用的数据类型之一，在文本处理、数据交换与自然语言处理等领域发挥着基础性作用。Python提供了丰富的字符串操作接口，支持切片、拼接、查找、替换、大小写转换等常规操作。同时，还内置了完善的编码与解码机制，可实现Unicode与字节序列之间的相互转换，为多语言环境下的数据处理提供关键支持。掌握这些基本能力，有助于构建具备健壮性与跨平台兼容性的文本处理逻辑。

【例3-2】实现字符串的基本操作与编码解码流程，演示字符串切片、替换、拼接以及Unicode编码转换的基本用法。

```
# 定义一个原始字符串
text = "Python编程是一种优雅的艺术"

# 字符串切片：取前6个字符
sliced = text[:6]

# 字符串替换：将“优雅”替换为“强大”
replaced = text.replace("优雅", "强大")

# 拼接：添加后缀
extended = text + "，适合AI时代"

# 大小写转换（对英文有效）
```

```
    english_text = "OpenAI is powerful"
    upper_text = english_text.upper()
    lower_text = english_text.lower()

    # 编码：将字符串编码为UTF-8字节流
    encoded = text.encode("utf-8")

    # 解码：将字节流还原为字符串
    decoded = encoded.decode("utf-8")

    # 输出结果
    print("原始字符串:", text)
    print("切片结果:", sliced)
    print("替换结果:", replaced)
    print("拼接结果:", extended)
    print("大写转换:", upper text)
    print("小写转换:", lower text)
    print("编码结果:", encoded)
    print("解码结果:", decoded)
```

测试结果：

```
    原始字符串: Python编程是一种优雅的艺术
    切片结果: Python编程
    替换结果: Python编程是一种强大的艺术
    拼接结果: Python编程是一种优雅的艺术，适合AI时代
    大写转换: OPENAI IS POWERFUL
    小写转换: openai is powerful
    编码结果:
    b'Python\xe7\xbc\x96\xe7\xa8\x8b\xe6\x98\xaf\xe4\xb8\x80\xe7\xa7\x8d\xe4
\xbc\x98\xe9\x9b\x85\xe7\x9a\x84\xe8\x89\xba\xe6\x9c\xaf'
    解码结果: Python编程是一种优雅的艺术
```

上述示例代码展示了字符串在Python中的多种操作方式，涵盖了基本的文本处理与字符编码控制。Python的字符串处理不仅直观灵活，还具备强大的跨语言兼容能力。

在处理包含中文、日文或其他非ASCII字符的文本时，合理使用编码与解码接口可以有效避免乱码与编码冲突，为多语言系统的开发提供可靠的技术保障。熟练掌握字符串处理技巧，是构建高质量文本程序的重要基础。

3.1.3　布尔逻辑与比较运算符

布尔逻辑与比较运算符构成了程序控制结构与条件判断的核心语法基础。在Python中，布尔类型包括True与False，通常由比较表达式或逻辑运算产生。

常见的比较运算符包括大于、小于、等于、不等于等，用于判断数值之间的关系；逻辑

运算符包括and、or与not，可用于构造更复杂的判断表达式。掌握布尔逻辑和比较规则，有助于建立清晰的程序分支与控制路径，是编写决策逻辑与循环控制的前提。

【例3-3】实现一组布尔逻辑与比较表达式的判定逻辑，输出多个条件组合的结果，以验证其运算特性与组合规则。

```
# 定义两个变量
x = 10
y = 20

# 比较运算
print("x 等于 y:", x == y)
print("x 不等于 y:", x != y)
print("x 小于 y:", x < y)
print("x 大于或等于 y:", x >= y)

# 布尔逻辑组合
a = True
b = False
print("a and b:", a and b)
print("a or b:", a or b)
print("not a:", not a)

# 组合表达式
print("x < y and a:", x < y and a)
print("x > y or b:", x > y or b)
```

测试结果：

```
x 等于 y: False
x 不等于 y: True
x 小于 y: True
x 大于或等于 y: False
a and b: False
a or b: True
not a: False
x < y and a: True
x > y or b: False
```

上述示例展示了Python中布尔值与比较运算的基本应用方式。通过变量之间的关系判断和逻辑组合，可以构建用于流程控制的逻辑判断体系。布尔表达式是条件语句、循环结构和异常

处理的基础，在复杂逻辑场景中尤为关键。

理解布尔值的生成机制与比较运算的语义，是开发健壮程序的重要前提，也是后续深入控制结构与条件推理不可或缺的核心知识。

3.1.4　类型转换与类型判断

在Python程序设计中，不同数据类型之间的转换与类型判断是构建动态数据处理逻辑的重要工具。类型转换主要分为显式与隐式两种：前者通过int()、float()、str()等函数实现，后者则依赖程序语句上下文自动推断。

类型判断通常通过type()与isinstance()等函数来识别变量的数据类型，从而实现逻辑分支控制或数据格式验证。掌握这些基本操作，对于处理用户输入、执行数据清洗以及实现动态类型兼容具有重要意义。

【例3-4】对多个变量进行类型转换操作，并通过type()与isinstance()函数验证其当前类型状态，演示Python中典型的动态类型处理能力。

```
# 原始变量
a = "123"
b = 3.14
c = 100

# 类型转换
a_int = int(a)
b_str = str(b)
c_float = float(c)

# 类型判断
print("a_int的类型是：", type(a_int))
print("b_str是不是字符串类型：", isinstance(b_str, str))
print("c_float是否为整数：", isinstance(c_float, int))
```

测试结果：

```
a_int的类型是： <class 'int'>
b_str是不是字符串类型： True
c_float是否为整数： False
```

类型转换与类型判断在实际开发中具有极高的实用价值，尤其是在处理外部输入或动态数据时，能够有效避免类型不一致而引发的运行错误。

Python提供了灵活的转换函数与类型判断机制，使开发者能够在不依赖强类型声明的前提下，仍可构建出类型安全且结构清晰的程序逻辑。理解并合理运用这些机制，对于构建高健壮性代码、实现跨类型兼容处理具有重要意义。

3.2　流程控制结构

程序的执行过程本质上是由一系列具有条件约束与执行节奏的控制指令组合而成的。通过分支判断、循环迭代与异常处理等流程控制结构，代码才能依据不同的输入条件进行动态演化与分流处理。

Python在流程控制方面保持了高度的语义清晰性，既强调逻辑表达的自然性，又兼顾结构控制的严谨性。掌握这些控制结构的组织方式，有助于构建具备逻辑分支、自主判断与稳定执行路径的程序框架，从而形成具备决策能力的代码骨架。

3.2.1　条件判断与多分支逻辑

在程序执行过程中，依据特定条件选择不同的执行路径，是构建具有判断能力的逻辑结构的基本方式。Python提供了if、elif与else语句，用于实现条件的单分支与多分支判断。当判断条件满足时，对应的代码块被执行；否则，程序将继续判断下一个条件，或执行默认分支。

条件表达式通常结合比较运算符与布尔逻辑使用，可灵活构建嵌套或层级化控制结构，广泛应用于输入校验、状态选择与流程分派等实际场景。

【例3-5】实现一个分数等级判断程序，根据用户输入的成绩值输出对应的等级描述，演示条件判断与多分支结构在实际决策逻辑中的应用。

```
# 学生成绩输入
score = 86

# 条件判断逻辑
if score >= 90:
    print("成绩等级：优秀")
elif score >= 80:
    print("成绩等级：良好")
elif score >= 70:
    print("成绩等级：中等")
elif score >= 60:
    print("成绩等级：及格")
else:
    print("成绩等级：不及格")
```

测试结果：

```
成绩等级：良好
```

条件判断语句是流程控制的核心工具之一，通过布尔表达式引导程序在不同执行路径之

间作出选择。Python的if-elif-else结构语义清晰、可读性强，支持嵌套组合与复合判断，适合构建多层次的逻辑分支。合理使用条件判断可显著提升程序的智能性与交互性，是编写响应性代码、处理用户输入与系统状态切换等常见问题的基本手段。

3.2.2　for 循环与 range 结构

在程序设计中，循环结构常用于处理具有重复特征的任务。Python的for循环配合range()函数，可高效地控制循环次数与迭代行为。range()函数用于生成一组等差整数序列，通常作为循环计数器使用，并支持对起始值、终止值与步长的灵活配置。该结构语义直观，尤其适用于计数、批量处理与索引访问等场景，在数据处理与流程控制中应用广泛。

【例3-6】实现一个简单的累加器，计算从1加到指定上限值的总和，展示for循环与range()函数的基本使用方法。

```
# 设置累加上限
upper_limit = 10

# 初始化总和
total = 0

# 使用for循环进行累加
for i in range(1, upper_limit + 1):
    total += i  # 累加每一个数字

# 输出最终结果
print("1到", upper_limit, "的累加和为：", total)
```

测试结果：

```
1到 10 的累加和为： 55
```

通过for循环与range()函数的组合，可以方便地实现有明确边界的循环控制。与传统的C语言风格循环相比，Python的写法更加简洁明了，无须显式定义循环变量的更新语句。range()函数的参数设置灵活，既能指定起止边界，又支持步长调整，能够满足多样化的迭代需求。熟练掌握该语法结构，有助于提升程序的可读性与控制力，在数据结构遍历、矩阵运算与算法实现中均具有重要意义。

3.2.3　while 循环与终止控制

在循环控制结构中，while语句以布尔表达式为判断条件，在条件成立的情况下持续执行特定代码块，直到条件不再满足为止。相较于for循环，while循环更适用于迭代次数不确定、需基于运行过程中动态状态判断是否继续执行的场景。通过合理设置终止条件，并结合break

语句进行中断控制，可有效防止无限循环的发生。该结构广泛应用于交互式输入验证、任务状态监听以及基于传感条件的持续执行逻辑中。

【例3-7】实现一个基于用户输入的累加系统，持续接收正整数并进行累加，当用户输入负数时终止程序，展示while循环的典型终止控制方式。

```
# 初始化总和
total = 0

# 提示用户输入
print("请输入正整数进行累加，输入负数结束：")

# 启动循环
while True:
    num = int(input("输入一个数字："))
    if num < 0:
        break        # 负数则跳出循环
    total += num     # 正数则累加

# 输出总和
print("所有输入的正整数的和为：", total)
```

测试结果：

```
请输入正整数进行累加，输入负数结束：
输入一个数字：10
输入一个数字：5
输入一个数字：20
输入一个数字：-1
所有输入的正整数的和为： 35
```

while循环提供了更为灵活的流程控制机制，适用于运行条件在执行过程中不断变化的程序逻辑。其典型应用包括监听式执行、无限轮询以及用户交互式任务等。通过配合break、continue等控制语句使用，可有效避免逻辑死循环与冗余判断。在使用while结构时，应特别注意终止条件的设计，确保逻辑完整性与程序安全性，从而提升代码的健壮性与容错能力。

3.2.4　嵌套结构与控制语句

在较为复杂的流程控制中，常常需要在一个循环或条件判断语句内部继续嵌套另一个控制结构，这种“结构中结构”的形式称为嵌套结构。嵌套结构使程序能够执行多层次、递进式的逻辑判断和操作。同时，配合break、continue等控制语句使用，可实现对嵌套层级中不同执行流程的跳转或中断控制。该模式广泛用于多条件判断、矩阵遍历以及数据分类与处理等场景，是构建多层次程序逻辑的重要手段。

【例3-8】实现一个嵌套的双层循环结构，用于输出一个简单的九九乘法表，并通过continue语句跳过特定计算，演示嵌套结构与控制语句的协同作用。

```
# 外层循环：控制行数
for i in range(1, 10):
    # 内层循环：控制列数
    for j in range(1, i + 1):
        if i == j:
            continue  # 跳过对角线元素
        print(f"{j}×{i}={i*j}", end='\t')
    print()  # 换行
```

测试结果：

```
1×2=2
1×3=3  2×3=6
1×4=4  2×4=8   3×4=12
1×5=5  2×5=10  3×5=15  4×5=20
1×6=6  2×6=12  3×6=18  4×6=24  5×6=30
1×7=7  2×7=14  3×7=21  4×7=28  5×7=35  6×7=42
1×8=8  2×8=16  3×8=24  4×8=32  5×8=40  6×8=48  7×8=56
1×9=9  2×9=18  3×9=27  4×9=36  5×9=45  6×9=54  7×9=63  8×9=72
```

嵌套结构是程序逻辑表达中的重要形式，通过在判断或循环语句内部继续嵌套其他控制结构，可以实现更加复杂的流程控制与逻辑分支。在实际应用中，应注意控制嵌套层级的数量与逻辑清晰度，避免因嵌套过深而导致代码可读性下降。配合break、continue等控制语句使用时，应明确其作用对象所处的层级位置，确保跳转行为的准确性，从而构建出高效、易维护的多层逻辑代码结构。

3.3　函数与作用域

函数不仅是代码复用的基本单位，也是实现抽象表达与逻辑封装的关键工具。在Python中，函数的定义、调用与嵌套机制决定了程序结构的模块化程度；而变量的作用域规则则直接影响命名空间的解析方式与数据生命周期的管理策略。深入掌握函数的构成语法、参数传递方式以及闭包特性，并配合对局部作用域与全局作用域的准确理解，能够有效提升代码的组织能力与调试效率，为后续复杂系统的开发提供清晰、可控的控制边界。

3.3.1　定义函数与函数注释

函数是Python编程中最基本、也是最重要的结构之一。通过函数的定义，可以将具有独立

功能的代码块进行封装，从而提升代码的复用性与结构清晰度。良好的函数设计不仅要求逻辑独立、输入明确、输出规范，还应辅以清晰、完整的函数注释，以便于后续维护与团队协作。函数注释通常用于说明函数的用途、参数含义、返回值类型以及可能出现的异常情况，是规范化软件开发过程中的重要组成部分。

【例3-9】实现一个计算圆面积的函数，包含完整的函数注释与示例调用，展示函数的定义规范、参数传入方式以及返回值的获取过程。

```
import math

def calculate_circle_area(radius):
    """
    计算圆的面积

    参数：
    radius (float)：圆的半径，必须为非负数

    返回：
    float：圆的面积，单位与半径一致
    """
    if radius < 0:
        raise ValueError("半径不能为负数")
    return math.pi * radius ** 2

# 调用函数并输出结果
r = 5
area = calculate_circle_area(r)
print(f"半径为{r}的圆的面积为：{area:.2f}")
```

测试结果：

```
半径为5的圆的面积为：78.54
```

通过对函数进行合理定义并编写规范的函数注释，可以显著提升代码的可读性与可维护性。在编写函数时，应确保函数命名具有良好的描述性，逻辑封装合理，并提供准确、翔实的文档说明，尤其在团队协作或大型项目开发中尤为重要。

函数注释应覆盖参数说明、返回值定义以及异常处理等关键信息。遵循统一的注释规范，不仅有助于开发者自身梳理实现思路，也便于他人理解和复用代码。函数式编程理念的推广，正是构建高质量代码体系的重要基础之一。

3.3.2　位置参数与关键字参数

在Python函数调用过程中，参数的传递方式直接影响函数接口的使用灵活性与代码的可读

性。位置参数按照参数定义的顺序进行传值，适用于参数较少且顺序固定的调用场景；关键字参数则通过显式指定参数名进行赋值，增加了调用的直观性与容错性，更适合参数较多或调用顺序不固定的场景。

合理区分并灵活使用这两种参数形式，有助于构建灵活、健壮的函数接口，尤其在面向框架设计或库开发时具有重要意义。

【例3-10】实现一个打印学生信息的函数，演示位置参数、关键字参数以及二者混合使用的调用方式，并通过不同示例展示其使用场景与效果。

```
def print_student_info(name, age, gender):
    """
    打印学生的基本信息

    参数：
    name (str)：学生姓名
    age (int)：学生年龄
    gender (str)：学生性别
    """
    print(f"姓名：{name}，年龄：{age}，性别：{gender}")

# 使用位置参数调用
print_student_info("张伟", 18, "男")

# 使用关键字参数调用
print_student_info(name="李娜", age=19, gender="女")

# 混合调用：位置参数在前，关键字参数在后
print_student_info("王强", gender="男", age=20)
```

测试结果：

```
姓名：张伟，年龄：18，性别：男
姓名：李娜，年龄：19，性别：女
姓名：王强，年龄：20，性别：男
```

位置参数与关键字参数各具优势：前者书写简洁、调用高效，后者表达明确、可读性强。在实际开发中，若参数数量较少且顺序确定，可优先使用位置参数；若参数较多或调用者不熟悉参数顺序时，推荐采用关键字参数以提升代码可读性。混合使用时应严格遵循“位置参数在前、关键字参数在后”的语法规则，以避免出现语法错误。

在函数接口设计过程中，应结合使用场景与调用习惯，合理选择参数传递方式，从而实现函数的易用性与健壮性的统一。

3.3.3　局部变量与 global 声明

在Python函数作用域中，变量的定义位置直接决定了其生命周期与访问权限。函数体内创建的变量默认属于局部作用域，仅在函数内部可见；若需在函数中修改外部定义的变量值，则必须使用global关键字进行显式声明。正确理解局部变量与global声明的使用边界，对于避免变量覆盖、作用域混淆等问题具有重要意义，尤其在涉及多函数协作或全局状态维护的场景中尤为关键。

【例3-11】实现一个简单的计数器函数，演示局部变量与全局变量在作用范围上的差异，并说明global关键字对外部变量的修改能力。

```
counter = 0        # 全局变量

def increase without global():
    counter = 10    # 局部变量，与全局变量同名但不影响外部
    print("局部计数器值：", counter)

def increase with global():
    global counter  # 显式声明使用全局变量
    counter += 1
    print("全局计数器值：", counter)

increase_without_global()
increase_with_global()
increase_with_global()
```

测试结果：

```
局部计数器值： 10
全局计数器值： 1
全局计数器值： 2
```

在Python中，变量的作用范围取决于其声明位置及所处的语义环境。局部变量的独立性有助于避免对外部状态的意外影响，适用于封装函数内部逻辑；而通过global关键字声明全局变量时，则需格外关注其副作用，防止不必要的状态修改导致程序行为难以预测。

在实际开发中，应优先使用局部变量以保持函数的纯粹性与可测试性，只有在确有需要修改全局状态时，才建议显式使用global声明，从而提高代码的可维护性与可控性。

3.3.4　函数式调用链与高阶函数

在Python中，函数不仅是执行指令的基本单元，同时也是一种可以被传递、返回和嵌套使用的对象。函数式调用链强调将多个函数按照特定顺序组合使用，形成流水线式的逻辑处理过

程；而高阶函数则是指能够接收函数作为参数，或将函数作为返回值的函数。

这种编程方式具有更强的抽象能力与表达灵活性，广泛应用于数据处理、事件驱动以及函数组合等场景，对于构建清晰、模块化的代码结构具有重要价值。

【例3-12】实现一个函数式数据处理链，结合高阶函数map与自定义函数的组合，对列表数据依次执行平方、过滤与求和操作。

```
def square(x):
    return x * x            # 计算平方

def is_even(x):
    return x % 2 == 0       # 判断是否为偶数

def process_data(data, func_map, func_filter):
    result = map(func_map, data)           # 应用映射函数
    result = filter(func_filter, result)   # 应用过滤函数
    return sum(result)      # 汇总结果

# 数据输入
data = [1, 2, 3, 4, 5]

# 执行函数式调用链
total = process_data(data, square, is_even)
print("结果汇总：", total)
```

测试结果：

```
结果汇总： 20
```

函数式调用链与高阶函数的结合，显著提升了代码的表达能力与可重用性。在实际开发中，借助map、filter、reduce等高阶函数，能以更简洁、直观的方式对数据流进行处理，避免传统循环结构中冗长而易出错的控制逻辑。通过合理组织函数的组合关系，使处理逻辑更清晰、模块边界更明确，是函数式编程思想的重要体现。尤其在数据转换、特征提取与逻辑重构等任务中，高阶函数为构建优雅而高效的解决方案提供了强有力的支持。

3.4　面向对象式编程

抽象现实世界中的行为与状态是程序设计中至关重要的能力。面向对象编程通过类与对象的构造机制，将数据与方法统一封装，使代码具备更强的模块性、继承性与可扩展性。

Python作为一种多范式语言，其对面向对象思想的支持体现在灵活的类定义语法、动态属

性绑定以及强大的继承与多态机制等方面。深入理解这一编程范式，有助于构建具有稳定接口、明确信息边界与高度复用能力的程序结构，从而形成系统化的逻辑建模能力。

3.4.1　类与对象的定义语法

在面向对象编程中，类是对现实世界中事物的一种抽象描述，而对象则是类的具体实例。Python作为一种动态语言，提供了简洁的类定义语法和灵活的对象操作方式。通过定义类，可以将数据和行为统一封装，从而提升代码的组织性和可维护性。

【例3-13】实现一个基本的Book类，包含属性初始化方法与信息展示方法，通过实例化两个对象展示其相关信息，演示了类与对象的基本语法。

```
# 定义一个类：表示图书
class Book:
    def __init__(self, title, author):
        self.title = title      # 图书标题
        self.author = author    # 图书作者

    def display_info(self):
        print(f"书名：《{self.title}》 作者：{self.author}")

# 创建对象实例
b1 = Book("人工智能简史", "李开复")
b2 = Book("深入理解Python", "张伟")

# 调用实例方法
b1.display_info()
b2.display_info()
```

测试结果：

```
书名：《人工智能简史》 作者：李开复
书名：《深入理解Python》 作者：张伟
```

类与对象的定义是Python面向对象编程的核心内容。通过__init__方法对对象进行初始化，可以灵活地设置实例属性。类中定义的方法可以通过对象进行调用，从而实现数据与行为的封装。掌握类的基本构造逻辑，不仅有助于组织结构清晰的程序模块，也为后续扩展功能、实现继承和多态等高级特性提供了必要的语言基础。在工程实践中，面向对象模型常用于表示业务实体、控制逻辑与系统结构，具有广泛的应用价值。

3.4.2　构造函数与属性初始化

在面向对象编程中，构造函数的主要作用是初始化新创建的对象。在Python中，构造函数

由特殊方法__init__实现，用于设定对象创建时的初始状态，包括必要的属性赋值。通过定义合适的构造函数，可以确保每个对象在创建时都具备完整的数据结构和明确的初始逻辑，有助于提高代码的健壮性与可读性。

【例3-14】实现一个Student类，通过构造函数自动接收初始化参数，将学生的姓名、年龄和成绩赋值为实例属性，并定义info()方法输出学生的完整信息。

```
# 定义一个表示学生的类
class Student:
    def __init__(self, name, age, grade):
        self.name = name    # 学生姓名
        self.age = age      # 学生年龄
        self.grade = grade  # 学生成绩

    def info(self):
        print(f"{self.name}，年龄：{self.age}，成绩：{self.grade}")

# 创建Student对象并自动初始化属性
stu1 = Student("王明", 17, 88)
stu2 = Student("李华", 18, 92)

# 调用对象方法显示信息
stu1.info()
stu2.info()
```

测试结果：

```
王明，年龄：17，成绩：88
李华，年龄：18，成绩：92
```

构造函数作为对象初始化的关键步骤，是类定义中不可忽视的重要组成部分。通过在构造函数中传入参数并赋值给实例属性，可以避免烦琐的手动赋值操作，提高代码的简洁性和规范性。

在实际项目中，合理设计构造函数不仅能够增强对象的可维护性，还可以有效防止属性未定义等逻辑错误。掌握构造函数的使用，有助于更稳健地管理对象生命周期，是面向对象编程的基础能力之一。

3.4.3　方法定义与封装访问控制

在面向对象的编程体系中，方法定义与访问控制是实现封装性的重要机制。方法是类中的函数，用于定义对象的行为逻辑；而访问控制则用于限定类属性与方法的可见范围，从而在逻辑边界内保护对象数据的完整性。在Python中，虽然并不存在强制性的访问修饰符，但可以

通过约定俗成的命名方式，实现公有（public）、保护（protected）与私有（private）级别的访问控制。

【例3-15】实现一个BankAccount类，通过公有方法控制对私有属性__balance的读写访问，展示封装机制在数据保护中的作用，避免在类外直接操作敏感数据。

```
# 定义一个银行账户类，封装账户信息
class BankAccount:
    def   init  (self, owner, balance):
        self.owner = owner              # 公有属性
        self.  balance = balance        # 私有属性，不能被类外直接访问

    def deposit(self, amount):          # 公有方法
        if amount > 0:
            self.  balance += amount

    def get_balance(self):              # 提供受控访问的接口方法
        return self.__balance

# 创建账户实例并调用方法
account = BankAccount("张三", 1000)
account.deposit(500)
print(f"当前余额：{account.get_balance()} 元")
```

测试结果：

```
当前余额：1500 元
```

通过方法定义与访问控制，可以构建更加安全、更加易于维护的程序结构。将敏感属性定义为私有成员，并通过公开方法进行数据交互，是防止误操作与非法访问的重要策略。

在Python中，虽然访问控制并不具有强制约束力，但遵循命名规范与封装原则，有助于开发结构清晰、功能明确的类定义。上述示例展示了如何将业务逻辑封装于方法内部，同时对外暴露受控接口，是掌握面向对象设计思想的关键一步。

3.4.4　类继承与方法重写机制

类继承是面向对象编程中最具代表性的特征之一，它允许新类在复用已有类结构与行为的基础上，进行功能扩展与个性化定制。方法重写机制则进一步赋予子类重新定义行为的能力，当子类重写了父类中的方法后，对应的方法调用将优先执行子类中的实现。这种机制不仅有助于构建层次清晰的类结构，也为多态性的实现提供了基础保障，是构建可扩展代码架构的重要手段。

【例3-16】实现一个Animal父类与Dog子类，通过方法重写的方式改变speak行为，使子类根据自身特性覆盖父类的默认实现，体现了继承机制与多态行为的结合。

```
# 定义父类：动物
class Animal:
    def speak(self):
        return "发出通用动物叫声"

# 定义子类：狗，继承Animal并重写speak方法
class Dog(Animal):
    def speak(self):
        return "汪汪叫"

# 创建实例并调用speak方法
a = Animal()
d = Dog()
print("动物叫声：", a.speak())
print("狗的叫声：", d.speak())
```

测试结果：

```
动物叫声： 发出通用动物叫声
狗的叫声： 汪汪叫
```

通过继承与方法重写，可以在保持程序结构清晰的同时，实现逻辑行为的差异化扩展。这种机制在大型系统中尤为重要，能够有效减少代码冗余，提升系统的可维护性与灵活性。Python支持单继承与多重继承，但在实际设计中应合理控制继承层级，并结合方法重写机制遵循开闭原则，以实现模块之间的稳定协作与良好扩展性。上述示例直观展示了如何通过继承构建功能层级、通过重写实现行为定制，是理解类设计体系的核心基础之一。

3.5　本章小结

本章系统梳理了Python语言的基础语法体系，从数据类型与变量机制入手，明确了程序设计中数据表达与内存引用的基本规则；通过流程控制结构，掌握了程序执行路径的动态调度方式；在函数与作用域部分，厘清了代码组织的逻辑单元及命名空间的解析边界；并进一步引入面向对象的编程思想，为后续构建复杂系统提供了抽象建模的语言工具。通过对本章内容的学习，可为后续AI辅助开发中的高层结构设计与自动化逻辑生成奠定坚实的语言基础。

3.6 练习题

（1）以下哪个不是Python的基本数值类型？（　　）

A. int　　B. float　　C. str　　D. bool

（2）关于字符串的切片操作，设s = "Python"，表达式s[1:4]的结果是（　　）。

A. "yth"　　B. "Pyt"　　C. "tho"　　D. "ytho"

（3）以下代码中，哪个语句能够正确判断变量x是否为整数类型？（　　）

A. x.type() == int　B. type(x) == int　C. int(x)　　D. x = int

（4）Python中，用于定义函数的关键字是（　　）

A. define　　B. func　　C. def　　D. lambda

（5）在类的定义中，用于初始化对象属性的方法名是（　　）。

A. __def__　　B. __init__　　C. setup　　D. initialize

（6）Python中，使用____可以进行整型和浮点型之间的类型转换。

（7）字符串是不可变类型，因此对其进行修改操作时，通常需要使用____方法或函数生成新的字符串。

（8）使用for循环遍历一个列表时，推荐的基本语法形式是：for item in ____。

（9）在函数内部若需要修改全局变量的值，应使用____关键字进行声明。

（10）Python中通过____机制实现类的继承与方法重写。

（11）编写一个函数，输入任意整数，返回其绝对值，要求对输入类型进行判断，并在类型不符合要求时给出提示信息。

（12）创建一个字符串变量，提取其中的前三个字符，并将其反转后打印输出。

（13）编写一个函数，接受两个参数，返回其中较大的一个，要求使用条件判断结构实现。

（14）定义一个Student类，包含name和score两个属性，并实现一个print_info方法用于输出学生信息。

（15）实现一个高阶函数，接受一个函数对象和一个列表作为参数，对列表中的每个元素应用该函数，并返回新的列表。

第 4 章

Cursor辅助生成常用模块与实用标准库

4

在实际开发过程中，标准库的使用频率往往远高于第三方框架，其稳定性、可移植性以及系统级支持，使其成为构建程序基础功能的重要依托。尤其在结合AI辅助编程工具的情境下，标准模块的生成已不再局限于文档查阅与手动构造，而是逐步转向通过语言模型实现的高效、智能调度。围绕文件读写、路径管理、日期处理、数据结构及正则表达等常规操作，AI能够精准理解开发意图，并快速构造相应模块。深入掌握标准库的使用范式，并学会借助Cursor高效调用，已成为现代Python编程中不可或缺的基础能力。

4.1 文件与路径操作

程序在运行过程中往往离不开对文件的读取与写入操作，也常常需要对文件路径进行动态构造与管理。相比于直接操作底层路径字符串，现代开发实践更强调抽象性与安全性的统一处理机制。在AI辅助的环境中，文件路径的构造、目录的遍历以及文件内容的批量处理等操作，均可通过自然语言指令一键生成，并自动兼容不同操作系统的路径格式规范。深入掌握文件与路径操作，不仅是实现数据落地与持久化的关键基础，更是提升程序健壮性与跨平台兼容能力的核心保障。

4.1.1　打开与读取文本文件

文本文件的读取操作是Python文件处理中的基础能力，也是数据处理与自动化脚本编写的起点。其重点在于如何借助提示词，将“文件路径”“读取模式”以及“内容处理逻辑”等信息传递给AI，使其能够生成语义正确、结构规范且具备容错能力的读取逻辑。掌握该能力后，可以快速实现日志分析、配置加载以及批量数据提取等常见任务。

下列实例的核心知识点涉及open()函数、文件模式（如'r'）、读取方法（如.read()、.readlines()）、文件关闭控制等，目标是利用Cursor生成一段通用且安全的文本读取逻辑。接下来，我们梳理需要向Cursor发送的提示词设计思路。

（1）明确目标：让AI编写一段能够读取指定路径文本文件并输出其内容的函数。
（2）明确上下文：是否需要说明文件路径、文件内容格式或期望的处理方式。
（3）明确格式：要求输出完整函数，包含注释、异常处理和文件关闭语句。

【例4-1】实现一个带有容错机制的文本文件读取函数，采用with open结构自动关闭文件，适配utf-8编码，并对文件不存在或IO错误时抛出的异常进行捕获。提示词如下：

```
请编写一个Python函数，接受一个文本文件路径作为参数，读取该文件的全部内容并打印到控制台。要求使用with语句自动关闭文件，并包含异常处理来防止文件未找到或读取失败。
```

Cursor生成的代码如下：

```
def read_text_file(file_path):
    """
    读取指定路径的文本文件内容并打印
    :param file_path: str，文件的路径
    """
    try:
        # 使用with语句打开文件，确保读取后自动关闭
        with open(file_path, 'r', encoding='utf-8') as file:
            content = file.read()        # 读取整个文件内容
            print(content)               # 输出内容
    except FileNotFoundError:
        print(f"错误：文件 {file_path} 不存在。")
    except IOError as e:
        print(f"读取文件时发生IO错误：{e}")
```

假设本地存在一个名为sample.txt的文件，内容如下：

```
Hello, this is a test file.
Used for demonstrating file reading in Python.
```

调用代码：

```
read_text_file('sample.txt')
```

输出结果为：

```
Hello, this is a test file.
Used for demonstrating file reading in Python.
```

如需让AI生成支持逐行处理的代码，可进一步优化提示词，例如：

```
请生成一个Python函数，逐行读取文本文件并将每行打印到控制台，读取时要求使用utf-8编码和with语句，添加必要的异常处理。
```

若需要返回内容而非直接打印，可改为：

```
请生成一个Python函数，读取指定文本文件的内容并以字符串形式返回，而非打印输出。
```

通过结构化的提示词设计，可以精确控制Cursor生成代码的格式与行为。尽管文本文件读取任务本身较为基础，但在实际项目中，它是日志分析、配置加载以及大批量数据处理的重要起点。只有在提示词中清晰表达读取模式、输入参数、输出形式与错误处理策略，才能确保Cursor返回高质量、可复用的代码结果。借助AI辅助，即使是初学者也能快速掌握标准的文件读取流程，并灵活改写以适配多种应用场景。

4.1.2　写入文件

将内容写入文件是Python文件操作中的核心能力之一，广泛应用于数据持久化、日志记录以及结果输出等场景。Python提供了多种写入模式，如覆盖写入（w）与追加写入（a）。采用AI辅助编程时，准确表达写入目标、内容来源与写入模式，是生成高质量文件输出函数的关键前提。

本节目标是创建一个能够将指定文本写入目标文件的函数。涉及的核心函数包括open()及其模式'w' 'a'，并结合write()或writelines()方法，同时强调使用with语句以避免资源泄露与异常情况下的文件未正确关闭。提示词的拆解思路如下。

（1）明确目标：生成一个函数，将输入字符串写入指定路径的文本文件中。

（2）明确上下文：需要提供写入内容与目标路径变量。

（3）明确格式：输出应为一个标准的Python函数，包含注释、异常处理以及可控的写入模式。

【例4-2】实现一个写入文本文件的标准函数，自动管理文件资源、支持utf-8编码，并具备基本的异常处理机制，适合用于日志写入或结果输出。标准化提示词如下：

```
请编写一个Python函数，接受两个参数：文件路径和字符串内容。要求将该字符串写入指定文件（使用写入模式'w'），覆盖旧内容。使用with语句管理文件，添加文件写入异常处理逻辑。
```

Cursor生成的代码如下：

```
def write_to_file(file_path, content):
    """
    将给定的内容写入指定路径的文本文件（覆盖模式）
    :param file_path: str，文件保存路径
    :param content: str，写入的文本内容
    """
    try:
        # 使用with语句打开文件并写入内容
        with open(file path, 'w', encoding='utf-8') as file:
            file.write(content)
        print("写入成功")
    except IOError as e:
        print(f"写入文件时发生错误：{e}")
```

调用示例如下：

```
write_to_file('output.txt', '本段内容将写入文件。\n支持中文和多行文本。')
```

文件output.txt将生成如下内容：

```
本段内容将写入文件。
支持中文和多行文本。
```

控制台输出为：

```
写入成功
```

若希望以追加方式写入文件，可将提示词修改为：

```
请生成一个Python函数，将传入的字符串追加写入指定文本文件（模式为'a'），并添加异常处理。
```

若需要写入列表数据，可进一步提示：

```
请生成一个Python函数，将字符串列表逐行写入文件，每个元素一行。
```

在提示词设计层面，写入文件的逻辑需要明确目标行为（覆盖或追加）、内容结构（字符串或列表）以及编码规范（utf-8是较为通用且安全的默认选择）。提示词不宜笼统描述“写文件”，而应细化具体行为细节，才能有效引导Cursor生成具备健壮性与通用性的代码逻辑。通过结构化设计提示词，可以持续提升代码生成质量，推动开发流程由手工编写转向AI驱动的高效开发流程。

4.2　时间与日期处理

时间与日期的处理贯穿于各类应用的底层逻辑之中，涵盖日志记录、任务调度、数据归

档以及时间序列分析等诸多场景。高效管理时间信息，不仅依赖对基础数据结构的理解，还需要掌握格式转换、时区处理与时间运算等实用技巧。借助AI辅助生成机制，复杂的时间格式化表达与逻辑判断可被自动化完成，从而大幅提升代码的准确性与开发效率。

4.2.1　获取当前时间与格式化

在各类应用开发中，获取系统当前时间并以符合业务需求的方式进行格式化输出是极为常见的操作。无论是日志记录、文件命名，还是时间戳生成，均需要对时间格式进行灵活掌控。Python标准库中的datetime模块提供了获取当前时间、格式化为字符串以及解析字符串为时间对象等核心能力。下列实例主要涉及模块datetime.datetime，使用的方法包括.now()与.strftime()，同时需注意时区差异、格式兼容性与边界处理等问题。提示词拆解思路如下。

（1）明确目标：生成一个函数，返回当前时间的格式化字符串表示。
（2）明确上下文：不依赖外部输入，函数内部应自动获取系统当前时间。
（3）明确格式：输出为一个包含时间格式说明的Python函数。

【例4-3】实现了一个无参函数，调用后返回当前时间的格式化字符串。该函数通过datetime.now()获取当前时间对象，再借助strftime将其转换为人类可读的形式，适用于日志输出、接口返回或命名标识等场景。提示词如下：

```
请生成一个Python函数，返回当前系统时间，格式为"YYYY-MM-DD HH:MM:SS"。使用datetime模块实现，函数应具备注释与示例。
```

Cursor生成的代码如下：

```
from datetime import datetime

def get_current_time():
    """
    获取当前系统时间，并格式化为 '年-月-日 时:分:秒' 格式的字符串
    :return: str，格式化的时间字符串
    """
    current_time = datetime.now()
    return current_time.strftime('%Y-%m-%d %H:%M:%S')
```

调用测试：

```
print(get_current_time())
```

运行结果如下：

```
2025-05-28 20:04:15
```

若需支持多种时间格式，可在提示词中增加说明，例如：

请生成一个Python函数，支持传入格式字符串参数，用以动态指定时间输出格式。

若需返回时间对象而非字符串，可提示：

生成一个函数，返回当前时间对象，要求包含详细注释。

时间获取与格式化属于高频操作，在构造提示词时应明确格式要求，以确保AI输出结果满足业务约束。建议在提示中引导生成结构清晰、注释完整的函数实现，同时关注时区处理、24小时制与前导零等细节问题。通过不断优化提示词表达，可以逐步构建一套可移植、可复用的时间工具函数库，从而提升项目整体的开发效率与代码一致性。

4.2.2　时间差与计时应用

在自动化测试、性能分析或任务超时处理等场景中，准确计算时间间隔是一项不可或缺的能力。Python通过datetime和time模块提供了多种精准的时间差计算方式，适用于事件持续时间统计、操作延时控制以及程序执行计时等应用需求。

下列实例采用time.time()进行浮点型时间戳捕捉，或通过datetime.now()获取时间点并相减得到timedelta对象，再从中提取秒数等字段，完成精准计时。提示词拆解思路如下。

（1）明确目标：生成一个函数，用于测量某一段代码执行的耗时。

（2）明确上下文：函数中应包含起始时间与结束时间的采集，不依赖外部参数。

（3）明确格式：输出为Python函数，返回值为耗时（秒数），并附带可选的打印信息。

【例4-4】实现一个可复用的计时函数，使用time.time()记录起止时间点，适用于任何需要测量运行耗时的代码块。以下以for循环作为示例任务，展示计时逻辑的灵活适配能力。提示词如下：

请生成一个Python函数，测量一段代码的执行耗时，单位为秒。使用time模块实现，函数应包括起始和终止时间的记录，并返回耗时。附带注释和使用示例。

Cursor生成的代码如下：

```
import time

def measure execution time():
    """
    测量样例代码块的执行时间，单位为秒
    :return: float，耗时秒数
    """
    start_time = time.time()

    # 示例操作：模拟耗时任务
    result = 0
    for i in range(1000000):
```

```
        result += i

    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"执行耗时：{elapsed_time:.6f} 秒")
    return elapsed_time
```

调用测试：

```
measure_execution_time()
```

运行结果如下：

```
执行耗时：0.053642 秒
```

若需支持将函数作为参数进行计时，可将提示词改写为：

```
请生成一个Python函数，接受另一个函数作为参数，测量其执行耗时。
```

若对精度要求更高（如毫秒甚至纳秒级），可引导使用time.perf_counter()替代time.time()：

```
请使用time.perf_counter()生成高精度计时函数，返回毫秒级耗时。
```

掌握时间差计算能力，是精准控制程序行为与优化执行效率的重要手段。在借助Cursor生成代码时，应明确操作的起点、终点以及耗时的表达方式。在提示词中充分描述精度需求、调用方式与模块选择，可有效提升AI生成代码的针对性与实用性。建议在实际项目中将计时逻辑封装为工具函数，以便在不同任务中统一调用并进行性能监控与分析。

4.2.3　日历计算与调度支持

在日常自动化开发中，涉及日期的计算、调度任务的安排以及对指定时间规则进行推演，都是极为常见的需求。Python标准库中的calendar与datetime模块提供了较为完整的日期处理能力，例如判断闰年、获取指定月份的日历结构、定位某一日期对应的星期信息，甚至可以作为构建调度算法时的基础时间轴逻辑。

合理使用上述工具，可以有效辅助日程安排、批量脚本调度以及系统定时任务的设计与实现。提示词的拆解思路如下。

（1）明确目标：构造一个函数，输出某年某月的日历表，并判断给定日期是星期几。

（2）明确上下文：需提供年份与月份作为函数参数，可选指定具体日期。

（3）明确格式：输出结果应包括日历文本，以及指定日期对应的星期名称。

【例 4-5】该示例函数综合利用calendar.month()获取文本格式的月历，并通过datetime.date().weekday()配合calendar.day_name完成星期名称的查询。整体结构清晰、参数设计合理，便于后续拓展为调度或日历相关应用。提示词如下：

请生成一个Python函数，输入年份与月份，返回该月的完整日历结构，并支持查询某天是星期几。使用calendar和datetime模块实现，附带注释与调用示例。

Cursor生成的代码如下：

```
import calendar
import datetime

def display calendar and weekday(year, month, day=None):
    """
    输出指定年份与月份的日历，并可选输出某日是星期几
    :param year: int 年份
    :param month: int 月份
    :param day: int 可选，具体日期
    :return: str 日历文本 + 星期信息
    """
    # 获取指定月的日历字符串
    cal = calendar.month(year, month)
    print("该月日历如下：\n", cal)

    if day:
        # 创建日期对象，获取星期信息
        date_obj = datetime.date(year, month, day)
        weekday_name = calendar.day_name[date_obj.weekday()]
        print(f"{year}年{month}月{day}日是：{weekday_name}")
        return cal, weekday_name
    return cal
```

调用测试：

```
display_calendar_and_weekday(2025, 12, 31)
```

运行结果如下：

```
该月日历如下：
   December 2025
Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

2025年12月31日是：Wednesday
```

若需支持中英文等多语言输出，可在提示词中补充说明：

```
请支持中英文输出，星期几名称支持翻译。
```

若希望函数自动获取当前年月日，可进一步提示：

```
请自动获取当前系统日期，返回当前月份日历和今日是星期几。
```

日历类操作虽然在形式上较为直观，但往往涉及多模块协同与格式精度控制，常用于定时程序、工作流引擎或日志周期管理等场景。通过Cursor自动生成此类结构化代码，可显著提高开发效率。建议在提示词中明确时间粒度、语言格式以及是否包含当前日期信息，以引导Cursor生成更具适配性与扩展性的逻辑结构。该类函数同样可用于定制型报表生成与日历可视化工具的底层支撑模块。

4.3　正则表达式基础

文本处理任务往往面临格式不一、内容冗余等挑战，传统字符串处理方法难以高效提取其中有价值的信息。正则表达式作为一种用于结构化匹配与文本解析的强大工具，在复杂数据清洗、内容校验以及批量替换等场景中扮演着关键角色。借助智能补全与提示功能，可显著降低正则表达式语法的学习门槛，从而更快速地构建符合业务需求的匹配模式。

4.3.1　模式定义与匹配方法

在实际文本处理任务中，提取结构化信息往往需要对特定文本模式进行精准识别。正则表达式（regular expression）为此提供了灵活而强大的支持，能够根据用户定义的字符规则进行复杂的文本匹配、提取与替换操作。Python内置的re模块包含构造模式、对象编译、搜索方法与匹配结果解析等完整机制。掌握这些基础匹配方法，不仅是数据清洗、日志处理和用户输入校验等任务的核心能力，也是构建自动化文本规则引擎的关键一步。

提示词拆解思路如下。

（1）明确目标：实现一个函数，使用正则表达式对输入文本进行多种方式的匹配，并输出匹配结果。

（2）明确上下文：可提供示例文本或常用匹配模式，如邮箱、手机号、IP地址等。

（3）明确格式：要求生成函数定义，包含必要注释，并能够打印匹配结果。

【例4-6】实现多个典型的正则匹配任务，包括邮箱、手机号与IP地址三种不同的匹配方式。示例分别使用match（仅匹配字符串起始位置）、search（查找任意位置的第一个匹配项）和findall（提取全部匹配项）演示不同方法之间的差异。同时引入compile编译机制，以提高多次匹配场景下的执行效率。整体结构模块化、注释清晰，便于后续扩展。提示词如下：

请生成一个Python函数，包含正则表达式基础匹配的演示内容，包括match、search、findall方法的使用，分别匹配邮箱地址、手机号与IPv4地址，要求写明注释并打印每步输出。

Cursor生成的代码如下：

```
import re

def regex_match_examples():
    """
    演示正则表达式的三种基本匹配方法：match、search、findall
    并以邮箱、手机号、IPv4地址为例进行匹配测试
    """

    ## 示例1：邮箱地址匹配
    email_pattern = r'[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+'
    email_text = "请联系邮箱：support@example.com 或 info123@data.cn 获取更多帮助。"

    # 使用search查找第一个邮箱
    email_search_result = re.search(email_pattern, email_text)
    print("邮箱search结果：", email_search_result.group() if email_search_result else "未匹配")

    # 使用findall查找所有邮箱
    email_all_matches = re.findall(email_pattern, email_text)
    print("邮箱findall结果：", email_all_matches)

    print("-" * 50)

    ## 示例2：手机号匹配（国内）
    phone_pattern = r'1[3-9]\d{9}'
    phone_text = "联系方式包括：13900001234, 15812345678。非法号码：1234567890"

    # 使用match检查是否从起始位置即为手机号
    phone_match_result = re.match(phone_pattern, phone_text)
    print("手机号match结果：", phone_match_result.group() if phone_match_result else "未匹配")

    # 使用search查找第一个合法手机号
    phone_search_result = re.search(phone_pattern, phone_text)
    print("手机号search结果：", phone_search_result.group() if phone_search_result else "未匹配")

    # 使用findall提取全部手机号
    phone_all_matches = re.findall(phone_pattern, phone_text)
    print("手机号findall结果：", phone_all_matches)
```

```
        print("-" * 50)

        ## 示例3：IPv4地址匹配
        ip_pattern = r'\b(?:\d{1,3}\.){3}\d{1,3}\b'
        ip_text = "当前网络地址包括：192.168.1.1, 10.0.0.255, 错误IP:
999.999.999.999"

        # search方式
        ip_search_result = re.search(ip_pattern, ip_text)
        print("IP search结果:", ip_search_result.group() if ip_search_result else
"未匹配")

        # findall方式
        ip_all_matches = re.findall(ip_pattern, ip_text)
        print("IP findall结果：", ip_all_matches)

        print("-" * 50)

        ## 模式编译优化（compile + match对象操作）
        email_compiled = re.compile(email_pattern)
        match = email_compiled.search(email_text)
        if match:
            print("使用compile匹配邮箱：", match.group())
            print("邮箱起始索引：", match.start())
            print("邮箱结束索引：", match.end())

        print("匹配对象span:", match.span() if match else "无")

    # 调用函数
    regex_match_examples()
```

运行结果如下：

```
    邮箱search结果： support@example.com
    邮箱findall结果： ['support@example.com', 'info123@data.cn']
    --------------------------------------------------
    手机号match结果： 未匹配
    手机号search结果： 13900001234
    手机号findall结果： ['13900001234', '15812345678']
    --------------------------------------------------
    IP search结果： 192.168.1.1
    IP findall结果： ['192.168.1.1', '10.0.0.255', '999.999.999.999']
    --------------------------------------------------
    使用compile匹配邮箱： support@example.com
    邮箱起始索引： 7
    邮箱结束索引： 26
    匹配对象span: (7, 26)
```

若需增加对非法输入的处理逻辑，可在提示词中补充说明，例如：

```
如果未匹配成功也要返回“无结果”，并写入日志记录。
```

若需支持更复杂的匹配规则，如国际手机号或更严格的邮箱校验，也可进一步提示：

```
请支持带国码的国际手机号，如+8613912345678。
```

正则表达式的学习不仅在于掌握语法符号，更在于理解不同匹配接口在实际任务中的行为差异。通过Cursor构建基础模板函数，可以快速搭建文本分析与校验的工具框架。上述示例覆盖了常见的信息抽取场景，建议读者在后续实践中结合网页爬虫、日志清洗或日志告警规则等实际项目进行演练，逐步提升正则表达式在项目工程应用中的灵活运用能力。

4.3.2　查找、替换与提取操作

在处理非结构化文本数据时，往往需要从大量内容中精准查找符合特定模式的信息，或对命中的文本进行替换与抽取。正则表达式不仅可用于模式识别，还支持基于匹配结果的内容提取（如分组提取）、内容替换（如敏感信息脱敏）以及文本拆分等操作。通过掌握re.sub、re.findall、re.split等接口，并结合分组提取机制与动态替换逻辑，可有效提升文本清洗、格式转换与内容提取等任务的处理效率与自动化水平。

提示词拆解思路如下。

（1）明确目标：使用正则表达式完成文本中多个字段的查找与替换操作，并提取指定字段。

（2）明确上下文：提供包含日期、邮箱、HTML标签等多种典型内容的文本。

（3）明确格式：输出形式为函数定义，并输出每一步的替换与提取结果。

【例4-7】该示例函数涵盖多种常见的文本处理场景，包括邮箱地址的提取、日期格式的标准化、HTML标签内容抽取、身份证号码脱敏、时间戳替换以及基于标签的文本拆分等操作。示例综合使用re.findall、re.sub、re.split等API，并配合分组与回调函数实现灵活的文本处理逻辑，整体结构清晰，示例覆盖面较广，具有较好的参考价值。提示词如下：

```
请帮我生成一个Python函数，使用正则表达式完成以下操作：
1．查找文本中的所有邮箱地址；
2．替换文本中的日期格式为统一格式（如2024-01-01）；
3．提取HTML文本中所有的<p>标签内容；
4．替换身份证号中的中间八位为星号；
要求每步操作都打印输出结果，函数代码要有详细注释，结构清晰。
```

Cursor生成的代码如下：

```
import re
```

```
    def regex_find_replace_extract():
        """
        综合演示正则表达式的查找、替换与提取操作，涵盖邮箱提取、日期标准化、HTML文本抽取、
敏感信息脱敏等典型应用场景。
        """

        # 1. 查找邮箱地址
        email_text = "注册用户请使用邮箱提交：test.user01@mail.com, 备用邮箱：
admin_123@backup.cn"
        email_pattern = r'[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+'
        email_matches = re.findall(email_pattern, email_text)
        print("邮箱地址提取结果：", email_matches)

        print("-" * 50)

        # 2. 替换日期格式为统一格式
        date_text = "事件发生在：01/23/2024 与 2024.01.25 和 2024-01-26"
        date_pattern = r'(\d{2})/(\d{2})/(\d{4})'
        # 将 MM/DD/YYYY 替换为 YYYY-MM-DD
        date_reformatted = re.sub(date_pattern, r'\3-\1-\2', date_text)
        print("日期格式替换结果：", date_reformatted)

        print("-" * 50)

        # 3. 提取<p>标签中的内容
        html_text = """
        <html>
        <body>
            <p>欢迎访问本网站</p>
            <div>其他内容</div>
            <p>本网站提供Python教程</p>
        </body>
        </html>
        """
        p_tag_pattern = r'<p>(.*?)</p>'
        p_contents = re.findall(p_tag_pattern, html_text, re.DOTALL)
        print("提取的<p>标签内容：", p_contents)

        print("-" * 50)

        # 4. 替换身份证中间八位为星号
        id_text = "用户身份证号为：110105199001018888，应做脱敏处理"
        id_pattern = r'(\d{6})(\d{8})(\d{4})'
        id_desensitized = re.sub(id_pattern, r'\1********\3', id_text)
        print("脱敏后的身份证：", id_desensitized)
```

```
    print("-" * 50)

    # 5. 提取时间戳并用函数替换为人类可读日期
    import datetime
    ts_text = "记录时间：timestamp=1706003200，另一个timestamp=1670001111"
    ts_pattern = r'timestamp=(\d{10})'

    def ts_to_date(match):
        ts = int(match.group(1))
        dt = datetime.datetime.fromtimestamp(ts)
        return dt.strftime("时间为：%Y-%m-%d %H:%M")

    ts_replaced = re.sub(ts_pattern, ts_to_date, ts_text)
    print("时间戳替换结果：", ts_replaced)

    print("-" * 50)

    # 6. 使用split切分带标签内容
    split_text = "标签：[Python] [正则表达式] [AI]"
    label_pattern = r'\[.*?\]'
    split_result = re.split(label_pattern, split_text)
    print("使用split分隔标签：", split_result)

# 调用函数
regex_find_replace_extract()
```

运行结果如下：

```
邮箱地址提取结果： ['test.user01@mail.com', 'admin_123@backup.cn']
--------------------------------------------------
日期格式替换结果： 事件发生在：2024-01-23 与 2024.01.25 和 2024-01-26
--------------------------------------------------
提取的<p>标签内容： ['欢迎访问本网站', '本网站提供Python教程']
--------------------------------------------------
脱敏后的身份证： 用户身份证号为：110105********8888，应做脱敏处理
--------------------------------------------------
时间戳替换结果： 记录时间：时间为：2024-01-23 08:26，另一个时间为：2022-12-02 15:51
--------------------------------------------------
使用split分隔标签： ['标签：', ' ', ' ', '', '']
```

可加入功能限定条件，例如：

```
只匹配以test开头的邮箱，替换内容添加日志。
```

可改为支持多种标签类型，提示词为：

```
提取所有HTML标签中的文本内容，不限于<p>。
```

针对脱敏可灵活控制：

```
如果身份证号不足18位则跳过，不替换。
```

正则表达式在查找、替换与提取文本信息方面的能力，是构建高效文本处理流程的核心工具之一。通过合理设计提示词与函数结构，并结合Cursor的快速生成与校验能力，开发者可以大幅提升数据预处理与内容识别的效率。上述示例全面展示了正则表达式在非结构化文本处理中的典型应用场景，建议读者结合日志解析、爬虫抓取或表单校验等实际项目反复练习，熟练掌握分组捕获与替换机制，为构建更复杂的文本处理系统奠定坚实基础。

4.3.3　编译正则与匹配对象分析

在复杂文本处理任务中，频繁重复使用同一正则表达式会带来不必要的性能开销。为此，Python提供了re.compile()方法，用于将正则模式预编译为Pattern对象，从而在多次匹配场景下显著提升执行效率。

此外，正则匹配所返回的Match对象不仅用于判断是否匹配成功，还包含丰富的匹配元信息，如匹配位置、分组内容以及原始输入文本的索引区间等。这些信息为后续的字段提取、结构分析与逻辑判断提供了精细化控制能力。

本节目标在于掌握如何使用re.compile()进行正则预处理，并结合match()、search()、finditer()等方法获取Match对象，深入理解其常用属性（如.group()、.span()、.start()、.end()等），并将其应用于日志解析、字段提取、结构分析等典型场景中。提示词拆解思路如下。

（1）明确目标：生成一个示例函数，展示如何通过编译正则与匹配对象提取多种字段，并分析其匹配位置与分组结构。

（2）明确上下文：提供包含邮箱、时间戳与IP地址等混合信息的示例文本。

（3）明确格式：以函数形式，逐项打印各类字段及其匹配详情。

【例4-8】该示例函数展示了使用re.compile()预编译正则表达式，并配合finditer()方法逐项遍历匹配结果、获取Match对象详细信息的完整流程。示例重点说明了如何通过.group()获取匹配内容与分组结果，通过.span()获取位置区间，.start()和.end()标记边界，从而实现精准的字段定位与文本结构分析。提示词如下：

```
请帮我写一个Python函数，使用re.compile编译正则表达式，并基于Match对象完成以下操作：
1．匹配文本中出现的邮箱地址、时间戳、IP地址；
2．对每类字段分别使用finditer()遍历所有匹配项；
3．对每个Match对象，打印group(0)、start()、end()、span()信息；
4．展示分组提取的用法，提取邮箱中的用户名部分；
5．所有代码需有清晰注释，并包含全部打印输出。
```

Cursor生成的代码如下：

```
    import re

    def analyze_matches_with_compile():
        """
        使用re.compile编译正则表达式，并利用Match对象分析邮箱、时间戳与IP地址的结构与位置。
        演示finditer遍历、分组提取、位置分析等高级匹配功能。
        """

        # 待分析的文本数据
        sample_text = """
            用户记录1：邮箱为 alice_001@example.com 登录时间为 2024-05-28 09:15:32 来自IP地址 192.168.1.5
            用户记录2：邮箱为 bob.test@sample.org 登录时间为 2024-05-27 22:05:45 来自IP地址 10.0.0.21
            用户记录3：邮箱为 charlie@company.cn 登录时间为 2024-05-25 13:49:10 来自IP地址 172.16.0.9
        """

        ## 1. 编译邮箱正则并遍历所有匹配项
        email_pattern = re.compile(r'([a-zA-Z0-9_.+-]+)@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+')
        print(">>> 邮箱地址匹配结果：")
        for match in email_pattern.finditer(sample_text):
            print("完整邮箱：", match.group(0))
            print("用户名部分：", match.group(1))
            print("位置（start, end）：", match.start(), match.end())
            print("位置span：", match.span())
            print("---")

        print("=" * 60)

        ## 2. 编译时间戳正则并输出匹配位置
        time_pattern = re.compile(r'\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}')
        print(">>> 登录时间匹配结果：")
        for match in time_pattern.finditer(sample_text):
            print("时间戳：", match.group())
            print("位置：", match.span())
            print("---")

        print("=" * 60)

        ## 3. 编译IP地址正则并提取结果
        ip_pattern = re.compile(r'(\d{1,3}\.){3}\d{1,3}')
        print(">>> IP地址匹配结果：")
        for match in ip_pattern.finditer(sample_text):
```

```
            print("匹配IP：", match.group())
            print("起始索引：", match.start())
            print("结束索引：", match.end())
            print("---")

        print("=" * 60)

        ## 4. 使用search()示例：查找首个邮箱
        first_email = email_pattern.search(sample_text)
        if first_email:
            print(">>> 首个匹配邮箱：", first_email.group(0))
            print("用户名分组：", first_email.group(1))
            print("位置span：", first_email.span())

        print("=" * 60)

        ## 5. 使用match()示例：尝试从字符串起始位置匹配
        match_result = email_pattern.match(sample_text)
        print(">>> 使用match()结果（通常None）：", match_result)

    # 调用函数
    analyze_matches_with_compile()
```

运行结果如下：

```
>>> 邮箱地址匹配结果：
完整邮箱： alice_001@example.com
用户名部分： alice_001
位置（start, end）： 16 38
位置span： (16, 38)
---
...
>>> 登录时间匹配结果：
时间戳： 2024-05-28 09:15:32
位置： (45, 64)
...
>>> IP地址匹配结果：
匹配IP： 192.168.1.5
起始索引： 75
结束索引： 87
...
>>> 首个匹配邮箱： alice_001@example.com
用户名分组： alice_001
位置span： (16, 38)
>>> 使用match()结果（通常None）： None
```

增加目标字段的标注，提升提示词的可控性，例如：

```
对每类字段使用不同颜色打印（可在交互界面实现）
```

可扩展多个分组，如提取邮箱的后缀部分：

```
捕获邮箱中的用户名和域名作为两个分组返回
```

对于大文本可要求性能优化：

```
使用预编译模式提升finditer效率，限制查找范围为前1000字符
```

编译正则表达式并深入使用Match对象，是实现高效、结构化文本解析的关键技术路径。相比直接使用re.findall()获取结果列表，基于finditer()与Match对象的方式能够提供更丰富的上下文信息与分组结构，使得正则表达式在工程实践中具备更强的逻辑控制能力与扩展空间。

4.4　系统操作与命令执行

在实际运行过程中，程序往往不仅局限于逻辑运算与数据处理，还需要与操作系统进行深度交互，以实现文件批处理、脚本自动化、运行环境检测等功能。系统操作与命令执行机制为Python程序提供了直接调用底层操作系统能力的途径，是脚本化运维、自动部署与系统管理的核心基础。通过统一的接口设计与灵活的模块封装，Python标准库能够高效调动系统资源，执行平台相关命令并获取执行结果，从而在保持代码可读性与安全性的同时，增强程序对运行环境的控制能力。

4.4.1　获取环境变量与系统信息

在进行操作系统交互编程中，获取运行环境的上下文信息对于程序的可移植性、调试效率与配置管理具有重要意义。Python通过os和platform模块，提供了访问环境变量、识别操作系统类型、查询处理器架构以及获取当前用户名等功能。

无论是部署自动化脚本、记录运行日志，还是构建跨平台工具，准确识别运行时的系统信息，都是程序执行前的重要准备步骤。

下面通过一段示例代码，演示如何获取当前系统平台、系统版本、处理器架构以及当前登录用户名，并列出部分环境变量信息。示例代码中使用platform模块采集系统级信息，使用os.environ访问环境变量字典，这些接口在部署配置和调试场景中被广泛使用。

```
import os
import platform

def print_system_info():
    # 获取当前系统平台（如Windows、Linux等）
```

```
    print("操作系统平台:", platform.system())

    # 获取系统版本信息
    print("系统版本详情:", platform.version())

    # 获取处理器架构
    print("处理器架构:", platform.machine())

    # 获取当前用户环境变量中的用户名
    print("当前用户:", os.environ.get("USERNAME") or os.environ.get("USER"))

    # 获取所有环境变量键名示例（展示前5个）
    print("部分环境变量示例:")
    for key in list(os.environ.keys())[:5]:
        print(f"  {key} = {os.environ[key]}")

print_system_info()
```

测试结果如下：

```
操作系统平台: Windows
系统版本详情: 10.0.22621
处理器架构: AMD64
当前用户: chen_zhao
部分环境变量示例:
  ALLUSERSPROFILE = C:\ProgramData
  APPDATA = C:\Users\chen_zhao\AppData\Roaming
  CHROME_CRASHPAD_PIPE_NAME = \\.\pipe\crashpad_2032_PFGEV...
  COMMONPROGRAMFILES = C:\Program Files\Common Files
  COMPUTERNAME = DESKTOP-ABC1234
```

通过Python标准库获取系统环境与平台信息的提取，不仅可以提升程序对运行环境的感知能力，也为跨平台兼容性提供了关键支持。尤其在构建安装脚本、日志自动标注、环境隔离检测等场景中，这类信息获取机制具有直接的工程实用价值。建议开发者进一步掌握platform.uname()、os.getenv()等扩展接口，在实践中逐步建立系统感知能力的开发习惯。

4.4.2　执行 Shell 命令并获取结果

在自动化工具、部署脚本或系统运维任务时，程序往往需要直接调用操作系统命令以完成底层操作。Python通过标准库中的subprocess模块，提供了对Shell命令的封装调用能力，使脚本能够安全地调用系统命令、执行批处理、获取返回结果并进行后续处理。

相较于早期的os.system()方法，subprocess模块在安全性、输入输出管理以及异常处理方面具有明显优势。其中，subprocess.run()是当前推荐的高层接口，适合大多数命令执行与结果采

集场景。

下面的示例演示了如何使用subprocess.run()执行系统命令，并根据不同平台自动选择命令（Windows使用dir，Linux/macOS使用ls -l），同时捕获标准输出与错误输出信息，实现与操作系统命令层的直接交互。

```
import subprocess

def run_shell_command(command):
    # 执行系统命令并捕获输出
    result = subprocess.run(command, shell=True, capture_output=True,
text=True)

    # 打印标准输出
    print("命令输出:")
    print(result.stdout.strip())

    # 如果有错误输出，则打印
    if result.stderr:
        print("错误信息:")
        print(result.stderr.strip())

# 示例命令：列出当前目录文件（适配Linux和Windows）
run_shell_command("dir" if subprocess.os.name == "nt" else "ls -l")
```

运行结果如下：

```
命令输出:
2025/05/28  17:23    <DIR>          .
2025/05/28  17:23    <DIR>          ..
2025/05/28  17:23               102 readme.txt
2025/05/28  17:23             20480 main.py

错误信息:
```

通过subprocess模块，Python程序能够以标准、安全的方式执行系统命令并捕获输出结果，广泛适用于批量部署、系统监控与远程执行等自动化场景。相比传统方法，该模块支持标准输出与错误输出的分离控制，更有利于调试与异常处理。在实际应用中，开发者应注意命令的跨平台差异和权限限制，并可结合shlex.split()等方法增强命令构造的安全性与可控性。

4.4.3　文件批量处理脚本生成

在操作系统环境下处理大量文件，是自动化任务中的常见需求。无论是批量重命名、内容修改，还是格式转换，均涉及对文件系统中多个对象的遍历与操作。借助Python的标准库（如os、pathlib、shutil等），可以方便地实现这些功能。而在Cursor辅助编程场景中，构建一个清

晰、可调的批处理脚本提示词，对于提升工作效率与代码质量具有重要意义。

提示词拆解思路如下。

（1）明确目标：生成一个Python脚本，支持对指定目录内所有TXT文件重新命名，并在文件末尾追加统一内容。

（2）明确上下文：需说明目录路径是否已知，是否存在既定的旧文件名规则，以及是否允许覆盖原始文件等关键约束条件。

（3）明确格式：输出应为一个完整函数或可直接运行的脚本，结构清晰，并包含必要的异常处理与注释说明。

【例4-9】实现文件批量处理脚本，提示词如下：

```
生成一个Python脚本，用于遍历指定目录下所有以.txt结尾的文件，对每个文件执行两项操作：一是将文件名加上后缀'_processed'并保存为新文件名，二是在文件内容末尾追加一行内容'Script processed by Cursor'。要求代码结构清晰，包含必要的注释和异常处理。
```

Cursor生成的代码如下：

```
import os
import shutil

def process txt files(directory path):
    """
    批量处理指定目录下的所有.txt文件：
    1. 将文件名加上' processed'后缀
    2. 在文件末尾追加特定内容
    """
    if not os.path.isdir(directory path):
        print(f"路径无效：{directory path}")
        return

    try:
        files = os.listdir(directory path)
        for file name in files:
            # 构造完整路径
            full path = os.path.join(directory path, file name)

            # 判断是否为TXT文件
            if os.path.isfile(full_path) and file_name.endswith('.txt'):
                base name = os.path.splitext(file name)[0]
                new_file_name = f"{base_name}_processed.txt"
                new full path = os.path.join(directory path, new file name)

                # 复制文件并重命名
                shutil.copy(full_path, new_full_path)

                # 追加内容
                with open(new full path, 'a', encoding='utf-8') as f:
                    f.write("\nScript processed by Cursor")
```

```
                print(f"已处理：{file name} → {new file name}")
        except Exception as e:
            print(f"处理过程中出现错误：{e}")

    # 示例调用
    if __name__ == "__main__":
        folder = "test dir"
        process_txt_files(folder)
```

Cursor返回的代码主要分为三部分处理逻辑：目录遍历、文件重命名以及内容追加。示例中使用了os和shutil模块，并结合os.path系列函数完成文件类型判断与路径拼接，具备较好的跨平台兼容性。通过try-except块捕获异常，保障了批处理过程的健壮性。同时，代码采用函数封装形式，便于后续维护与功能扩展。

运行结果如下：

```
已处理：log1.txt → log1 processed.txt
已处理：report.txt → report_processed.txt
已处理：summary.txt → summary_processed.txt
```

若需进一步扩展功能，可在提示词中加入如下条件：“要求可配置重命名规则（前缀或后缀）、追加内容可自定义，并添加日志文件以记录每一步操作”。此外，还可提示加入命令行参数解析能力，使脚本具备一定程度的可执行性与用户交互性。

上述示例展示了如何通过标准提示词生成文件批量处理脚本，并在明确需求的前提下，引导生成具备逻辑完整性与健壮性的Python代码。通过Cursor生成的结果，不仅涵盖了核心处理逻辑，还体现了良好的结构设计与可维护性。在实际应用中，这类脚本常用于数据准备、日志处理以及自动化运维等场景。只要掌握提示词拆解技巧，即可迅速构建各种定制化脚本，大幅提高开发效率。

4.5　本章小结

本章围绕Python语言中常用的标准库模块与辅助功能展开，系统介绍了文件路径处理、时间与日期管理、正则表达式匹配及系统命令执行等基础能力在编程实践中的具体应用。通过Cursor的辅助生成机制，开发者能够高效调取相关API并完成快速集成，从而大幅减少模板代码的编写负担。上述功能不仅构成了各类脚本任务的底层支撑，也在数据处理、自动化运维和信息抽取等应用场景中具有广泛适用性。掌握本章内容，有助于提升模块化编程能力与系统操作水平，为后续构建工程级项目奠定坚实基础。

4.6　练习题

（1）使用Python打开文件并进行读取操作，应使用哪种模式？（　　）

A. "w"　　B. "r"　　C. "a"　　D. "x"

（2）要将当前时间格式化为“年-月-日”形式，应使用哪个模块函数？（　　）

A. time.sleep()　　B. datetime.now()　　C. strftime()　　D. calendar.get_date()

（3）re.sub()函数的主要作用是（　　）。

A. 提取字符串　　B. 替换匹配内容

C. 校验正则格式　　D. 返回布尔值

（4）若需创建临时文件并写入数据，推荐使用哪个标准库？（　　）

A. tempfile　　B. os　　C. io　　D. shutil

（5）执行Shell命令并捕获输出结果的推荐方法是（　　）。

A. os.remove()　　B. open()　　C. subprocess.run()　　D. shutil.copy()

（6）使用with open(filename, "r") as f:语句可以安全地_____并读取文件内容。

（7）利用os.path.join()可以实现跨平台的文件路径_____操作。

（8）获取当前时间戳应使用time.time()或datetime.now()_____。

（9）正则表达式中，“\d+”用于匹配连续的_____字符。

（10）Python中可通过subprocess.run(["ls"], capture_output=True)方式执行____命令。

（11）编写一个函数，接收文件路径，统计该文件的行数并返回。要求使用with open()语句。

（12）创建一个脚本，获取当前时间并将其格式化为“YYYY-MM-DD HH:MM:SS”后打印输出。

（13）编写一个正则函数，从一段文本中提取所有Email地址，并以列表形式返回。

（14）设计一个脚本，将指定文件夹中的所有TXT文件名称写入一个log文件。

（15）编写代码，执行Shell命令ls或dir（根据系统自动判断），并将执行结果打印出来。