CA6000 Python Quiz复习指南（Topic1,2,3,4,6,7）

Topic 1: Python Basics（Python 基础）

一、完整知识点（中英文对照）

1. **Python Overview（Python 概述）**
   - English: An easy-to-learn programming language widely used in many domains, especially popular in Data Science (DS) and AI applications. There are two incompatible versions: Python 2 and Python 3.
   - 中文：一种易于学习的编程语言，广泛应用于多个领域，在数据科学（DS）和人工智能（AI）应用中尤为流行。存在两个不兼容版本：Python 2 和 Python 3。
2. **Environment Setup（环境搭建）**
   - English: Check if Python is installed on the computer via `python -V` command in Anaconda Prompt. Use Jupyter Notebook / Jupyter Lab platform for program development.
   - 中文：通过 Anaconda Prompt 中的 `python -V` 命令检查电脑是否已安装 Python。使用 Jupyter Notebook / Jupyter Lab 平台进行程序开发。
3. **Basic I/O Operations（基本 I/O 操作）**
   - English: Use the `print()` function to output messages to users; use the `input()` function (available in Python 3.6 and later) to receive user input. The `input()` function treats received content as a string by default, which needs type casting for further manipulation.
   - 中文：使用 `print()` 函数向用户输出信息；使用 `input()` 函数（Python 3.6 及以上版本可用）接收用户输入。`input()` 函数默认将接收的内容视为字符串，如需进一步处理需进行类型转换。
4. **Variables（变量）**
   - English: Variables are "containers" for storing data values, allowing data to be retained and modified as needed. Python has no command to declare a variable; a variable is automatically created when a value is first assigned to it via an assignment statement.
   - 中文：变量是存储数据值的"容器"，可根据需要保留和修改数据。Python 没有声明变量的命令，首次通过赋值语句为变量赋值时，变量会自动创建。
5. **Variable Characteristics（变量特性）**
   - English: Variables in Python do not need to be declared with a specific type, and their type can even be changed after being set.
   - 中文：Python 中的变量无需指定类型声明，甚至在赋值后还可更改类型。
6. **Variable Naming Rules（变量命名规则）**
   - English:
     - Must start with a letter or underscore (_);
     - Cannot start with a number;
     - Can only contain alphanumeric characters and underscores (A-z, 0-9, _);

    - Case-sensitive.
  - 中文：
    - 必须以字母或下划线（_）开头；
    - 不能以数字开头；
    - 只能包含字母、数字和下划线（A-z、0-9、_）；
    - 区分大小写。

7. **Constants（常量）**
   - English: Constants are variables whose values are intended to remain unchanged during program execution. By convention, constants are named with uppercase letters to signal their immutability.
   - 中文：常量是在程序执行期间值预计保持不变的变量。按惯例，常量用大写字母命名，以标识其不可变性。

8. **Comments（注释）**
   - English:
     - Single-line comments start with #, and Python treats the rest of the line as a comment;
     - Multi-line comments (docstrings) are enclosed using triple quotes, which provide more comprehensive information about the code's purpose and usage than single-line comments;
     - Comments are used to explain code (improving readability) and skip execution during testing/debugging.
   - 中文：
     - 单行注释以 # 开头，Python 会将该行剩余部分视为注释；
     - 多行注释（文档字符串）用三重引号包裹，相比单行注释，能更全面地说明代码的用途和使用方法；
     - 注释用于解释代码（提升可读性），以及在测试/调试时跳过代码执行。

9. **Special Characters（特殊字符）**
   - English: The backslash (\) is a special escape character in Python. When used with specific characters (e.g., \n), it forms a new line character. It is also used to insert "illegal" characters in a string.
   - 中文：反斜杠（\）是 Python 中的特殊转义字符。与特定字符（如 \n）结合使用时，可形成换行符；也可用于在字符串中插入“非法”字符。

10. **Data Types（数据类型）** Python has several common built-in data types, as shown in the table below:

变量赋值示例	英文类型	中文类型
`x = "apple"`	String	字符串（文本/字符序列）
`x = 3.14`	Float	浮点数（小数）
`x = True` / `x = False`	Boolean	布尔值（逻辑真/假）
`x = ["apple", "banana", "cherry"]`	List	列表（有序可修改的元素集合）
`x = ("apple", "banana", "cherry")`	Tuple	元组（有序不可修改的元素集合）

变量赋值示例	英文类型	中文类型
`x = {"apple", "banana", "cherry"}`	Set	集合（无序无重复的元素集合）
`x = range(6)`	Range	范围（整数序列生成器）
`x = {"name": "John", "age": 36}`	Dictionary	字典（键值对集合）

11. **Type Casting & Annotation（类型转换与注解）**
    - English:
      - Type casting is used to change/specify the data type of a variable (e.g., `int(input())` converts string input to integer);
      - The `type()` function is used to check the data type of a variable;
      - Type annotation explicitly indicates the intended data type of variables/function parameters/return values (e.g., `def add(a: int, b: int) -> int`), improving code readability.
    - 中文：
      - 类型转换用于更改/指定变量的数据类型（如 `int(input())` 将字符串输入转换为整数）；
      - `type()` 函数用于检查变量的数据类型；
      - 类型注解明确指定变量/函数参数/返回值的预期数据类型（如 `def add(a: int, b: int) -> int`），提升代码可读性。

12. **In-place Operations（原地操作）**
    - English: Any mathematical operator can be used before the `=` character to form an in-place operation, which modifies the variable directly without reassigning a new variable. Common examples:
      - `x += 1` (increment in place);
      - `x *= 2` (multiply in place);
      - `x /= 2` (divide in place);
      - `x %= 2` (modulus in place);
      - `x **= 2` (raise to power in place).
    - 中文：任何数学运算符都可放在 `=` 前构成原地操作，直接修改变量而无需重新赋值新变量。常见示例：
      - `x += 1`（原地自增）；
      - `x *= 2`（原地自乘）；
      - `x /= 2`（原地自除）；
      - `x %= 2`（原地取模）；
      - `x **= 2`（原地求幂）。

13. **String Manipulation（字符串操作）**
    - English:
      - Strings are created by enclosing text in single or double quotes;
      - Access string elements via square brackets (indexing, e.g., `x[0]`; slicing, e.g., `x[1:4]`);
      - Common functions/methods:
        - `len()`: Get the length of a string;
        - `upper()`: Convert string to uppercase;
        - `lower()`: Convert string to lowercase;

        - `isupper()`: Check if string is uppercase;
        - `islower()`: Check if string is lowercase;
        - `find()`: Search for specific characters;
        - `strip()`: Remove whitespace from the beginning or end of a string.
    - 中文：
        - 字符串通过将文本用单引号或双引号包裹创建；
        - 通过方括号访问字符串元素（索引，如 `x[0]`；切片，如 `x[1:4]`）；
        - 常用函数/方法：
            - `len()`：获取字符串长度；
            - `upper()`：将字符串转为大写；
            - `lower()`：将字符串转为小写；
            - `isupper()`：检查字符串是否为大写；
            - `islower()`：检查字符串是否为小写；
            - `find()`：查找特定字符；
            - `strip()`：去除字符串开头或结尾的空格。

14. **Conditional Execution（条件执行）**

    - English:
        - Use `if-elif-else` statements to change program execution based on conditions; the expression in `if` is called a logical expression;
        - Python 3.10+ has the `match-case` statement, which can be used for comparison like `if-elif-else` but supports more complex pattern matching, reducing code and improving readability;
        - Python uses colons (`:`) and indentation (whitespace/Tab) to define code scope;
        - Logical expressions (Boolean expressions) return `True` or `False` (can also be interpreted as 1 or 0), commonly used for conditional execution.
    - 中文：
        - 使用 `if-elif-else` 语句根据条件改变程序执行流程，`if` 中的表达式称为逻辑表达式；
        - Python 3.10+ 新增 `match-case` 语句，可像 `if-elif-else` 一样用于比较，但支持更复杂的模式匹配，减少代码量并提升可读性；
        - Python 使用冒号（`:`）和缩进（空格/Tab）定义代码作用域；
        - 逻辑表达式（布尔表达式）返回 `True` 或 `False`（也可解释为 1 或 0），广泛用于条件执行。

15. **Common Logical Operators（常用逻辑运算符）**

英文运算符	中文含义	示例
`==`	等于	`x == 5`
`!=`	不等于	`x != 5`
`<`	小于	`x < 5`
`<=`	小于等于	`x <= 5`
`>`	大于	`x > 5`
`>=`	大于等于	`x >= 5`

16. **Loops（循环）**
    - English:
        - `while` loop: Executes a set of statements as long as a condition is true; supports infinite loops, which can be broken out of using the `break` statement; the `continue` statement skips the current iteration and proceeds to the next;
        - `for` loop: Typically used when the number of iterations is known, often used with the `range()` function (generates a sequence of numbers, starts from 0 by default, increments by 1 by default, ends at a specified number); can also iterate over sequences (list, tuple, dictionary, string);
        - `pass` statement: Used as a placeholder for future code to skip operations (e.g., in loops or conditional blocks where no action is needed temporarily);
        - When the element returned by the sequence is not needed in the loop body, a single underscore (_) can be used instead of a variable name.
    - 中文：
        - `while` 循环：只要条件为真，就执行一组语句；支持无限循环，可通过 `break` 语句跳出；`continue` 语句跳过当前迭代，进入下一次迭代；
        - `for` 循环：通常用于已知迭代次数的场景，常与 `range()` 函数（生成整数序列，默认从 0 开始，默认步长为 1，到指定数字结束）配合使用；也可遍历序列（列表、元组、字典、字符串）；
        - `pass` 语句：作为未来代码的占位符，用于跳过操作（如在暂时无需执行操作的循环或条件块中）；
        - 若循环体中无需使用序列返回的元素，可用单个下划线（_）代替变量名。

二、重点掌握内容

1. **Python 版本差异**：明确当前学习和使用的是 Python 3（与 Python 2 不兼容），避免语法混淆。
2. **变量与常量**：熟练掌握变量命名规则、赋值方式，以及常量的命名惯例（大写字母）。
3. **基本 I/O 操作**：重点理解 `input()` 函数的默认返回类型（字符串），以及类型转换的必要性（如 `int(input())`、`float(input())`）。
4. **数据类型**：能区分并正确使用常用数据类型（字符串、布尔值、列表、元组等），掌握 `type()` 函数的使用。
5. **字符串操作**：熟练运用索引、切片，以及 `len()`、`strip()`、`find()`、`upper()`、`lower()` 等常用方法。
6. **条件与循环**：
    - 掌握 `if-elif-else` 的逻辑流程，理解缩进对代码作用域的影响；
    - 区分 `while`（条件驱动）和 `for`（迭代次数驱动）的使用场景，熟练运用 `break`、`continue`、`pass` 语句；
    - 理解 `range()` 函数的参数含义（如 `range(2, 10, 2)` 表示从 2 到 9，步长为 2）。
7. **注释**：能正确编写单行注释和多行文档字符串，明确注释的作用（解释代码、调试跳过）。

三、例题（英文题目 + 中文解析）

1. 选择题（Multiple Choice）

1. Which of the following is a valid variable name in Python? A. 1student B. student-name C. student_name D. Student!

- **Answer**: C
- **Analysis**: According to variable naming rules:
  - Option A starts with a number (invalid);
  - Option B contains a hyphen (`-`, invalid);
  - Option C starts with a letter and only contains letters and underscores (valid);
  - Option D contains an exclamation mark (`!`, invalid).

2. What will be the output of the following code?

```
x = "Python Basics"
print(x[2:7])
```

A. "thon " B. "thonB" C. "ython" D. "ython "

- **Answer**: A
- **Analysis**: String slicing `x[start:end]` includes the character at `start` but excludes the character at `end`. Here, `x[2]` is "t", `x[6]` is " " (space), so the sliced result is "thon ".

3. Which statement is correct about the `while` loop and `for` loop in Python? A. The `while` loop can only be used for infinite loops. B. The `for` loop is suitable when the number of iterations is unknown. C. The `break` statement can exit both `while` and `for` loops. D. The `continue` statement exits the entire loop directly.

- **Answer**: C
- **Analysis**:
  - Option A is wrong: `while` loops can be used for finite loops (e.g., `x = 0; while x < 5: x += 1`);
  - Option B is wrong: `for` loops are suitable when the number of iterations is known (e.g., iterating over a fixed-length list);
  - Option C is correct: `break` exits the current loop regardless of `while` or `for`;
  - Option D is wrong: `continue` skips only the current iteration, not the entire loop.

2. 简答题（Short Answer）

1. Explain the difference between `strip()`, `lstrip()`, and `rstrip()` methods for Python strings.

- **Answer**: All three methods are used to remove whitespace from strings, but their scopes differ:
  - `strip()`: Removes whitespace from both the beginning and the end of the string;
  - `lstrip()`: Removes whitespace only from the **left** (beginning) of the string;
  - `rstrip()`: Removes whitespace only from the **right** (end) of the string.
  - Example: For `s = " Hello World "`, `s.strip()` returns "Hello World", `s.lstrip()` returns "Hello World ", `s.rstrip()` returns " Hello World".

2. What is the role of the `range()` function? Give an example to explain its three parameters (start, stop, step).

- **Answer**: The `range()` function is used to generate a sequence of integers, which is often used with `for` loops to control the number of iterations. It has three parameters:

    - `start`: The starting value of the sequence (default is 0 if not specified);
    - `stop`: The ending value of the sequence (the sequence does not include this value, so it is a "left-closed and right-open" interval);
    - `step`: The increment (step size) between consecutive values in the sequence (default is 1 if not specified, cannot be 0).
  - Example: `range(2, 10, 2)` generates the sequence [2, 4, 6, 8]. Here, `start=2` (starts at 2), `stop=10` (ends before 10), `step=2` (increments by 2 each time).

Topic 2: Data Structures（数据结构）

一、完整知识点（中英文对照）

1. **Four Built-in Data Structures（四种内置数据结构）** Python provides four built-in data structures for complex data storage needs, with distinct characteristics:

英文名称	中文名称	核心特性（English）	核心特性（中文）
List	列表	Ordered, changeable, allows duplicate members; created with square brackets (`[]`).	有序、可修改、允许重复元素；用方括号（`[]`）创建。
Tuple	元组	Ordered, unchangeable (immutable), allows duplicate members; created with parentheses (`()`).	有序、不可修改（不可变）、允许重复元素；用圆括号（`()`）创建。
Dictionary	字典	Ordered (Python 3.7+), changeable, no duplicate members; stores data in key-value pairs; created with curly braces (`{}`).	有序（Python 3.7+）、可修改、无重复元素；以键值对形式存储数据；用大括号（`{}`）创建。
Set	集合	Unordered, unchangeable (elements cannot be modified individually), unindexed, no duplicate members; created with curly braces (`{}`) or `set()` constructor.	无序、不可修改（元素无法单独修改）、无索引、无重复元素；用大括号（`{}`）或 `set()` 构造函数创建。

2. **List Operations（列表操作）**
   - English:
     - **Access items**: Use square brackets for indexing (e.g., `my_list[0]`) or slicing (e.g., `my_list[1:3]`); check if an item exists with `in` (e.g., `if "apple" in my_list`);
     - **Add items**: `append(item)` (adds to the end), `insert(index, item)` (adds at the specified index);
     - **Modify items**: Assign a new value via index (e.g., `my_list[2] = "orange"`);
     - **Sort items**: `sort()` (sorts in ascending order by default; use `reverse=True` for descending order, e.g., `my_list.sort(reverse=True)`);
     - **Extend list**: `extend(another_list)` (adds all elements of `another_list` to the end of the current list);

    - **Remove items**: `remove(item)` (removes the first occurrence of the specified item), `pop(index)` (removes the item at the specified index; removes the last item if index is not specified), `clear()` (removes all items);
    - **Other methods**: `count(item)` (returns the number of occurrences of the specified item), `index(item)` (returns the index of the first occurrence of the specified item), `copy()` (returns a copy of the list), `reverse()` (reverses the order of the list).
  - 中文：
    - **访问元素**：用方括号索引（如 `my_list[0]`）或切片（如 `my_list[1:3]`）；用 `in` 检查元素是否存在（如 `if "apple" in my_list`）；
    - **添加元素**：`append(item)`（添加到末尾）、`insert(index, item)`（在指定索引处添加）；
    - **修改元素**：通过索引赋值新值（如 `my_list[2] = "orange"`）；
    - **排序元素**：`sort()`（默认升序；用 `reverse=True` 降序，如 `my_list.sort(reverse=True)`）；
    - **扩展列表**：`extend(another_list)`（将 `another_list` 的所有元素添加到当前列表末尾）；
    - **删除元素**：`remove(item)`（删除指定元素的第一个匹配项）、`pop(index)`（删除指定索引处的元素；未指定索引时删除最后一个元素）、`clear()`（清空所有元素）；
    - **其他方法**：`count(item)`（返回指定元素的出现次数）、`index(item)`（返回指定元素的第一个匹配项的索引）、`copy()`（返回列表的副本）、`reverse()`（反转列表顺序）。

3. **List Comprehension（列表推导式）**
  - English: A concise Python syntax to create a new list by extracting or processing elements from an existing iterable (list, tuple, range, etc.). It is more efficient than traditional `for` loops (implemented in C) and reduces code length.
    - Example 1 (extract even numbers from `range(10)`): `even_numbers = [x for x in range(10) if x % 2 == 0]` (result: `[0, 2, 4, 6, 8]`);
    - Example 2 (square each number in `[1,2,3]`): `squares = [x**2 for x in [1,2,3]]` (result: `[1, 4, 9]`).
  - 中文：一种简洁的 Python 语法，通过从现有可迭代对象（列表、元组、range 等）中提取或处理元素来创建新列表。比传统 `for` 循环更高效（基于 C 实现），且减少代码量。
    - 示例 1（从 `range(10)` 中提取偶数）：`even_numbers = [x for x in range(10) if x % 2 == 0]`（结果：`[0, 2, 4, 6, 8]`）；
    - 示例 2（对 `[1,2,3]` 中的每个数求平方）：`squares = [x**2 for x in [1,2,3]]`（结果：`[1, 4, 9]`）。

4. **Tuple Operations（元组操作）**
  - English:
    - **Creation**: Created with parentheses (e.g., `my_tuple = ("apple", "banana")`); a single-element tuple needs a trailing comma (e.g., `my_tuple = ("apple",)`), otherwise it is treated as a string;
    - **Access items**: Use indexing (e.g., `my_tuple[1]`) or slicing (e.g., `my_tuple[0:2]`), same as lists;
    - **Methods**: Fewer methods than lists; only `count(item)` (returns the number of occurrences of the specified item) and `index(item)` (returns the index of the first occurrence of the specified item);

    - **Immutability**: Tuples cannot be modified (cannot add/remove/change items); to modify, convert the tuple to a list first (e.g., `temp_list = list(my_tuple)`), modify the list, then convert back to a tuple (e.g., `my_tuple = tuple(temp_list)`);
    - **Advantages**: Faster execution than lists; "safer" for data that should not be modified (defensive programming); can be used as dictionary keys (since they are hashable, while lists cannot).
  - 中文：
    - **创建**：用圆括号创建（如 `my_tuple = ("apple", "banana")`）；单个元素的元组需加尾随逗号（如 `my_tuple = ("apple",)`），否则会被视为字符串；
    - **访问元素**：与列表相同，用索引（如 `my_tuple[1]`）或切片（如 `my_tuple[0:2]`）；
    - **方法**：方法比列表少，仅 `count(item)`（返回指定元素的出现次数）和 `index(item)`（返回指定元素的第一个匹配项的索引）；
    - **不可变性**：元组无法修改（不能添加/删除/更改元素）；如需修改，需先将元组转为列表（如 `temp_list = list(my_tuple)`），修改列表后再转回元组（如 `my_tuple = tuple(temp_list)`）；
    - **优势**：执行速度比列表快；对不应修改的数据更“安全”（防御性编程）；可作为字典的键（因元组可哈希，而列表不可）。

5. **Dictionary Operations（字典操作）**

  - English:
    - **Data storage**: Stores data in key-value pairs; keys must be unique (duplicate keys will overwrite the previous value); keys must be hashable (e.g., string, integer, tuple; lists cannot be keys);
    - **Access items**: Access values via keys (e.g., `my_dict["name"]`); use `keys()` to get all keys (e.g., `my_dict.keys()`), `values()` to get all values (e.g., `my_dict.values()`), `items()` to get all key-value pairs (e.g., `my_dict.items()`);
    - **Modify items**: Add a new key-value pair (e.g., `my_dict["age"] = 25`), update an existing value (e.g., `my_dict["name"] = "Alice"`), delete a key-value pair (e.g., `del my_dict["age"]` or `my_dict.pop("age")`);
    - **Nested dictionaries**: A dictionary can contain another dictionary as a value (e.g., `my_dict = {"student": {"name": "Bob", "grade": "A"}}`); access nested values via chained keys (e.g., `my_dict["student"]["grade"]`);
    - **Dictionary Comprehension**: A concise syntax to create a dictionary in one line, similar to list comprehension. Example: `square_dict = {x: x**2 for x in range(1, 4)}` (result: `{1:1, 2:4, 3:9}`).
  - 中文：
    - **数据存储**：以键值对形式存储数据；键必须唯一（重复键会覆盖之前的值）；键必须可哈希（如字符串、整数、元组；列表不能作为键）；
    - **访问元素**：通过键访问值（如 `my_dict["name"]`）；用 `keys()` 获取所有键（如 `my_dict.keys()`）、`values()` 获取所有值（如 `my_dict.values()`）、`items()` 获取所有键值对（如 `my_dict.items()`）；
    - **修改元素**：添加新键值对（如 `my_dict["age"] = 25`）、更新已有值（如 `my_dict["name"] = "Alice"`）、删除键值对（如 `del my_dict["age"]` 或 `my_dict.pop("age")`）；
    - **嵌套字典**：字典的值可包含另一个字典（如 `my_dict = {"student": {"name": "Bob", "grade": "A"}}`）；通过链式键访问嵌套值（如 `my_dict["student"]["grade"]`）；

- **字典推导式**：与列表推导式类似，用简洁的单行语法创建字典。示例：`square_dict = {x: x**2 for x in range(1, 4)}`（结果：`{1:1, 2:4, 3:9}`）。

6. **Set Operations（集合操作）**

   - English:
     - **Creation**: Created with curly braces (e.g., `my_set = {"apple", "banana"}`) or `set()` constructor (e.g., `my_set = set(["apple", "banana"])`); empty sets must use `set()` (curly braces `{}` create empty dictionaries);
     - **Duplicate removal**: Automatically removes duplicate elements (e.g., `set([1,2,2,3])` returns `{1,2,3}`);
     - **Key characteristics**: Unordered (elements have no fixed order; order may change when accessed) and unindexed (cannot access elements via index);
     - **Common operations**:
       - `add(item)`: Adds a single element to the set;
       - `update(another_iterable)`: Adds multiple elements from an iterable (list, tuple, etc.) to the set;
       - `union()`: Returns a new set containing all elements from both sets (e.g., `set1.union(set2)`);
       - `intersection()`: Returns a new set containing common elements of both sets (e.g., `set1.intersection(set2)`);
       - `difference()`: Returns a new set containing elements in `set1` but not in `set2` (e.g., `set1.difference(set2)`);
     - **Use cases**: Membership testing (check if an element exists, faster than lists), duplicate removal, and mathematical set operations (union, intersection, difference).
   - 中文：
     - **创建**：用大括号（如 `my_set = {"apple", "banana"}`）或 `set()` 构造函数（如 `my_set = set(["apple", "banana"])`）；空集合必须用 `set()`（大括号 `{}` 创建空字典）；
     - **自动去重**：自动删除重复元素（如 `set([1,2,2,3])` 返回 `{1,2,3}`）；
     - **核心特性**：无序（元素无固定顺序，访问时顺序可能变化）、无索引（无法通过索引访问元素）；
     - **常用操作**：
       - `add(item)`：向集合添加单个元素；
       - `update(another_iterable)`：从可迭代对象（列表、元组等）向集合添加多个元素；
       - `union()`：返回包含两个集合所有元素的新集合（如 `set1.union(set2)`）；
       - `intersection()`：返回包含两个集合共同元素的新集合（如 `set1.intersection(set2)`）；
       - `difference()`：返回包含 `set1` 中有但 `set2` 中没有的元素的新集合（如 `set1.difference(set2)`）；
     - **应用场景**：成员检测（检查元素是否存在，比列表更快）、去重、数学集合运算（并集、交集、差集）。

二、重点掌握内容

1. **四种数据结构的核心区别**：从“有序/无序”“可修改/不可修改”“是否允许重复”三个维度对比列表、元组、字典、集合，明确各自适用场景（如列表用于动态数据序列，元组用于固定配置，字典用于键值映射，集合用于去重和成员检测）。

2. **列表操作**：熟练掌握 `append()`、`insert()`、`pop()`、`remove()`、`sort()`、`extend()` 的用法，理解列表推导式的语法和优势。
3. **元组的不可变性**：明确元组无法修改的特性，以及与列表的性能差异（元组更快）和应用差异（元组可作字典键）。
4. **字典操作**：重点掌握键值对的访问、添加、更新、删除，以及 `keys()`、`values()`、`items()` 的使用，理解字典推导式的应用。
5. **集合操作**：掌握集合的自动去重特性，以及 `add()`、`union()`、`intersection()` 的用法，明确集合"无序无索引"的特点。

三、例题（英文题目 + 中文解析）

1. 选择题（Multiple Choice）

1. Which data structure in Python does NOT allow duplicate elements? A. List B. Tuple C. Set D. Dictionary

   - **Answer**: C
   - **Analysis**:
     - Lists (A) and tuples (B) allow duplicates;
     - Dictionaries (D) do not allow duplicate keys, but values can be duplicated;
     - Sets (C) automatically remove duplicates and do not allow any duplicate elements.

2. What will be the output of the following code?

   ```
   my_list = [1, 3, 2, 4]
   my_list.sort()
   my_list.reverse()
   print(my_list)
   ```

   A. [4, 3, 2, 1] B. [1, 2, 3, 4] C. [1, 3, 2, 4] D. [4, 2, 3, 1]

   - **Answer**: A
   - **Analysis**: `my_list.sort()` sorts the list in ascending order, resulting in `[1, 2, 3, 4]`; `my_list.reverse()` reverses the sorted list, resulting in `[4, 3, 2, 1]`.

3. Which of the following statements about dictionaries is correct? A. Dictionary keys can be lists. B. Duplicate keys in a dictionary will cause an error. C. We can access dictionary values via indexes. D. The `items()` method returns all key-value pairs of the dictionary.

   - **Answer**: D
   - **Analysis**:
     - Option A is wrong: Dictionary keys must be hashable, and lists are unhashable (cannot be keys);
     - Option B is wrong: Duplicate keys overwrite the previous value, not causing an error;
     - Option C is wrong: Dictionaries are unindexed (before Python 3.7, they were unordered), so values cannot be accessed via indexes;
     - Option D is correct: `items()` returns a view object containing all key-value pairs (e.g., `dict_items([("name", "John"), ("age", 36)])`).

2. 简答题（Short Answer）

1. Explain the difference between `list.append()` and `list.extend()` methods with examples.
    - **Answer**: Both methods are used to add elements to a list, but their handling of input elements differs:
        - `append(item)`: Adds the entire `item` as a single element to the end of the list, regardless of whether `item` is an iterable (list, tuple, etc.).
            - Example: `my_list = [1,2]; my_list.append([3,4])` → `my_list` becomes `[1, 2, [3, 4]]` (the input list `[3,4]` is added as one element);
        - `extend(another_iterable)`: Adds each element of `another_iterable` (must be iterable) to the end of the list, "expanding" the list with individual elements.
            - Example: `my_list = [1,2]; my_list.extend([3,4])` → `my_list` becomes `[1, 2, 3, 4]` (each element of `[3,4]` is added separately).
2. Why can tuples be used as dictionary keys, but lists cannot?
    - **Answer**: Dictionary keys require the property of "hashability" (the hash value of the key remains unchanged during its lifetime).
        - Tuples are immutable: Once created, their elements cannot be modified, so their hash value is fixed (hashable), making them eligible as dictionary keys;
        - Lists are mutable: Their elements can be added, removed, or modified at any time, which changes their hash value (unhashable), so they cannot be used as dictionary keys.

Topic 3: Functions（函数）

一、完整知识点（中英文对照）

1. **Function Basics（函数基础）**
    - English: A function is a reusable block of code that runs only when called. It improves code readability and reusability. Python has built-in functions (e.g., `print()`, `input()`, `len()`) and allows users to define custom functions.
    - 中文：函数是可重用的代码块，仅在被调用时执行。它提升代码可读性和复用性。Python 有内置函数（如 `print()`、`input()`、`len()`），也允许用户定义自定义函数。
2. **Custom Function Definition & Call（自定义函数定义与调用）**
    - English:
        - **Definition**: Use the `def` keyword to define a function, followed by the function name, parentheses (`()`), and a colon (`:`). The function body is indented. Syntax:

```
def function_name(parameters):
    # Function body (indented)
    function_logic
    return return_value  # Optional
```

        - **Call**: Execute the function by its name followed by parentheses (with arguments if needed). Example: `result = function_name(arguments)`.
    - 中文：

- **定义**：用 `def` 关键字定义函数，后跟函数名、圆括号（`()`）和冒号（`:`）。函数体需缩进。语法：

```
def 函数名(参数):
    # 函数体（缩进）
    函数逻辑
    return 返回值  # 可选
```

- **调用**：通过“函数名 + 圆括号”（需传参时在括号内传实参）执行函数。示例：`result = 函数名(实参)`。

3. **Function Naming Guidelines（函数命名规范）**
   - English:
     - Can use letters, digits (0-9), and underscores (_);
     - Should clearly describe the function's action (e.g., `calculate_sum()` instead of `func1()`);
     - Follow camelCase (e.g., `getUserInput()`) or snake_case (more common in Python, e.g., `get_user_input()`) convention;
     - A single leading underscore (e.g., `_internal_function()`) indicates the function is for internal use only (not intended to be called externally).
   - 中文：
     - 可包含字母、数字（0-9）和下划线（_）；
     - 应清晰描述函数功能（如用 `calculate_sum()` 而非 `func1()`）；
     - 遵循驼峰命名法（如 `getUserInput()`）或蛇形命名法（Python 中更常用，如 `get_user_input()`）；
     - 单个前导下划线（如 `_internal_function()`）表示函数仅用于内部（不建议外部调用）。
4. **Parameters & Arguments（参数与实参）**
   - English:
     - **Parameters**: Variables defined in the function definition (e.g., `a` and `b` in `def add(a, b):`), representing the data the function expects to receive.
     - **Arguments**: Actual values passed to the function when called (e.g., `3` and `5` in `add(3, 5)`), which are assigned to the corresponding parameters.
     - **Argument passing types**:
       - Positional arguments: Passed in the same order as parameters (e.g., `add(3,5)` assigns `3` to `a` and `5` to `b`);
       - Keyword arguments: Passed with `parameter_name=value` (order does not matter, e.g., `add(b=5, a=3)`).
   - 中文：
     - **参数（Parameters）**：函数定义中声明的变量（如 `def add(a, b):` 中的 `a` 和 `b`），表示函数预期接收的数据。
     - **实参（Arguments）**：调用函数时传入的实际值（如 `add(3, 5)` 中的 `3` 和 `5`），会赋值给对应的参数。
     - **实参传递方式**：
       - 位置实参：按参数定义顺序传递（如 `add(3,5)` 中 `3` 赋值给 `a`，`5` 赋值给 `b`）；
       - 关键字实参：用 `参数名=值` 传递（顺序无关，如 `add(b=5, a=3)`）。
5. **Default Parameters（默认参数）**

- English: Parameters with a predefined value (assigned in the function definition). If no argument is passed for the parameter when calling the function, the default value is used. Syntax: `def function_name(param1, param2=default_value):`.
  - Example:

```
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")
greet("Bob")  # Uses default greeting: "Hello, Bob!"
greet("Alice", "Hi")  # Overrides default: "Hi, Alice!"
```

  - Note: Default parameters must be placed after non-default parameters (otherwise, a syntax error occurs).
- 中文：函数定义中预先赋值的参数。调用函数时若未给该参数传实参，则使用默认值。语法：`def 函数名(参数1, 参数2=默认值):`。
  - 示例：

```
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")
greet("Bob")  # 使用默认问候语："Hello, Bob!"
greet("Alice", "Hi")  # 覆盖默认值："Hi, Alice!"
```

  - 注意：默认参数必须放在非默认参数之后（否则会报语法错误）。

6. **Variable-length Arguments（变长参数）**
   - English: Used when the number of arguments is unknown in advance. Two types:
     - `*args`: Accepts a variable number of positional arguments, which are stored as a tuple. Syntax: `def function_name(*args):`.
       - Example:

```
def sum_all(*args):
    return sum(args)
print(sum_all(1,2,3))  # Returns 6 (args = (1,2,3))
```

     - `**kwargs`: Accepts a variable number of keyword arguments, which are stored as a dictionary. Syntax: `def function_name(**kwargs):`.
       - Example:

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
print_info(name="Bob", age=25)  # kwargs = {"name":"Bob", "age":25}
```

   - 中文：用于预先未知实参数量的场景，分两种类型：

- `*args`：接收可变数量的位置实参，存储为元组。语法：`def 函数名(*args):`。
  - 示例：

```
def sum_all(*args):
    return sum(args)
print(sum_all(1,2,3))  # 返回 6 (args = (1,2,3))
```

- `**kwargs`：接收可变数量的关键字实参，存储为字典。语法：`def 函数名(**kwargs):`。
  - 示例：

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
print_info(name="Bob", age=25)  # kwargs = {"name":"Bob",
"age":25}
```

7. **Function Return Value（函数返回值）**
   - English:
     - Use the `return` statement to send a value from the function back to the caller. A function can return one or multiple values (multiple values are returned as a tuple).
     - Example 1 (single return value):

```
def add(a, b):
    return a + b
result = add(3,5)  # result = 8
```

     - Example 2 (multiple return values):

```
def get_name_and_age():
    return "Bob", 25  # Returns a tuple ("Bob", 25)
name, age = get_name_and_age()  # Unpacks the tuple: name="Bob",
age=25
```

     - A function with no `return` statement returns `None` by default.
   - 中文：
     - 用 `return` 语句将函数的计算结果返回给调用者。函数可返回单个或多个值（多个值以元组形式返回）。
     - 示例 1（单个返回值）：

```
def add(a, b):
    return a + b
result = add(3,5)  # result = 8
```

- 示例 2（多个返回值）：

```
def get_name_and_age():
    return "Bob", 25  # 返回元组 ("Bob", 25)
name, age = get_name_and_age()  # 解包元组：name="Bob", age=25
```

- 无 `return` 语句的函数默认返回 `None`。

8. **Type Annotation（类型注解）**
   - English: Explicitly indicates the data types of function parameters and return values (improves code readability, but Python does not enforce type checks). Syntax:

```
def function_name(param1: type1, param2: type2) -> return_type:
    # Function body
```

   - Example:

```
def calculate_area(radius: float) -> float:
    return 3.14159 * radius ** 2
```

   - Here, `radius: float` indicates `radius` should be a float, and `-> float` indicates the function returns a float.
   - 中文：明确指定函数参数和返回值的数据类型（提升代码可读性，但 Python 不强制类型检查）。语法：

```
def 函数名(参数1: 类型1, 参数2: 类型2) -> 返回值类型:
    # 函数体
```

   - 示例：

```
def calculate_area(radius: float) -> float:
    return 3.14159 * radius ** 2
```

   - 其中 `radius: float` 表示 `radius` 应为浮点数，`-> float` 表示函数返回浮点数。

9. **Docstrings（文档字符串）**
   - English: Multi-line comments enclosed in triple quotes (`""" """`), used to document the function's purpose, parameters, return values, and usage. Accessible via the `__doc__` attribute or `help()` function.
     - Example:

```
def add(a, b):
    """Calculate the sum of two numbers.

    Parameters:
        a (int/float): The first number to add.
        b (int/float): The second number to add.

    Returns:
        int/float: The sum of a and b.
    """
    return a + b
print(add.__doc__)  # Prints the docstring
help(add)  # Displays the docstring in a user-friendly format
```

- 中文：用三重引号（`""" """`）包裹的多行注释，用于说明函数的用途、参数、返回值和使用方法。可通过 `__doc__` 属性或 `help()` 函数查看。
  - 示例：

```
def add(a, b):
    """计算两个数的和。

    参数:
        a (int/float): 要相加的第一个数。
        b (int/float): 要相加的第二个数。

    返回值:
        int/float: a 和 b 的和。
    """
    return a + b
print(add.__doc__)  # 打印文档字符串
help(add)  # 以友好格式显示文档字符串
```

10. **Variable Scope（变量作用域）**
    - English: The range in which a variable is accessible. Two main scopes:
      - **Local scope**: Variables defined inside a function, accessible only within the function.
      - **Global scope**: Variables defined outside all functions, accessible throughout the program (including inside functions if declared with the `global` keyword).
    - Example (global variable):

```
game_status = "running"  # Global variable

def stop_game():
    global game_status  # Declare to use the global variable
    game_status = "stopped"

stop_game()
print(game_status)  # Output: "stopped"
```

- **Nonlocal scope**: For variables defined in an outer nested function (not global), declared with the `nonlocal` keyword to modify them in an inner nested function.
- 中文：变量可被访问的范围，主要分两种：
  - **局部作用域**：函数内部定义的变量，仅在函数内部可访问。
  - **全局作用域**：所有函数外部定义的变量，整个程序均可访问（函数内部需用 `global` 关键字声明才能修改）。
- 示例（全局变量）：

```
game_status = "running"  # 全局变量

def stop_game():
    global game_status  # 声明使用全局变量
    game_status = "stopped"

stop_game()
print(game_status)  # 输出: "stopped"
```

- **非局部作用域**：用于外层嵌套函数中定义的变量（非全局），需用 `nonlocal` 关键字声明，才能在内层嵌套函数中修改。

11. **Special Function Types（特殊函数类型）**
    - **Lambda Function（匿名函数）**：
      - English: An anonymous function (no formal name) defined with the `lambda` keyword. Used for simple, one-time operations. Syntax: `lambda parameters: expression` (returns the result of the expression).
        - Example: `add = lambda a, b: a + b` (equivalent to the `add()` function defined with `def`); often used with `filter()` or `map()`.
      - 中文：用 `lambda` 关键字定义的匿名函数（无正式名称），用于简单、一次性的操作。语法：`lambda 参数: 表达式`（返回表达式的计算结果）。
        - 示例：`add = lambda a, b: a + b`（等效于用 `def` 定义的 `add()` 函数）；常与 `filter()` 或 `map()` 配合使用。
    - **Generator Function（生成器函数）**：
      - English: A function that uses the `yield` statement instead of `return` to return an iterator. It generates values on-the-fly (one at a time) instead of storing all values in memory, improving efficiency for large sequences.
        - Example:

```
def generate_numbers(n):
    for i in range(n):
        yield i  # Returns i one by one, pauses after each
yield
gen = generate_numbers(3)
print(next(gen))  # Output: 0
print(next(gen))  # Output: 1
```

- 中文：使用 `yield` 语句而非 `return` 返回迭代器的函数。它动态生成值（一次一个），而非将所有值存储在内存中，对大型序列处理更高效。
  - 示例：

```
def generate_numbers(n):
    for i in range(n):
        yield i  # 逐个返回 i，每次 yield 后暂停
gen = generate_numbers(3)
print(next(gen))  # 输出: 0
print(next(gen))  # 输出: 1
```

- **Decorator Function（装饰器函数）**：
  - English: A function that takes another function as an argument, extends its functionality, and returns the modified function. Denoted with the `@decorator_name` syntax above the target function.
    - Example (add logging to a function):

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Function {func.__name__} finished")
        return result
    return wrapper

@log_decorator  # Apply the decorator
def add(a, b):
    return a + b

add(3,5)  # Logs: "Calling function: add" → computes 8 → logs
"Function add finished"
```

  - 中文：接收另一个函数作为参数，扩展其功能并返回修改后函数的函数。用 `@装饰器名` 语法标注在目标函数上方。
    - 示例（为函数添加日志功能）：

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"调用函数: {func.__name__}")
        result = func(*args, **kwargs)
        print(f"函数 {func.__name__} 执行完成")
        return result
    return wrapper

@log_decorator  # 应用装饰器
def add(a, b):
    return a + b
```

```
add(3,5)  # 日志："调用函数：add" → 计算 8 → 日志 "函数 add 执行完成"
```

12. **Filter Function（filter() 函数）**
    - English: A built-in function that filters elements from an iterable based on a function's condition. It takes two arguments: a function (returns `True`/`False`) and an iterable. Returns an iterator containing elements for which the function returns `True`. Often used with lambda functions.
      - Example (filter even numbers):

```
numbers = [1,2,3,4,5]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # Output: [2,4]
```

    - 中文：内置函数，根据函数条件过滤可迭代对象中的元素。接收两个参数：一个返回 `True`/`False` 的函数，以及一个可迭代对象。返回包含"函数返回 `True` 的元素"的迭代器。常与 lambda 函数配合使用。
      - 示例（过滤偶数）：

```
numbers = [1,2,3,4,5]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # 输出: [2,4]
```

二、重点掌握内容

1. **函数定义与调用**：熟练写出自定义函数的语法，掌握函数调用时实参的传递方式（位置实参、关键字实参）。
2. **参数类型**：理解默认参数、`*args`（变长位置参数）、`**kwargs`（变长关键字参数）的用法，明确默认参数的位置规则（需在非默认参数后）。
3. **返回值**：掌握 `return` 语句的使用，理解多返回值的本质（元组解包），知道无 `return` 时函数返回 `None`。
4. **类型注解与文档字符串**：能为函数添加规范的类型注解和文档字符串，理解其提升代码可读性的作用。
5. **变量作用域**：区分局部变量和全局变量，掌握 `global` 关键字的使用（修改全局变量），了解 `nonlocal` 关键字的场景。
6. **特殊函数**：
   - 掌握 lambda 函数的语法和应用场景（简单、一次性操作，配合 `filter()`/`map()`）；
   - 理解装饰器的核心作用（扩展函数功能），能看懂基础的装饰器代码；
   - 了解生成器函数的优势（节省内存）和 `yield` 语句的作用。
7. **filter() 函数**：掌握其过滤逻辑，能结合 lambda 函数实现简单的元素筛选。

三、例题（英文题目 + 中文解析）

1. 选择题（Multiple Choice）

1. What will be the output of the following code?

```
def calculate(a, b=2):
    return a * b
print(calculate(3), calculate(3, 4))
```

A. 6 12 B. 3 12 C. 6 3 D. 2 4

- **Answer**: A
- **Analysis**: The function `calculate()` has a default parameter `b=2`.
  - `calculate(3)` uses the default `b=2`, so `3*2=6`;
  - `calculate(3,4)` overrides `b` to 4, so `3*4=12`. The output is "6 12".

2. Which of the following is a correct use of the `lambda` function? A. `lambda x, y: x + y(3,5)` B. `add = lambda x, y: x + y; add(3,5)` C. `def lambda(x): return x**2` D. `lambda x: x**2(4)`

- **Answer**: B
- **Analysis**:
  - Option A is wrong: The lambda function is not assigned or called correctly (the `(3,5)` should be after the lambda, not `y`);
  - Option B is correct: Assign the lambda function to `add`, then call `add(3,5)` (returns 8);
  - Option C is wrong: `lambda` is a keyword for anonymous functions, cannot be used with `def`;
  - Option D is wrong: The lambda function is not called correctly (should be `(lambda x: x**2)(4)`).

3. What is the purpose of the `global` keyword in a Python function? A. To define a new global variable inside the function. B. To modify a global variable inside the function. C. To make a local variable accessible globally. D. To delete a global variable inside the function.

- **Answer**: B
- **Analysis**: The `global` keyword is used to declare that a variable used inside the function is a global variable (defined outside the function). This allows modification of the global variable inside the function.
  - Option A is wrong: `global` does not define new global variables (new global variables are defined outside functions);
  - Option C is wrong: Local variables cannot be made global via `global`;
  - Option D is wrong: `global` is not for deleting variables.

2. 简答题（Short Answer）

1. Explain the difference between `return` and `print()` in a Python function.

- **Answer**: The two have completely different purposes:
  - `return`: Used to send the function's result back to the caller. It terminates the function execution (code after `return` is not executed) and the returned value can be assigned to a variable or used in other calculations. Example: `result = add(3,5)` (the value returned by `add()` is stored in `result`);

    - `print()`: Used to output information to the console for display. It does not return a value (or returns `None`) and does not affect the function's return value. Example: A function with only `print(a + b)` will return `None` by default, even if it prints the sum.

2. What is a decorator function? Give a simple example to explain how it works.

    - **Answer**: A decorator function is a special function that takes another function as input, extends its functionality without modifying its original code, and returns the modified function. It simplifies code reuse for cross-cutting concerns (e.g., logging, authentication).
    - Example (decorator to measure function execution time):

```
import time

# Define the decorator function
def time_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()  # Record start time (extended
functionality)
        result = func(*args, **kwargs)  # Call the original function
        end_time = time.time()  # Record end time (extended
functionality)
        print(f"Function {func.__name__} took {end_time -
start_time:.2f} seconds")
        return result  # Return the original function's result
    return wrapper

# Apply the decorator to the target function
@time_decorator
def slow_function():
    time.sleep(1)  # Simulate time-consuming operation
    print("Function executed")

# Call the decorated function
slow_function()
```

    - **Working principle**:
        1. When `@time_decorator` is added above `slow_function`, Python automatically passes `slow_function` as an argument to `time_decorator`;
        2. `time_decorator` defines a nested `wrapper` function that adds "time measurement" functionality before and after calling `slow_function`;
        3. `time_decorator` returns `wrapper`, so calling `slow_function()` actually calls `wrapper()`;
        4. `wrapper()` executes the extended logic (time recording) and the original function logic, then returns the result.

Topic 4: Modules and Packages（模块与包）

一、完整知识点（中英文对照）

1. **Module Basics（模块基础）**

- English: A module is a `.py` file containing Python code (functions, variables, classes) that can be reused in other programs. It improves code organization and reusability. Python has built-in modules (e.g., `time`, `re`) and allows custom modules.
- 中文：模块是包含 Python 代码（函数、变量、类）的 `.py` 文件，可在其他程序中重用，提升代码组织性和复用性。Python 有内置模块（如 `time`、`re`），也支持自定义模块。

2. **Importing Modules（导入模块）**
   - English: Use the `import` keyword to load a module's code into the current program. Four common import methods:
     - **Import the entire module**: `import module_name`; access elements via `module_name.element` (e.g., `import math; print(math.pi)`).
     - **Import a specific element**: `from module_name import element` (e.g., `from math import pi; print(pi)`); multiple elements can be imported with commas: `from math import pi, sqrt`.
     - **Import an element with an alias**: `from module_name import element as alias` (e.g., `from math import sqrt as square_root; print(square_root(16))`).
     - **Import the entire module with an alias**: `import module_name as alias` (e.g., `import numpy as np; arr = np.array([1,2,3])`; common in AI libraries to shorten names).
   - 中文：用 `import` 关键字将模块的代码加载到当前程序中，常见四种导入方式：
     - **导入整个模块**：`import 模块名`；通过 `模块名.元素` 访问（如 `import math; print(math.pi)`）。
     - **导入特定元素**：`from 模块名 import 元素`（如 `from math import pi; print(pi)`）；多个元素用逗号分隔：`from math import pi, sqrt`。
     - **给元素起别名**：`from 模块名 import 元素 as 别名`（如 `from math import sqrt as square_root; print(square_root(16))`）。
     - **给模块起别名**：`import 模块名 as 别名`（如 `import numpy as np; arr = np.array([1,2,3])`；AI 库中常用，简化名称）。
3. **Viewing Module Elements（查看模块元素）**
   - English: Use the `dir()` function to get a list of all elements (functions, variables, classes) in a module. Example: `import math; print(dir(math))` (outputs a list including `pi`, `sqrt`, `sin`, etc.).
   - 中文：用 `dir()` 函数获取模块中所有元素（函数、变量、类）的列表。示例：`import math; print(dir(math))`（输出包含 `pi`、`sqrt`、`sin` 等的列表）。
4. **Custom Modules（自定义模块）**
   - English:
     - **Create**: Write code (functions, variables) in a `.py` file (e.g., `my_module.py` with a function `calculate_sum(a, b)`).
     - **Import and use**: In another Python file (in the same directory), import the custom module and use its elements. Example:

```
# In main.py
import my_module
print(my_module.calculate_sum(3,5))  # Calls the function from
my_module.py
```

- **Test the module**: Add test code at the end of the custom module, wrapped in `if __name__ == "__main__":` (this code runs only when the module is executed directly, not when imported). Example:

```
# In my_module.py
def calculate_sum(a, b):
    return a + b

if __name__ == "__main__":
    # Test code
    print(calculate_sum(2,3))  # Runs when my_module.py is
executed directly
```

- 中文：
  - **创建**：在 `.py` 文件中编写代码（函数、变量）（如 `my_module.py` 中定义函数 `calculate_sum(a, b)`）。
  - **导入使用**：在另一个 Python 文件（同一目录下）中导入自定义模块并使用其元素。示例：

```
# 在 main.py 中
import my_module
print(my_module.calculate_sum(3,5))  # 调用 my_module.py 中的函数
```

  - **测试模块**：在自定义模块末尾添加测试代码，用 `if __name__ == "__main__":` 包裹（该代码仅在模块被直接执行时运行，被导入时不运行）。示例：

```
# 在 my_module.py 中
def calculate_sum(a, b):
    return a + b

if __name__ == "__main__":
    # 测试代码
    print(calculate_sum(2,3))  # 仅当直接执行 my_module.py 时运行
```

5. **Built-in Module: re (Regular Expressions)（内置模块：re 正则表达式）**
   - English: The `re` module provides functions for working with regular expressions (patterns used to match character sequences in strings). Common functions and patterns:
     - **Common functions**:
       - `re.search(pattern, string)`: Searches the string for the first match of the pattern; returns a match object if found, `None` otherwise.
       - `re.match(pattern, string)`: Matches the pattern only at the **start** of the string; returns a match object if found, `None` otherwise.
       - `re.findall(pattern, string)`: Returns a list of all non-overlapping matches of the pattern in the string.
     - **Common patterns**:

        - `^`: Starts with (e.g., `^Hello` matches strings starting with "Hello");
        - `$`: Ends with (e.g., `World$` matches strings ending with "World");
        - `.`: Matches any single character (except a newline);
        - `*`: Matches zero or more occurrences of the preceding character (e.g., `a*` matches "", "a", "aa", etc.);
        - `[abc]`: Matches any one of the characters in the set (e.g., matches "a", "b", or "c").
    - Example (check if a string starts with "Python"):

```
import re
string = "Python Basics"
if re.match("^Python", string):
    print("Matches")  # Output: "Matches"
```

- 中文：`re` 模块提供正则表达式相关功能（正则表达式是用于匹配字符串中字符序列的模式）。常用函数和模式：
    - **常用函数**：
        - `re.search(pattern, string)`：在字符串中搜索模式的第一个匹配项；找到则返回匹配对象，否则返回 `None`。
        - `re.match(pattern, string)`：仅在字符串**开头**匹配模式；找到则返回匹配对象，否则返回 `None`。
        - `re.findall(pattern, string)`：返回字符串中模式所有非重叠匹配项的列表。
    - **常用模式**：
        - `^`：以指定字符开头（如 `^Hello` 匹配以“Hello”开头的字符串）；
        - `$`：以指定字符结尾（如 `World$` 匹配以“World”结尾的字符串）；
        - `.`：匹配任意单个字符（换行符除外）；
        - `*`：匹配前面字符的零次或多次出现（如 `a*` 匹配“”、“a”、“aa”等）；
        - `[abc]`：匹配集合中的任意一个字符（如匹配“a”、“b”或“c”）。
    - 示例（检查字符串是否以“Python”开头）：

```
import re
string = "Python Basics"
if re.match("^Python", string):
    print("匹配成功")  # 输出："匹配成功"
```

6. **Packages（包）**
    - English: A package is a collection of related modules, organized in a directory structure. To be recognized as a Python package, the directory must contain a `__init__.py` file (can be empty, used to mark the directory as a package). Packages help manage large-scale codebases (e.g., AI libraries like NumPy, Pandas are packages composed of multiple modules).
    - 中文：包是相关模块的集合，以目录结构组织。要被识别为 Python 包，该目录必须包含 `__init__.py` 文件（可为空，用于标记目录为包）。包用于管理大规模代码库（如 NumPy、Pandas 等 AI 库就是由多个模块组成的包）。
7. **Third-party Packages（第三方包）**
    - English:

    - Third-party packages are non-built-in code libraries developed by the community, available for installation and use. Examples include NumPy (for numerical computing), Pandas (for data handling), Matplotlib (for data visualization).
    - **Installation tools**:
        - **PIP**: The official Python package manager. Command: `pip install package_name` (e.g., `pip install pandas`).
        - **Conda**: Used with Anaconda/Miniconda (common in AI environments). Commands:
            - Default channel: `conda install package_name` (e.g., `conda install pandas`);
            - Custom channel (e.g., conda-forge, a popular community channel): `conda install -c conda-forge package_name`.
    - **Import after installation**: Once installed, import the package/module in code (e.g., `import pandas as pd`).
  - 中文：
    - 第三方包是社区开发的非内置代码库，可安装后使用。示例包括 NumPy（数值计算）、Pandas（数据处理）、Matplotlib（数据可视化）。
    - **安装工具**：
        - **PIP**：Python 官方包管理器。命令：`pip install 包名`（如 `pip install pandas`）。
        - **Conda**：配合 Anaconda/Miniconda 使用（AI 环境中常用）。命令：
            - 默认频道：`conda install 包名`（如 `conda install pandas`）；
            - 自定义频道（如社区热门频道 conda-forge）：`conda install -c conda-forge 包名`。
    - **安装后导入**：安装完成后，在代码中导入包/模块（如 `import pandas as pd`）。
8. **Key Concepts: Library vs. Package vs. Module（核心概念：库 vs 包 vs 模块）**
  - English:

英文术语	中文术语	定义（English）	定义（中文）
Module	模块	A single `.py` file with Python code (functions, variables).	单个包含 Python 代码（函数、变量）的 `.py` 文件。
Package	包	A directory of related modules (with `__init__.py`).	包含相关模块的目录（需有 `__init__.py` 文件）。
Library	库	A collection of packages/modules that provide a set of related functionalities (e.g., Matplotlib is a library for visualization).	提供一组相关功能的包/模块集合（如 Matplotlib 是可视化库）。

  - 中文：

英文术语	中文术语	定义（English）	定义（中文）

英文术语	中文术语	定义（English）	定义（中文）
Module	模块	单个包含 Python 代码（函数、变量）的 `.py` 文件。	A single `.py` file with Python code (functions, variables).
Package	包	包含相关模块的目录（需有 `__init__.py` 文件）。	A directory of related modules (with `__init__.py`).
Library	库	提供一组相关功能的包/模块集合（如 Matplotlib 是可视化库）。	A collection of packages/modules that provide a set of related functionalities (e.g., Matplotlib is a library for visualization).

二、重点掌握内容

1. **模块导入方式**：熟练掌握四种导入方式（导入整个模块、导入特定元素、给元素/模块起别名），理解不同方式的适用场景（如给模块起短别名简化 AI 库使用，如 `import numpy as np`）。
2. **自定义模块**：掌握自定义模块的创建、导入和测试方法，理解 `if __name__ == "__main__":` 的作用（仅直接执行模块时运行测试代码）。
3. **re 模块基础**：熟悉 `re.search()`、`re.match()`、`re.findall()` 的区别（`match` 仅匹配开头，`search` 全串搜索，`findall` 返回所有匹配），掌握常用正则模式（`^`、`$`、`.`、`*`、`[abc]`）。
4. **第三方包管理**：明确 PIP 和 Conda 的安装命令，知道 conda-forge 等自定义频道的使用场景，理解"安装后才能导入"的逻辑。
5. **概念区分**：能准确区分模块、包、库的定义和层级关系（模块 → 包 → 库）。

三、例题（英文题目 + 中文解析）

1. 选择题（Multiple Choice）

1. Which of the following commands is used to install the Pandas package via Conda from the conda-forge channel? A. `conda install pandas` B. `conda install -c conda-forge pandas` C. `pip install pandas` D. `conda forge install pandas`

    - **Answer**: B
    - **Analysis**: The `-c` parameter in Conda specifies the channel. To install from conda-forge, the command is `conda install -c conda-forge pandas`. Option A uses the default channel; Option C is a PIP command; Option D has incorrect syntax.

2. What is the difference between `re.match()` and `re.search()` in the `re` module? A. `re.match()` searches the entire string, while `re.search()` matches only the start. B. `re.match()` matches only the start of the string, while `re.search()` searches the entire string. C. Both functions return all matches in the string. D. Both functions match only the start of the string.

    - **Answer**: B

- **Analysis**: `re.match()` checks for a match **only at the beginning** of the string; if the pattern is not at the start, it returns `None`. `re.search()` checks for a match **anywhere in the string** (not just the start). Options A, C, D are all incorrect descriptions.

3. Which file is required to mark a directory as a Python package? A. `package.py` B. `__init__.py` C. `module.py` D. `__package__.py`

   - **Answer**: B
   - **Analysis**: A directory must contain an `__init__.py` file (even empty) to be recognized as a Python package. The other options are not standard files for this purpose.

2. 简答题（Short Answer）

1. Explain the purpose of `if __name__ == "__main__":` in a custom module. Give an example.

   - **Answer**: The `__name__` variable is a built-in variable in Python. When a module is **executed directly**, `__name__` is set to `"__main__"`; when the module is **imported into another program**, `__name__` is set to the module's name. This condition ensures that the code inside it runs **only when the module is executed directly** (for testing) and not when imported (to avoid interfering with the main program).
   - Example:

```
# In my_module.py
def add(a, b):
    return a + b

# Test code (runs only when my_module.py is executed directly)
if __name__ == "__main__":
    print("Testing add function:")
    print(f"2 + 3 = {add(2, 3)}")  # Output: 5 (only when running
my_module.py directly)
```

   - When `my_module.py` is imported into `main.py` (e.g., `import my_module`), the test code inside the condition will not run.

2. What are the four common ways to import a module? Give examples for each.

   - **Answer**: The four common import methods are:
     1. **Import the entire module**: Use `import module_name`, access elements via `module_name.element`.
        Example: `import math; print(math.sqrt(16))` (output: 4).
     2. **Import specific elements**: Use `from module_name import element`, access elements directly without the module name.
        Example: `from math import pi; print(pi)` (output: 3.141592653589793).
     3. **Import an element with an alias**: Use `from module_name import element as alias` to simplify long element names.
        Example: `from math import factorial as fact; print(fact(5))` (output: 120).

4. **Import the entire module with an alias**: Use `import module_name as alias` (common for AI libraries with long names).
   Example: `import numpy as np; arr = np.array([1, 2, 3]); print(arr)` (output: [1 2 3]).

Topic 6: Files（文件操作）

一、完整知识点（中英文对照）

1. **File Basics（文件基础）**
   - English: Files are used to store data permanently on storage devices (e.g., hard drives). Python provides built-in functions to create, read, update, and delete files. The core function for file operations is `open()`, which creates a file object.
   - 中文：文件用于在存储设备（如硬盘）上永久存储数据。Python 提供内置函数实现文件的创建、读取、更新和删除。文件操作的核心函数是 `open()`，用于创建文件对象。
2. **The `open()` Function（`open()` 函数）**
   - English:
     - Syntax: `open(file_path, mode)`; returns a file object for subsequent operations.
     - Two key parameters:
       - `file_path`: The path to the file (absolute or relative, e.g., `"myfile.txt"` for the current directory).
       - `mode`: The file operation mode (determines whether to read, write, or append).
   - 中文：
     - 语法：`open(文件路径, 模式)`；返回文件对象，用于后续操作。
     - 两个关键参数：
       - `file_path`：文件路径（绝对路径或相对路径，如当前目录下的文件为 `"myfile.txt"`）。
       - `mode`：文件操作模式（决定读、写、追加等操作类型）。
3. **Common File Modes（常用文件模式）**

模式（Mode）	英文描述	中文描述
`r`	Read mode (default). Opens a file for reading; raises an error if the file does not exist. Stream is positioned at the start.	读模式（默认）。打开文件用于读取；文件不存在则报错。流定位在文件开头。
`w`	Write mode. Truncates the file to zero length (overwrites content) or creates a new file if it does not exist. Stream is positioned at the start.	写模式。将文件截断为零长度（覆盖内容），文件不存在则创建。流定位在文件开头。
`a`	Append mode. Opens a file for appending; creates a new file if it does not exist. Stream is positioned at the end (new content is added to the end).	追加模式。打开文件用于追加；文件不存在则创建。流定位在文件末尾（新内容添加到末尾）。
`r+`	Read and write mode. Opens a file for reading and writing; raises an error if the file does not exist. Stream is positioned at the start.	读写模式。打开文件用于读写；文件不存在则报错。流定位在文件开头。

模式（Mode）	英文描述	中文描述
`w+`	Read and write mode. Truncates the file or creates a new file. Stream is positioned at the start.	读写模式。截断文件或创建新文件。流定位在文件开头。
`a+`	Read and write mode. Opens or creates a file for appending and reading. Stream is positioned at the end for writing; can read the entire file after seeking.	读写模式。打开或创建文件用于追加和读取。流定位在末尾用于写入；可通过定位读取整个文件。

4. **File Reading Methods（文件读取方法）**
   - English:
     - `read()`: Reads the entire file content as a string (optional parameter: number of bytes to read, e.g., `read(10)` reads the first 10 bytes).
       - Example: `f = open("myfile.txt", "r"); print(f.read()); f.close()`
     - `readline()`: Reads one line of the file (ends at the newline character `\n`). Subsequent calls read the next line.
       - Example: `f = open("myfile.txt", "r"); print(f.readline()); print(f.readline()); f.close()`
     - `Loop through lines`: Iterate over the file object directly to read lines one by one (memory-efficient for large files).
       - Example: `f = open("myfile.txt", "r"); for line in f: print(line.strip()); f.close()` (`.strip()` removes newline characters)
   - 中文：
     - `read()`：读取整个文件内容为字符串（可选参数：读取的字节数，如 `read(10)` 读取前 10 个字节）。
       - 示例：`f = open("myfile.txt", "r"); print(f.read()); f.close()`
     - `readline()`：读取文件的一行（到换行符 `\n` 结束）。后续调用读取下一行。
       - 示例：`f = open("myfile.txt", "r"); print(f.readline()); print(f.readline()); f.close()`
     - `循环读取行`：直接迭代文件对象，逐行读取（对大文件内存效率高）。
       - 示例：`f = open("myfile.txt", "r"); for line in f: print(line.strip()); f.close()` （`.strip()` 去除换行符）
5. **File Writing Methods（文件写入方法）**
   - English:
     - `write(content)`: Writes a string to the file. Returns the number of characters written.
       - Example (write mode): `f = open("myfile.txt", "w"); f.write("Hello World"); f.close()` (overwrites existing content)
       - Example (append mode): `f = open("myfile.txt", "a"); f.write("\nNew Line"); f.close()` (adds content to the end)
     - `writelines(lines)`: Writes a list of strings to the file (does not add newline characters automatically; need to include `\n` if needed).
       - Example: `f = open("myfile.txt", "w"); f.writelines(["Line 1\n", "Line 2\n"]); f.close()`
   - 中文：
     - `write(内容)`：将字符串写入文件。返回写入的字符数。

        - 示例（写模式）：`f = open("myfile.txt", "w"); f.write("Hello World"); f.close()`（覆盖现有内容）
        - 示例（追加模式）：`f = open("myfile.txt", "a"); f.write("\nNew Line"); f.close()`（在末尾添加内容）
    - `writelines(行列表)`：将字符串列表写入文件（不会自动添加换行符，需手动加 `\n`）。
        - 示例：`f = open("myfile.txt", "w"); f.writelines(["Line 1\n", "Line 2\n"]); f.close()`

6. **Closing Files（关闭文件）**
    - English:
        - **Explicit close**: Use the `close()` method (e.g., `f.close()`). Important to free memory (file buffer) and ensure content is flushed to the file.
        - **Automatic close**: Use the `with` statement (recommended). The file is automatically closed when exiting the `with` block, even if an error occurs.
            - Example:

```
with open("myfile.txt", "r") as f:
    print(f.read())  # File is automatically closed after this block
```

    - 中文：
        - **显式关闭**：使用 `close()` 方法（如 `f.close()`）。必须关闭以释放内存（文件缓冲区），确保内容写入文件。
        - **自动关闭**：使用 `with` 语句（推荐）。退出 `with` 块时文件自动关闭，即使发生错误也不例外。
            - 示例：

```
with open("myfile.txt", "r") as f:
    print(f.read())  # 此块结束后文件自动关闭
```

7. **Common File Formats（常用文件格式）**
    - **CSV (Comma-Separated Values)**:
        - English: A text format for storing tabular data (each line is a row, values separated by commas). Can be read with Python's built-in `csv` module or the Pandas library (more common in practice).
            - Example (read with `csv` module):

```
import csv
with open("data.csv", "r") as f:
    reader = csv.DictReader(f)  # Reads rows as dictionaries (keys = column names)
    for row in reader:
        print(row["name"], row["age"])  # Access values via column names
```

- 中文：用于存储表格数据的文本格式（每行是一行数据，值用逗号分隔）。可通过 Python 内置 `csv` 模块或 Pandas 库（实际中更常用）读取。
  - 示例（用 `csv` 模块读取）：

```
import csv
with open("data.csv", "r") as f:
    reader = csv.DictReader(f)  # 将行读取为字典（键 = 列名）
    for row in reader:
        print(row["name"], row["age"])  # 通过列名访问值
```

- **JSON (JavaScript Object Notation)**:
  - English: A lightweight data-interchange format (uses key-value pairs, similar to Python dictionaries). Python's built-in `json` module is used to read/write JSON files; Pandas can also process JSON files easily.
    - Example (read JSON file):

```
import json
with open("data.json", "r") as f:
    data = json.load(f)  # Converts JSON content to a Python dictionary
    print(data["name"], data["age"])
```

  - 中文：轻量级数据交换格式（使用键值对，类似 Python 字典）。通过 Python 内置 `json` 模块读写 JSON 文件；Pandas 也可轻松处理 JSON 文件。
    - 示例（读取 JSON 文件）：

```
import json
with open("data.json", "r") as f:
    data = json.load(f)  # 将 JSON 内容转换为 Python 字典
    print(data["name"], data["age"])
```

二、重点掌握内容

1. **`open()` 函数与文件模式**：熟练掌握常用模式（`r`、`w`、`a`、`r+`、`a+`）的区别，尤其是 `w`（覆盖）和 `a`（追加）的差异，避免误删数据。
2. **文件读取方法**：理解 `read()`（读全部）、`readline()`（读一行）、循环读行的适用场景，知道循环读行对大文件更高效。
3. **`with` 语句的使用**：明确 `with` 语句自动关闭文件的优势，优先使用该方式避免内存泄漏。
4. **CSV 与 JSON 处理**：掌握用 `csv.DictReader()` 读取 CSV（按列名访问数据）、用 `json.load()`/`json.dump()` 读写 JSON 的基础语法，理解两种格式的应用场景（CSV 用于表格数据，JSON 用于结构化数据）。
5. **文件操作注意事项**：知道写模式（`w`）会覆盖现有内容，追加模式（`a`）在末尾添加；关闭文件的必要性（显式/自动）。

三、例题（英文题目 + 中文解析）

1. 选择题（Multiple Choice）

1. What happens if you open an existing file with the `"w"` mode in Python? A. The file is opened for reading, and an error is raised if it does not exist. B. The file's content is truncated (overwritten), and a new file is created if it does not exist. C. New content is added to the end of the file. D. The file is opened for reading and writing, and an error is raised if it does not exist.
    - **Answer**: B
    - **Analysis**: The `"w"` (write) mode truncates the existing file to zero length (overwrites all content) if the file exists; if the file does not exist, it creates a new one. Option A describes `"r"` mode; Option C describes `"a"` mode; Option D describes `"r+"` mode.
2. Which method is the most memory-efficient for reading a large text file? A. `read()` B. `readline()` C. Looping through the file object directly D. `readlines()`
    - **Answer**: C
    - **Analysis**: Looping through the file object (e.g., `for line in f:`) reads one line at a time and does not load the entire file into memory, making it the most efficient for large files. `read()` and `readlines()` load the entire file into memory (risk of memory overflow for large files); `readline()` reads one line at a time but requires manual handling of loops, which is less convenient than direct iteration.
3. Which module is used to read and write JSON files in Python? A. `csv` B. `json` C. `pandas` D. `file`
    - **Answer**: B
    - **Analysis**: Python's built-in `json` module provides `load()` (read JSON) and `dump()` (write JSON) functions for JSON file operations. The `csv` module is for CSV files; `pandas` can process JSON but is not a built-in module for core JSON operations; there is no standard `file` module for JSON.

2. 简答题（Short Answer）

1. Explain the difference between the `"w"` and `"a"` modes when opening a file. Give examples for each.
    - **Answer**: The key difference lies in how they handle existing files:
        - `"w"` **(Write mode)**: Overwrites the entire content of an existing file (truncates it to zero length). If the file does not exist, it creates a new file.
          Example: If `myfile.txt` contains "Hello", running `f = open("myfile.txt", "w"); f.write("Hi"); f.close()` will change the file content to "Hi" (original content is lost).
        - `"a"` **(Append mode)**: Adds new content to the **end** of an existing file (does not overwrite existing content). If the file does not exist, it creates a new file.
          Example: If `myfile.txt` contains "Hello", running `f = open("myfile.txt", "a"); f.write(" World"); f.close()` will change the file content to "Hello World" (original content is preserved).
2. How to read a CSV file using the `csv` module and access data by column name? Explain with an example.

- **Answer**: Use the `csv.DictReader()` class from the `csv` module. This class reads each row of the CSV file as a dictionary, where the keys are the column names (from the first row of the CSV) and the values are the corresponding data in the row. This allows accessing data directly by column name.
- Example (assuming `students.csv` has the following content):

```
name,age,grade
Alice,20,A
Bob,19,B
```

- Code to read and access data:

```
import csv

# Open the CSV file with 'with' statement (auto-closes the file)
with open("students.csv", "r") as f:
    # Create a DictReader object (uses first row as column names)
    reader = csv.DictReader(f)

    # Iterate over each row (each row is a dictionary)
    for row in reader:
        # Access data by column name
        print(f"Name: {row['name']}, Age: {row['age']}, Grade:
{row['grade']}")
```

- Output:

```
Name: Alice, Age: 20, Grade: A
Name: Bob, Age: 19, Grade: B
```

Topic 7: Error Handling（错误处理）

一、完整知识点（中英文对照）

1. **Two Types of Errors（两种错误类型）**
   - English:
     - **Syntax Error**: Occurs when the code violates Python's grammar rules (e.g., missing colon after `if`, incorrect indentation). The program cannot run at all until the syntax error is fixed.
     - **Runtime/Logic Error**: Occurs during program execution (even if syntax is correct). Examples include dividing by zero, accessing a non-existent list index, or using an undefined variable. Also called "exceptions".
   - 中文：
     - **语法错误（Syntax Error）**：代码违反 Python 语法规则时发生（如 `if` 后缺少冒号、缩进错误）。语法错误修复前，程序无法运行。

- **运行时/逻辑错误（Runtime/Logic Error）**：语法正确但程序执行时发生的错误（如除以零、访问不存在的列表索引、使用未定义变量）。又称"异常（Exception）"。

2. **Try-Except Statement（Try-Except 语句）**
   - English: The core mechanism for handling runtime errors in Python. It tests a block of code for exceptions and handles them gracefully (avoids program crash).
     - Syntax:

```
try:
    # Code block to test for exceptions (may raise an error)
    risky_code
except ExceptionType1:
    # Code to handle ExceptionType1
    handle_error1
except ExceptionType2:
    # Code to handle ExceptionType2
    handle_error2
except:
    # Code to handle all other uncaught exceptions (general
exception handler)
    handle_all_other_errors
```

     - Example (handle division by zero):

```
try:
    result = 10 / 0  # Risky code (raises ZeroDivisionError)
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")  # Handles the specific
error
```

   - 中文：Python 中处理运行时错误的核心机制。测试可能发生异常的代码块，并优雅地处理错误（避免程序崩溃）。
     - 语法：

```
try:
    # 测试异常的代码块（可能引发错误）
    风险代码
except 异常类型1:
    # 处理异常类型1的代码
    错误处理1
except 异常类型2:
    # 处理异常类型2的代码
    错误处理2
except:
    # 处理所有未捕获的其他异常（通用异常处理器）
    处理所有其他错误
```

- 示例（处理除以零错误）：

```
try:
    result = 10 / 0  # 风险代码（引发 ZeroDivisionError）
except ZeroDivisionError:
    print("错误：不能除以零！")  # 处理特定错误
```

3. **Else Clause（Else 子句）**
   - English: The `else` clause is executed **only if no exceptions are raised** in the `try` block. It separates "normal execution code" from "error handling code".
     - Example:

```
try:
    result = 10 / 2  # No error here
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print(f"Result: {result}")  # Executed (output: Result: 5.0)
```

   - 中文：`else` 子句仅在 `try` 块中**未引发任何异常**时执行。用于区分“正常执行代码”和“错误处理代码”。
     - 示例：

```
try:
    result = 10 / 2  # 此处无错误
except ZeroDivisionError:
    print("不能除以零！")
else:
    print(f"结果：{result}")  # 执行（输出：结果：5.0）
```

4. **Finally Clause（Finally 子句）**
   - English: The `finally` clause is executed **regardless of whether an exception is raised** (whether the `try` block succeeds or fails). Used for "cleanup operations" (e.g., closing files, releasing resources).
     - Example:

```
try:
    f = open("myfile.txt", "r")
    print(f.read())
except FileNotFoundError:
    print("File not found!")
finally:
    # Ensures the file is closed even if an error occurs
    if 'f' in locals():  # Check if 'f' exists
```

```
        f.close()
    print("Cleanup done (file closed if opened)")
```

- 中文：`finally` 子句无论是否引发异常（`try` 块成功或失败），**都会执行。**用于"清理操作"（如关闭文件、释放资源）。
  - 示例：

```
try:
    f = open("myfile.txt", "r")
    print(f.read())
except FileNotFoundError:
    print("文件未找到！")
finally:
    # 确保即使发生错误，文件也会关闭
    if 'f' in locals():  # 检查 'f' 是否存在
        f.close()
    print("清理完成（文件若已打开则已关闭）")
```

5. **Common Built-in Exceptions（常见内置异常）**

异常类型（Exception Type）	英文描述	中文描述
`NameError`	Raised when a variable is not found in local or global scope.	访问未定义的变量时引发。
`IndexError`	Raised when the index of a sequence (list, tuple) is out of range.	访问序列（列表、元组）的索引超出范围时引发。
`KeyError`	Raised when a key is not found in a dictionary.	访问字典中不存在的键时引发。
`ZeroDivisionError`	Raised when dividing by zero (e.g., `5 / 0`).	除以零时引发（如 `5 / 0`）。
`FileNotFoundError`	Raised when trying to open a file that does not exist (e.g., `open("nonexistent.txt", "r")`).	尝试打开不存在的文件时引发（如 `open("nonexistent.txt", "r")`）。
`TypeError`	Raised when an operation is performed on an incompatible data type (e.g., `1 + "2"`).	对不兼容的数据类型执行操作时引发（如 `1 + "2"`）。
`AssertionError`	Raised when an `assert` statement fails (the condition is `False`).	`assert` 语句条件为 `False` 时引发。

6. **Assert Statement（Assert 语句）**
   - English: Used for debugging and validating conditions during development. Syntax: `assert condition, message`. If `condition` is `False`, it raises an `AssertionError` with the optional `message`; if `True`, the program continues.

- Example 1 (validate variable value):

```
age = 15
assert age >= 18, "Age must be at least 18"  # Raises
AssertionError: Age must be at least 18
```

- Example 2 (validate function output):

```
def add(a, b):
    return a + b
# Assert that add(2,3) returns 5
assert add(2, 3) == 5, "add(2,3) should return 5"  # No error
(condition is True)
```

- 中文：用于开发过程中的调试和条件验证。语法：`assert 条件，提示信息`。若条件为 `False`，引发 `AssertionError` 并附带可选提示信息；若为 `True`，程序继续执行。
  - 示例 1（验证变量值）：

```
age = 15
assert age >= 18, "年龄必须至少 18 岁"  # 引发 AssertionError: 年龄
必须至少 18 岁
```

  - 示例 2（验证函数输出）：

```
def add(a, b):
    return a + b
# 验证 add(2,3) 返回 5
assert add(2, 3) == 5, "add(2,3) 应返回 5"  # 无错误（条件为 True）
```

7. **Code Profiling（代码性能分析）**
   - English: A technique to measure program performance (e.g., execution time, CPU usage) to identify bottlenecks. Python's built-in `time` module can be used for basic profiling:
     - `time.time()`: Returns the current wall-clock time (real time) in seconds since the epoch.
     - `time.process_time()`: Returns the CPU time used by the current process (excludes sleep time).
     - Example (measure function execution time):

```
import time

def slow_function():
    time.sleep(1)  # Simulate time-consuming operation (sleeps for
1 second)
    total = sum(range(1000000))
```

```
    return total

# Measure real time and CPU time
start_real = time.time()
start_cpu = time.process_time()

slow_function()

end_real = time.time()
end_cpu = time.process_time()

print(f"Real time: {end_real - start_real:.2f} seconds")  # ~1.00s
(includes sleep)
print(f"CPU time: {end_cpu - start_cpu:.2f} seconds")    # ~0.05s
(excludes sleep)
```

- 中文：用于测量程序性能（如执行时间、CPU 使用率）以定位瓶颈的技术。Python 内置 `time` 模块可用于基础性能分析：
  - `time.time()`：返回从纪元时间到当前的挂钟时间（实时时间），单位为秒。
  - `time.process_time()`：返回当前进程使用的 CPU 时间（不包含休眠时间）。
  - 示例（测量函数执行时间）：

```
import time

def slow_function():
    time.sleep(1)  # 模拟耗时操作（休眠 1 秒）
    total = sum(range(1000000))
    return total

# 测量实时时间和 CPU 时间
start_real = time.time()
start_cpu = time.process_time()

slow_function()

end_real = time.time()
end_cpu = time.process_time()

print(f"实时时间：{end_real - start_real:.2f} 秒")  # ~1.00 秒（包含
休眠）
print(f"CPU 时间：{end_cpu - start_cpu:.2f} 秒")    # ~0.05 秒（不
包含休眠）
```

二、重点掌握内容

1. **错误类型区分**：明确语法错误（无法运行）和运行时错误（执行中发生）的差异，知道运行时错误可通过 Try-Except 处理。
2. **Try-Except 核心用法**：掌握捕获特定异常（如 `ZeroDivisionError`、`FileNotFoundError`）和通用异常的语法，理解“避免程序崩溃”的核心作用。

3. **Else 与 Finally 子句**：区分两者的执行条件（Else 仅无异常时执行，Finally 始终执行），知道 Finally 用于清理操作（如关闭文件）。
4. **常见异常识别**：能识别并对应常见异常的场景（如 `KeyError` 对应字典键不存在，`IndexError` 对应列表索引越界）。
5. **Assert 语句**：理解其调试作用（开发时验证条件），知道条件为 `False` 时引发 `AssertionError`。
6. **基础性能分析**：掌握用 `time.time()` 和 `time.process_time()` 测量执行时间的方法，理解实时时间和 CPU 时间的区别（实时时间包含休眠，CPU 时间仅计算实际运算时间）。

三、例题（英文题目 + 中文解析）

1. 选择题（Multiple Choice）

1. Which clause in the try-except statement is executed regardless of whether an exception is raised? A. `try` B. `except` C. `else` D. `finally`

    - **Answer**: D
    - **Analysis**: The `finally` clause runs every time, whether the `try` block succeeds (no exception) or fails (exception raised). The `try` clause is the code being tested; the `except` clause runs only if an exception is raised; the `else` clause runs only if no exception is raised.

2. Which exception is raised when you try to access a key that does not exist in a dictionary? A. `IndexError` B. `KeyError` C. `NameError` D. `TypeError`

    - **Answer**: B
    - **Analysis**: `KeyError` is specifically for missing dictionary keys. `IndexError` is for out-of-range sequence indexes; `NameError` is for undefined variables; `TypeError` is for incompatible data types.

3. What is the purpose of the `assert` statement in Python? A. To handle runtime errors and prevent program crashes. B. To validate conditions during development; raises an error if the condition is False. C. To automatically fix syntax errors in the code. D. To measure the execution time of a code block.

    - **Answer**: B
    - **Analysis**: The `assert` statement is a debugging tool. It checks if a condition is True; if not, it raises `AssertionError` to alert developers of invalid states. Option A describes try-except; Option C is incorrect (assert does not fix errors); Option D describes code profiling.

2. 简答题（Short Answer）

1. Write a Python code snippet using try-except-else-finally to handle the following scenario: Open a file named "data.txt" for reading, print its content. If the file is not found, print "Error: File not found". If any other error occurs, print "Error: Something went wrong". Ensure the file is closed even if an error occurs, and print "Operation completed" at the end.

    - **Answer**: The code uses try-except to handle specific (FileNotFoundError) and general exceptions, else to print content when no error occurs, and finally to close the file and print the completion message:

```
try:
    # Try to open and read the file
    f = open("data.txt", "r")
except FileNotFoundError:
    # Handle file not found error
    print("Error: File not found")
except Exception:
    # Handle all other errors
    print("Error: Something went wrong")
else:
    # No error: print file content
    print("File content:")
    print(f.read())
finally:
    # Ensure the file is closed if it was opened
    if 'f' in locals():  # Check if 'f' exists (file was opened)
        f.close()
    # Print completion message regardless of error
    print("Operation completed")
```

2. Explain the difference between `time.time()` and `time.process_time()` in the `time` module. Give an example to show their output difference.

   - **Answer**: The two functions measure different types of time:
     - `time.time()`: Measures **wall-clock time** (real time) — the actual time elapsed from the start to the end of an operation, including time spent sleeping (e.g., `time.sleep()`) or waiting for other processes.
     - `time.process_time()`: Measures **CPU time** — the time the CPU actually spends executing the code of the current process, excluding sleep time or waiting time.
   - Example (with a 1-second sleep):

```
import time

def example_function():
    time.sleep(1)  # Simulate waiting (sleeps for 1 second, no CPU
usage)
    sum(range(1000000))  # CPU-intensive operation

# Record start times
start_real = time.time()
start_cpu = time.process_time()

example_function()

# Record end times
end_real = time.time()
end_cpu = time.process_time()

# Calculate and print differences
print(f"Real time elapsed: {end_real - start_real:.2f} seconds")  #
```

```
    ~1.05s (includes 1s sleep)
    print(f"CPU time elapsed: {end_cpu - start_cpu:.2f} seconds")    #
    ~0.01s (only CPU work)
```

- Output difference: Real time is much longer because it includes the 1-second sleep, while CPU time is short (only the time spent summing numbers).