Python核心面试

第一部分：数据类型与底层实现 (★★★★★)

1.1 列表与元组

Q1: (★★★★★) 深入对比 list 和 tuple：除了可变性，它们在内存分配、缓存机制和性能上还有哪些差异?

- **内存分配**:
  - `tuple`: 元组是不可变的，其大小在创建时就已确定。因此，Python 会一次性为元组分配一块连续的内存。
  - `list`: 列表是可变的，它是一个动态数组。除了存储元素的内存外，它还额外存储了已分配容量、元素数量等信息。由于需要支持动态扩容，其内存管理比元组复杂。
- **缓存机制**:
  - CPython 对一些小的、不可变的对象（如短字符串、小整数）有缓存机制。虽然对元组没有明确的全局缓存，但由于其不可变性，编译器可以在编译时进行一些优化，例如在代码块内重复使用同一个元组常量。列表因为是可变的，无法被缓存。
- **性能**:
  - **创建**: 创建元组通常比创建列表稍快，因为它不需要考虑额外的容量分配。
  - **访问**: 两者访问元素的时间复杂度都是 O(1)，性能上没有显著差异。
  - **整体**: 由于不可变性带来的优化和更紧凑的内存结构，元组在整体上被认为是比列表更"轻量级"的数据结构。

Q2: (★★★★☆) 列表的 `append` 和 `extend` 有何区别？`+` 操作符和 `extend` 在性能上有何差异?

- `append` **vs** `extend`:
  - `list.append(x)`: 将参数 `x` 作为一个**单一元素**添加到列表末尾。列表长度增加 1。
  - `list.extend(iterable)`: 将一个**可迭代对象** `iterable` 中的所有元素逐个添加到列表末尾。
- `+` **vs** `extend` **性能**:
  - `+`: 会创建一个**全新的列表**对象，并将两个原始列表的元素复制到新列表中。这涉及额外的内存分配和数据复制，效率较低，尤其是在循环中。
  - `extend`: **就地修改**原始列表，将新元素直接追加到末尾。它不创建新列表，因此内存效率和时间效率都更高。

Q3: (★★★★☆) CPython 中 list 的动态数组是如何实现扩容的？其增长因子是多少，为何这样设计?

- **实现方式**: 当列表的容量不足以添加新元素时，CPython 会申请一块更大的内存空间，然后将所有旧元素从旧地址复制到新地址，最后释放旧的内存空间。
- **增长策略**: 列表的增长不是按固定倍数（如2倍）进行的，而是一种平滑的增长策略。其近似公式为：`new_allocated = old_allocated + (old_allocated >> 3) + (3 if old_allocated < 9 else 6)`。
- **设计原因**: 这种"超额分配"（over-allocating）的策略旨在**摊销**（amortize）内存重分配的成本。虽然单次扩容成本较高（O(n)），但它确保了连续多次 `append` 操作的平均时间复杂度为 **O(1)**。相比于简单的倍增策略，这种方式在保证效率的同时，也试图避免在列表较小时浪费过多内存。

Q4: (★★★☆☆) `list.sort()` 和 `sorted()` 函数的根本区别是什么？它们是稳定排序吗？

- **根本区别**:
  - `list.sort()`: 是列表的**就地（in-place）方法**。它会直接修改原始列表，并且返回值为 `None`。
  - `sorted(iterable)`: 是一个**内置函数**。它会返回一个**新的排好序的列表**，而原始的可迭代对象保持不变。
- **稳定性**: 两者都是**稳定排序**（Stable Sort）。这意味着如果两个元素有相等的键（或值），它们在排序后的相对顺序将与输入时保持一致。CPython 内部使用 Timsort 算法来实现。

Q5: (★★★☆☆) 如何高效地反转一个列表?

1. `list.reverse()`: 就地反转，直接修改原列表。空间复杂度 O(1)，是内存效率最高的方式。
2. **切片** `[::-1]`: `new_list = my_list[::-1]`。创建一个新的反转后的列表（浅拷贝）。代码简洁，是 Pythonic 的写法。
3. `reversed(list)`: 返回一个反向的**迭代器**（iterator），而不是列表。它非常节省内存，当你只需要遍历反转后的列表而不需要一个完整的列表副本时，这是最佳选择。

1.2 字典与集合

Q6: (★★★★★) 剖析 dict 的底层实现：哈希冲突的解决方法是什么（开放寻址法）？dict 的扩容机制是怎样的?

- **底层实现**: 字典和集合的底层都是基于**哈希表**（Hash Table）或称散列表。
- **哈希冲突解决**: CPython 使用**开放寻址法**（Open Addressing）。当发生哈希冲突时（即两个 key 计算出相同的哈希槽位），它会根据一个探测序列去寻找下一个可用的空槽位来存放元素。相比于拉链法（用链表存储冲突元素），开放寻址法缓存友好性更好。
- **扩容机制**: 当字典中元素的数量达到总容量的 **2/3** 时，会触发扩容。Python 会创建一个更大的哈希表（通常是原容量的2倍或4倍，取决于当前大小），然后将所有旧表中的键值对重新计算哈希值，并插入到新的哈希表中。这个过程称为 **rehash**。

Q7: (★★★★★) Python 3.7+ 的字典是如何实现有序的？这种"紧凑字典"的实现方式对内存和性能有何具体影响?

- **实现方式**: 从 Python 3.7 开始（实际上从 3.6 就已实现，但 3.7 才成为语言规范），字典保证维护元素的**插入顺序**。这是通过一个分离的、紧凑的数组（`dk_entries`）来实现的。
  1. 一个稀疏的**索引数组**（`dk_indices`）用于哈希查找。
  2. 一个紧凑的条目数组（dk_entries）按插入顺序存储 (hash, key, value)。

     索引数组中的值是指向条目数组的索引。
- **具体影响**:
  - **内存**: 这种"紧凑字典"比旧式字典**更节省内存**，因为它消除了稀疏哈希表中的大量空槽，迭代也更高效。
  - **性能**: 迭代字典的速度变快了，因为它只需遍历那个紧凑的条目数组。插入和删除操作的性能基本保持不变。

Q8: (★★★★☆) 为什么字典的键必须是可哈希的？如何让一个自定义类的实例变得可哈希?

- **原因**: 字典通过键的哈希值来决定其在哈希表中的存储位置，以实现 O(1) 的平均查找速度。一个可哈希的对象必须满足两个条件：
  1. 实现了 `__hash__()` 方法，返回一个整数。

2. 实现了 `__eq__()` 方法，用于比较两个对象是否相等。
3. **核心原则**: 如果 `a == b`，那么必须保证 `hash(a) == hash(b)`。同时，对象的哈希值在其生命周期内必须**不可变**。可变对象（如列表）的哈希值会随内容变化而变化，因此不能作为键。

- **自定义类可哈希**: 要让一个类的实例可哈希，必须同时实现 `__hash__` 和 `__eq__` 方法。通常，哈希值应该基于实例中那些不可变的属性来计算。

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        return isinstance(other, Person) and self.name == other.name and self.age
== other.age

    def __hash__(self):
        # 基于不可变属性计算哈希值
        return hash((self.name, self.age))
```

Q9: (★★★★☆) Python 中 set 的底层实现是什么？add 和 remove 操作的时间复杂度是多少?

- **底层实现**: `set` 的底层实现与 `dict` 非常相似，也是一个**哈希表**。你可以把它看作一个只有键而没有值的特殊字典。
- **时间复杂度**: 由于基于哈希表，`add`（添加）、`remove`（移除）和 `in`（成员检查）的平均时间复杂度都是 **O(1)**。最坏情况下的时间复杂度为 O(n)，但这在实际中极为罕见。

Q10: (★★★☆☆) `dict.keys()`, `dict.values()`, `dict.items()` 返回的是什么对象？它们与列表有何不同?

- **返回对象**: 它们返回的是**字典视图对象**（Dictionary View Objects）。
- **与列表的不同**:
    1. **动态性**: 视图对象是动态的，它们会**实时反映**字典内容的变化。如果字典被修改，视图会立即体现这些修改。而将它们转换为列表（如 `list(d.keys())`）则会创建一个静态的副本。
    2. **类集合操作**: `keys()` 和 `items()` 视图支持类集合操作，如 `&` (交集), `|` (并集) 等。
    3. **内存效率**: 视图不复制数据，只是提供一个窗口来查看数据，因此比创建列表副本更节省内存。

Q11: (★★★☆☆) `collections.defaultdict` 和 `dict.setdefault()` 的使用场景和区别是什么?

- `collections.defaultdict`:
    - **是什么**: `dict` 的子类，在构造时接收一个**默认工厂函数**（如 `list`, `int`）。
    - **行为**: 当访问一个**不存在的键**时，它会自动调用工厂函数创建一个默认值，将该值与键关联，并返回这个值。
    - **场景**: 非常适合用于对元素进行分组或计数，代码更简洁。
- `dict.setdefault(key, default)`:
    - **是什么**: 标准 `dict` 的一个方法。
    - **行为**: 如果 `key` 存在，返回其值。如果不存在，则设置 `dict[key] = default`，并返回 `default`。
- **核心区别**:

- `defaultdict` 在键不存在时才调用工厂函数。
- `setdefault` **无论键是否存在，都会对** `default` **参数进行求值**，这在 `default` 是一个昂贵函数调用时可能造成性能浪费。

Q12: (★★★☆☆) `collections.OrderedDict` **在现代Python版本中还有使用价值吗?**

有，但在特定场景下。

- **背景**: 从 Python 3.7+ 开始，标准 `dict` 已保证维护插入顺序。
- **剩余价值**:
  1. **对顺序敏感的等价判断**: `OrderedDict` 的 `==` 比较会同时检查键值对和**顺序。**而标准 `dict` 的 `==` 只关心键值对，不关心顺序。
  2. **专有方法**: `OrderedDict` 拥有 `move_to_end()` 和 `popitem(last=...)` 等用于重新排序的专有方法，这对于实现 LRU 缓存等算法非常有用。
  3. **向后兼容**: 需要兼容 Python 3.6 及更早版本的代码。

1.3 字符串

Q13: (★★★★☆) 什么是字符串驻留 (string interning) 机制？它对性能和内存有什么影响?

- **机制**: 字符串驻留是一种优化技术，它会维护一个字符串池。对于池中已存在的字符串，Python 不会创建新对象，而是返回池中对象的引用。这确保了内容相同的不可变字符串在内存中只有一份拷贝。
- **影响**:
  - **内存**: 极大地节省了内存，特别是当程序中存在大量重复的短字符串时。
  - **性能**: 字符串比较的性能得到提升。对于驻留的字符串，可以直接使用 `is` 进行身份比较（比较内存地址），这比使用 `==` 进行逐字符比较要快得多。
- **注意**: Python 会自动驻留一些符合标识符规则的短字符串。可以使用 `sys.intern()` 强制驻留一个字符串。

Q14: (★★★☆☆) f-string, str.format(), % 操作符格式化字符串，三者在性能和用法上有何区别?

- `%` **操作符**: 最古老的方式。用法简单，但功能有限，类型匹配不当时容易出错。
- `str.format()`: 更强大、更灵活。支持命名参数、位置参数和更复杂的格式化说明符。
- **f-string (Formatted String Literals)**: Python 3.6+ 引入。**性能最高、用法最简洁。**它在运行时直接将表达式嵌入到字符串中，避免了函数调用的开销。是目前推荐的首选方式。
- **性能排序**: **f-string > str.format() > %**

Q15: (★★★☆☆) 如何高效地拼接大量字符串？为什么不推荐在循环中使用 `+` **操作?**

- **高效方法**: 将所有待拼接的字符串放入一个列表，然后调用 `''.join(list_of_strings)`。
- **不推荐** `+` **的原因**: 字符串是**不可变**的。在循环中使用 `+`（如 `s += part`）每次都会创建一个**新的字符串对象**，并将旧字符串和新部分的内容复制到新对象中。如果循环 N 次，其时间复杂度会达到 $O(N^2)$，并产生大量无用的中间对象，造成巨大的内存和性能开销。而 `''.join()` 只会进行一次内存分配和一次数据复制，时间复杂度为 O(N)。

1.4 拷贝机制

Q16: (★★★★★) `b = a`, `b = a[:]`, `b = copy.copy(a)`, `b = copy.deepcopy(a)` 四者在处理嵌套可变对象时有何根本区别?

- `b = a` **(赋值)**:
  - **本质**: 引用赋值。`b` 和 `a` 指向**同一个内存地址**。
  - **效果**: 对 `b` 的任何修改都会影响 `a`，反之亦然。`id(a) == id(b)`。
- `b = a[:]` **或** `b = copy.copy(a)` **(浅拷贝)**:
  - **本质**: 创建一个新的顶层对象，但其内部元素是原始对象中元素的**引用**。
  - **效果**: `b` 是一个新列表，`id(a) != id(b)`。但如果列表内包含可变对象（如子列表），修改 `b[i]` 这个子列表的内容，`a[i]` 的内容也会跟着改变，因为它们指向同一个子列表对象。
- `b = copy.deepcopy(a)` **(深拷贝)**:
  - **本质**: 完全独立地复制。它会**递归地**复制原始对象及其包含的所有子对象。
  - **效果**: 创建一个与 `a` 完全隔离的新对象。修改 `b` 或其任何子对象，都不会对 `a` 产生任何影响。

Q17: (★★★☆☆) `copy.deepcopy` 是如何处理循环引用的?

- `deepcopy` 内部维护一个 `memo` 字典，用于在递归复制过程中**记录已经复制过的对象。**
- **机制**:
  1. 在复制一个对象前，`deepcopy` 会先检查该对象的 `id` 是否已存在于 `memo` 字典中。
  2. 如果存在，说明遇到了循环引用，它会直接返回 `memo` 中已存的对应副本，而不再进行新的复制，从而避免无限递归。
  3. 如果不存在，它会正常创建新副本，并将 `(原始对象id, 新副本)` 这个键值对存入 `memo` 字典，以备后续检查。

补充核心问题

Q18(补): CPython 中小整数对象池和短字符串驻留是如何工作的?

- 为了性能和内存优化，CPython 会预先创建并缓存一些常用的不可变对象。
- **小整数对象池**: 默认情况下，整数 `[-5, 256]` (包含-5和256) 会被缓存。程序中任何对这个范围内整数的引用都会指向同一个预先创建好的对象。因此 `a = 100; b = 100`，`a is b` 为 `True`。但 `a = 257; b = 257`，`a is b` 通常为 `False`。
- **短字符串驻留**: (见 Q13)

Q19(补): `None` 是什么? 为什么 `x is None` 是首选的判断方式?

- `None` **是什么**: `None` 是 `NoneType` 类型的**唯一实例**，它是一个**单例**（Singleton）对象，表示值的缺失。
- **为什么用** `is`:
  1. **速度**: `is` 是身份运算符，直接比较两个对象的内存地址 (`id`)，速度极快。
  2. **可靠性**: 因为 `None` 是单例，所有值为 `None` 的变量都指向同一个对象，所以 `is` 判断永远准确。
  3. **安全性**: `==` 是相等性运算符，它会调用对象的 `__eq__` 方法。某些对象可能会重载 `__eq__` 方法，导致 `obj == None` 的结果不符合预期。而 `is` 不会被重载，结果更可靠。

第二部分：函数、闭包与装饰器 (★★★★★)

Q18: (★★★★★) 什么是闭包？闭包的三个要素是什么？如何解决在循环中创建闭包时的"延迟绑定"问题?

- **什么是闭包**: 闭包（Closure）是指一个函数对象，它记住了其定义时的环境（即包含该函数的外部作用域）。即使外部函数已经执行完毕，闭包仍然可以访问和操作其外部作用域中的变量。
- **闭包三要素**:
  1. **嵌套函数**：必须有一个内部函数。
  2. **引用外部变量**：内部函数必须引用其外部函数作用域中的变量。
  3. **返回内部函数**：外部函数必须返回那个内部函数。
- **延迟绑定问题 (Late Binding)**: 在循环中创建闭包时，闭包内的变量并不会在定义时立即绑定当前循环的值，而是在闭包被**调用时**才去查找其作用域中的变量值。此时循环已经结束，所有闭包都会引用到循环变量的**最终值**。
- **解决方法**: 利用函数参数的默认值。默认参数的值在函数**定义时**就被计算和绑定了。

```
# 错误示例：所有函数都返回 9
def create_multipliers_wrong():
    return [lambda x: i * x for i in range(10)]

# 正确解法：使用默认参数
def create_multipliers_correct():
    # 在定义 lambda 时，i 的值被立即绑定到默认参数 n
    return [lambda x, n=i: n * x for i in range(10)]
```

Q19: (★★★★★) 手写一个能接收参数的装饰器，并解释其执行流程。

- **手写代码**:

```
import functools

def logger(level):  # 1. 接收装饰器参数
    def decorator(func):  # 2. 接收被装饰的函数
        @functools.wraps(func)
        def wrapper(*args, **kwargs):  # 3. 包装函数，接收原始函数参数
            print(f"[{level}] Calling {func.__name__}...")
            result = func(*args, **kwargs)
            print(f"[{level}] {func.__name__} finished.")
            return result
        return wrapper
    return decorator

@logger(level="INFO")
def say_hello(name):
    print(f"Hello, {name}!")

say_hello("World")
```

- **执行流程**:
  1. `@logger(level="INFO")`: 首先调用最外层的 `logger()` 函数，并传入参数 `"INFO"`。
  2. `logger("INFO")` 返回中间层的 `decorator` 函数。

3. Python 将被装饰的 `say_hello` 函数作为参数传递给上一步返回的 `decorator` 函数，即执行 `decorator(say_hello)`。
4. `decorator(say_hello)` 返回最内层的 `wrapper` 函数。
5. `say_hello` 这个名字现在指向了返回的 `wrapper` 函数。
6. 当调用 `say_hello("World")` 时，实际上是在执行 `wrapper("World")`。

Q20: (★★★★★) `functools.wraps` 在装饰器中的作用是什么？如果不用它会有什么具体问题?

- **作用**: `@functools.wraps(func)` 是一个装饰器，它的作用是将**原始函数 `func` 的元信息**（如 `__name__`、`__doc__`、`__annotations__` 等）复制到包装函数 `wrapper` 上。
- **具体问题**: 如果不使用 `@wraps`，被装饰后的函数其元信息会变成包装函数 `wrapper` 的信息。这会导致：
  1. **函数名和文档丢失**: `say_hello.__name__` 会变成 `'wrapper'`，`say_hello.__doc__` 会变成 `None`。
  2. **调试困难**: 调试和内省工具（如 `help()`、pdb）看到的是包装函数的信息，而不是我们真正想了解的原始函数。
  3. **签名不匹配**: 一些依赖函数签名的库（如 `inspect`）可能无法正常工作。

Q21: (★★★★☆) 深入解释 `nonlocal` 和 `global` 关键字的区别和使用场景。

- `global`:
  - **作用**: 声明一个变量为**全局变量**。它告诉 Python，在当前作用域内对该变量的赋值操作，应该直接修改**模块顶层**的那个同名变量。
  - **使用场景**: 当你需要在函数内部修改一个定义在模块最外层的全局变量时使用。
- `nonlocal`:
  - **作用**: 声明一个变量为**非局部变量**。它用于在嵌套函数中，修改**最近的（非全局的）外层函数作用域**中的变量。
  - **使用场景**: 主要用于闭包中，当内部函数需要修改外部（但非全局）函数的变量时使用。
- **核心区别**: `global` 直接跳到最外层的全局作用域，而 `nonlocal` 只向外层查找一层（或多层，直到找到非全局的那个）。

Q22: (★★★★☆) Python 的函数参数是如何传递的？（是值传递、引用传递还是共享传参/对象引用传递？）

- **标准答案**: Python 的参数传递方式是**共享传参**（Pass by Sharing）或称**对象引用传递**（Pass by Object Reference）。
- **具体解释**:
  1. 函数接收到的参数是原始对象的**引用**（内存地址的副本）。
  2. **如果对象是不可变的 (如 `int`, `str`, `tuple`)**: 在函数内部对参数进行重新赋值，实际上是创建了一个新对象，并将函数内的局部变量名指向这个新对象，这**不会影响**到函数外部的原始对象。这看起来像“值传递”。
  3. **如果对象是可变的 (如 `list`, `dict`)**: 在函数内部修改该对象的内容（如 `list.append()`），会直接修改原始对象，因为内外都引用着同一个对象。这看起来像“引用传递”。
  4. **关键**: 无论对象可变与否，如果在函数内部对参数名本身进行**重新赋值**（`arg = new_value`），都只是改变了函数内局部变量的指向，**永远不会改变**调用者作用域中原始变量的指向。

Q23: (★★★★☆) 如何实现一个类装饰器？它和函数装饰器有何不同？

- **实现方式**: 类装饰器是一个实现了 `__init__` 和 `__call__` 方法的类。
  - `__init__`：接收被装饰的函数作为参数，并将其存储起来。
  - `__call__`：实现包装逻辑，当被装饰后的函数被调用时，`__call__` 方法会被执行。

```
class Counter:
    def __init__(self, func):
        self.func = func
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        print(f"'{self.func.__name__}' has been called {self.count} times.")
        return self.func(*args, **kwargs)

@Counter
def greet():
    print("Hello!")
```

- **与函数装饰器的不同**:
  - **状态管理**: 类装饰器可以利用实例属性（如上例的 `self.count`）来**保存状态**，这对于需要跨多次调用跟踪信息的场景（如计数、缓存）非常方便。函数装饰器要实现同样功能通常需要借助闭包或 `nonlocal`。
  - **代码组织**: 对于复杂的装饰逻辑，使用类可以更好地组织代码，使其更符合面向对象的原则。

Q24: (★★★☆☆) `\*args` 和 `\**kwargs` 在函数定义和函数调用中有什么区别和用法?

- **在函数定义中 (打包/Packing)**:
  - `*args`: 将所有未匹配的**位置参数**打包成一个**元组 (tuple)**。
  - `**kwargs`: 将所有未匹配的**关键字参数**打包成一个**字典 (dict)**。
  - **用法**: 用于创建能接收任意数量参数的灵活函数，常见于装饰器和高阶函数。
- **在函数调用中 (解包/Unpacking)**:
  - `*iterable`: 将一个可迭代对象（如列表、元组）解包成独立的**位置参数**。
  - `**mapping`: 将一个字典解包成独立的**关键字参数**。
  - **用法**: 用于将已有的序列或字典作为参数传递给另一个函数。

Q25: (★★★☆☆) `lambda` 表达式有什么优点和局限性?

- **优点**:
  1. **简洁**: 无需 `def` 和 `return` 关键字，可以在一行内定义简单的函数。
  2. **匿名**: 它是一个匿名函数，不需要为其命名，非常适合作为高阶函数（如 `map`, `filter`, `sorted`）的参数。
- **局限性**:
  1. **单一表达式**: `lambda` 的函数体只能是一个**单一的表达式**，不能包含多行语句、赋值、循环等。
  2. **可读性差**: 对于稍微复杂的逻辑，使用 `lambda` 会降低代码的可读性，此时应使用常规的 `def` 函数。

Q26: (★★★☆☆) 什么是偏函数 (Partial Function)？`functools.partial` 的应用场景是什么？

- **什么是偏函数**: 偏函数是指通过"冻结"一个函数的部分参数来创建一个新的、更简单的函数。
- `functools.partial`: 它是创建偏函数的主要工具。`partial(func, *args, **keywords)` 会返回一个新的可调用对象，该对象被调用时，会如同调用 `func` 一样，但预先填充了 `partial` 中提供的 `args` 和 `keywords`。
- **应用场景**:
  1. **简化函数签名**: 当你需要频繁调用一个具有很多参数的函数，且其中一些参数总是固定的，可以用 `partial` 创建一个简化版。
  2. **适配接口**: 当一个 API 需要一个特定签名的回调函数时，可以用 `partial` 来适配一个已有但不完全匹配的函数。

```
from functools import partial
# 创建一个总是计算2的次方的函数
power_of_two = partial(pow, 2)
print(power_of_two(5))  # 相当于 pow(2, 5)，输出 32
```

Q27: (★★★☆☆) 函数注解 (Function Annotations) 的作用是什么？它在运行时有何影响？

- **作用**: 函数注解是一种为函数参数和返回值附加**元数据**的方式，其主要用途是**类型提示 (Type Hints)**。
- **运行时影响**:
  1. **无直接影响**: Python 解释器在运行时**完全忽略**函数注解，它们不会自动进行类型检查或转换。代码的执行逻辑不会因注解而改变。
  2. **可供访问**: 注解信息存储在函数的 `__annotations__` 属性（一个字典）中，可以在运行时通过内省来访问和使用。
  3. **工具利用**: 静态类型检查工具（如 `mypy`）、IDE 和一些库（如 `FastAPI`, `Pydantic`）会利用这些注解来提供代码补全、错误检查、数据验证和序列化等功能。

第三部分：面向对象编程 (OOP) 深度剖析 (★★★★★)

Q28: (★★★★★) 什么是描述符 (Descriptor) 协议？`property`, `staticmethod`, `classmethod` 是如何基于描述符实现的？

- **描述符协议**: 一个实现了 `__get__`, `__set__`, 或 `__delete__` 中至少一个方法的对象，就是描述符。它允许对象自定义当其作为另一个类（属主类）的属性被访问时的行为。
  - `__get__(self, instance, owner)`: 当访问属性时调用。
  - `__set__(self, instance, value)`: 当设置属性时调用。
  - `__delete__(self, instance)`: 当删除属性时调用。
- **实现原理**:
  - `property`: 是一个用 Python 实现的描述符。`property(fget, fset, fdel)` 将 getter, setter, deleter 函数包装起来，当对属性进行访问、赋值、删除时，会分别触发这几个函数。
  - `staticmethod`: `__get__` 方法直接返回原始函数，不绑定 `instance` 或 `owner`，从而使其行为像一个普通函数。
  - `classmethod`: `__get__` 方法将 `owner` (即类本身) 作为第一个参数绑定到原始函数上，返回一个绑定方法，从而实现了将类作为第一个参数 `cls` 传递的效果。

Q29: (★★★★★) 什么是元类 (Metaclass)？请解释一个实际的应用场景（如 ORM 或 API 框架）。

- **什么是元类**: 在 Python 中，一切皆对象。类也是对象，而元类就是**创建类的"类"**。`type` 是 Python 默认的元类。你可以通过自定义元类来拦截和修改类的创建过程。
- **应用场景 (ORM)**:
  - 在 Django 或 SQLAlchemy 等 ORM 框架中，我们定义一个模型类来映射数据库表：

    ```
    class User(models.Model):
        name = models.CharField(max_length=50)
        age = models.IntegerField()
    ```

  - 这里的 `models.Model` 就使用了一个元类（如 `ModelBase`）。
  - **工作流程**: 当 `User` 类被定义时，元类会启动。它会遍历类中定义的属性（`name`, `age`），将这些声明式的字段（`CharField`, `IntegerField`）转换成真正的数据库列描述，并自动为 `User` 类添加数据库操作方法（如 `.save()`, `.delete()`, `.objects.all()`）。
  - **核心作用**: 元类实现了从**声明式代码**到**完整功能对象**的自动转换，极大地简化了开发者的工作。

Q30: (★★★★★) 深入解释 `__new__` 和 `__init__` 的区别。在什么场景下必须重写 `__new__`？

- **区别**:
  - `__new__` **(构造器)**:
    - 是一个**类方法**（即使没有用 `@classmethod` 装饰）。
    - 它的**第一个参数是类** `cls`。
    - 它的作用是**创建并返回一个实例对象**。
    - 在 `__init__` 之前被调用。
  - `__init__` **(初始化器)**:
    - 是一个**实例方法**。
    - 它的**第一个参数是实例** `self`（由 `__new__` 返回）。
    - 它的作用是**初始化已经创建好的实例**，通常是给实例赋属性值。
    - 它不应该返回任何东西（除了 `None`）。
- **必须重写** `__new__` **的场景**:
  1. **实现单例模式**: 在 `__new__` 中控制，确保一个类只有一个实例。
  2. **继承不可变类型**: 当你需要继承一个不可变类型（如 `str`, `int`, `tuple`）并想在创建时修改其行为时。因为 `__init__` 被调用时实例已经创建完毕，对于不可变类型为时已晚。

Q31: (★★★★☆) Python 的方法解析顺序 (MRO) 是什么？C3 线性化算法是如何解决菱形继承问题的？

- **MRO (Method Resolution Order)**: MRO 是在多重继承中，一个类在调用方法时，Python 解释器搜索其基类的顺序列表。可以通过 `ClassName.mro()` 或 `ClassName.__mro__` 查看。
- **C3 线性化算法**: Python 使用 C3 线性化算法来计算 MRO。该算法保证了 MRO 的两个关键特性：
  1. **单调性**: 子类的 MRO 总是在其父类的 MRO 之后保留父类的顺序。
  2. **一致性**: 所有父类的 MRO 顺序都被保留。

- **解决菱形继承**: 在菱形继承（一个类同时继承自两个有共同基类的类）中，C3 算法能确保共同的基类只在 MRO 中出现一次，并且位置在所有派生自它的子类之后，从而避免了方法调用的歧义和无限递归。

Q32: (★★★★☆) Python 中 `super()` 的工作原理是什么？尤其是在多重继承中。

- **工作原理**: `super()` 返回一个**代理对象**，该对象会将方法调用委托给 MRO 中当前类或实例的**下一个**类。
- **关键点**:
  1. `super()` **不是**简单地调用父类的方法。它查找的是 MRO 列表中的下一个类。
  2. `super(MyClass, self).method()` 的意思是："在 `MyClass` 的 MRO 中，找到 `self` 这个实例所属的类的下一个类，并调用它的 `method` 方法"。
  3. 在多重继承中，`super()` 确保了继承链上的每个类的 `__init__` (或其他方法) 都会被**有且仅有一次**地调用（只要所有类都使用了 `super()`），形成了一条协作链。

Q33: (★★★★☆) `__getattr__` 和 `__getattribute__` 方法有什么核心区别和不同的应用场景?

- `__getattribute__(self, name)`:
  - **调用时机**: **无条件地**拦截每一次对实例属性的访问（点号 `.` 操作）。
  - **风险**: 必须非常小心地实现，为了避免无限递归，通常需要在其内部使用 `super().__getattribute__(name)` 来访问属性。
  - **应用场景**: 代理模式、实现需要完全控制属性访问权限的安全层。
- `__getattr__(self, name)`:
  - **调用时机**: 仅当对一个**不存在的属性**进行访问时，才会作为**最后的备选方案**被调用。
  - **应用场景**: 实现属性的动态生成、将属性访问重定向到其他对象（如包装器）。

Q34: (★★★★☆) `__slots__` 的作用是什么？它的实现原理是什么？使用它会带来哪些实际的限制?

- **作用**:
  1. **节省内存**: 告诉 Python 不要为实例创建 `__dict__`，而是使用一个类似元组的固定结构来存储实例属性。这对于需要创建大量实例的场景可以显著减少内存占用。
  2. **加快属性访问速度**: 访问 `__slots__` 中的属性比访问 `__dict__` 中的属性要稍快一些。
- **实现原理**: `__slots__` 是一个类变量，通常是一个字符串元组，列出了所有合法的实例属性名。
- **限制**:
  1. **无法动态添加属性**: 实例不能再拥有 `__slots__` 中未声明的属性。
  2. **继承问题**: 子类只有在也定义了 `__slots__` 的情况下才会继承父类的内存优化效果。如果子类定义了 `__dict__` 或没有定义 `__slots__`，则会同时拥有 `__slots__` 和 `__dict__`。
  3. **不支持弱引用**: 除非在 `__slots__` 中显式加入 `'__weakref__'`。

Q35: (★★★★☆) 什么是抽象基类 (ABC)？它与"鸭子类型"是什么关系?

- **抽象基类 (ABC)**: ABC (Abstract Base Class) 是一种特殊的类，它不能被实例化，其主要目的是定义一个标准的接口（一组方法和属性），并强制其所有子类都必须实现这个接口。通过 `abc` 模块和 `@abstractmethod` 装饰器实现。
- **鸭子类型 (Duck Typing)**: "如果它走起来像鸭子，叫起来也像鸭子，那么它就是一只鸭子。" 这是 Python 的一种编程哲学，即不关心对象的类型，只关心对象是否具有所需的方法和属性。
- **关系**:

- ABC 可以被看作是**对鸭子类型的形式化和补充**。
- 鸭子类型是隐式的、非正式的。
- ABC 提供了**显式的、正式的**接口定义。你可以使用 `isinstance()` 来检查一个对象是否遵循了某个 ABC 的接口，即使它没有直接继承自该 ABC（通过 `__subclasshook__` 实现），这让鸭子类型变得更加健壮和明确。

Q36: (★★★☆☆) 如何在Python中优雅地实现单例模式？（至少说出三种方法）

1. **使用 `__new__`**: 重写 `__new__` 方法，在类中保存一个实例。每次调用 `__new__` 时，检查实例是否存在，如果存在则直接返回，否则创建并保存。
2. **使用装饰器**: 创建一个装饰器，它内部维护一个字典来存储类的实例。当装饰的类被第一次实例化时，创建实例并存入字典，后续再实例化时直接从字典返回。
3. **使用元类**: 创建一个元类，重写其 `__call__` 方法。`__call__` 方法负责类的实例化过程，可以在这里控制实例的创建逻辑。
4. **模块级单例**: Python 模块在第一次导入时会被执行并缓存。因此，在一个模块中创建实例，然后在其他地方导入这个实例，天然就是单例的。

Q37: (★★★☆☆) `__` (双下划线) 引发的名称改写 (Name Mangling) 是为了解决什么问题?

- **解决的问题**: 名称改写主要是为了**避免在子类中意外地覆盖父类的"私有"属性或方法**。
- **机制**: 当一个属性名以两个或更多下划线开头，且结尾最多只有一个下划线时（如 `__spam`），Python 解释器会自动将其名称改写为 `_ClassName__spam` 的形式。
- **目的**: 这不是真正的私有化（因为你仍然可以通过改写后的名字访问它），而是一种**命名冲突的避免机制**，特别是在复杂的继承体系中非常有用。

Q38: (★★★☆☆) Python 中的 Mixin 模式是什么？使用它有什么优缺点?

- **Mixin 模式**: Mixin 是一种类，它包含一组方法，旨在被其他类**"混入"**（通过多重继承）以提供特定的功能，但它本身不打算被单独实例化。
- **优点**:
    1. **功能复用**: 可以在不建立复杂继承关系的情况下，向多个类添加相同的功能。
    2. **高内聚**: 将特定功能（如日志、序列化）封装在一个独立的 Mixin 类中。
    3. **灵活性**: 可以按需组合多个 Mixin。
- **缺点**:
    1. **命名冲突**: 如果多个 Mixin 或主类中有同名方法，可能会导致冲突。
    2. **MRO 复杂化**: 过多的 Mixin 会使方法解析顺序变得复杂，难以理解。
    3. **隐式依赖**: Mixin 可能隐式地依赖于主类中的某些属性或方法。

Q39: (★★★☆☆) `__subclasshook__` 方法有什么用?

- **作用**: `__subclasshook__` 是定义在**抽象基类 (ABC)** 中的一个特殊类方法。
- **目的**: 它允许你自定义 `issubclass(A, B)` 的行为。通过实现 `__subclasshook__`，你可以让一个完全没有继承自你的 ABC 的类 `A`，在 `issubclass(A, YourABC)` 的检查中返回 `True`，只要类 `A` 满足了你定义的某些条件（通常是实现了特定的方法）。
- **本质**: 这是将**鸭子类型**正式纳入 ABC 体系的方式，实现了所谓的**虚拟子类**。

Q40: (★★★☆☆) Python 操作符重载是如何通过魔法方法实现的?

- **实现方式**: Python 允许类通过实现特定的**魔法方法**（Magic Methods，也称 Dunder Methods）来重载内置的操作符。当解释器遇到一个作用于自定义对象的操作符时，它会查找并调用相应的魔法方法。
- **示例**:
  - `a + b` -> `a.__add__(b)`
  - `a * b` -> `a.__mul__(b)`
  - `len(a)` -> `a.__len__()`
  - `a[k]` -> `a.__getitem__(k)`
  - `a == b` -> `a.__eq__(b)`
- 通过实现这些方法，你可以让你的自定义对象表现得像内置类型一样自然。

第四部分：内存管理与执行模型 (★★★★★)

Q41: (★★★★★) 深入剖析 GIL (全局解释器锁)：它为何存在？它锁的是什么？为什么说它对 I/O 密集型任务影响不大？

- **为何存在**: GIL 的存在主要是为了保护 CPython 解释器自身的内存管理。CPython 的垃圾回收机制是基于引用计数的，这个计数器不是线程安全的。如果没有 GIL，多个线程同时修改同一个对象的引用计数时，可能会导致计数错误，从而造成内存泄漏或程序崩溃。GIL 是一种简单的解决方案，确保任何时候只有一个线程在执行 Python 字节码。
- **锁的是什么**: GIL 锁住的是 **Python 解释器**，而不是 Python 对象。它保证了单个进程中，在任意时刻只有一个线程能执行 Python 字节码。
- **对 I/O 密集型任务影响不大**:
  - 当一个 Python 线程执行 I/O 操作时（如文件读写、网络请求），它会**主动释放 GIL**，让其他线程有机会执行。
  - 当 I/O 操作完成后，该线程会重新尝试获取 GIL 继续执行。
  - 因此，在 I/O 密集型任务中，GIL 允许多个线程在等待 I/O 的间隙中"穿插"执行，实现了并发，从而提高了效率。对于 CPU 密集型任务，由于线程始终在执行计算，无法释放 GIL，因此多线程无法利用多核优势。

Q42: (★★★★★) Python 的垃圾回收机制是怎样的？详细说明引用计数、标记-清除和分代回收的原理以及它们如何协同工作。

Python 的垃圾回收（GC）是一个以**引用计数**为主，**标记-清除**和**分代回收**为辅的组合策略。

1. **引用计数 (Reference Counting)**:
   - **原理**: 每个对象都有一个引用计数字段。当有新的引用指向该对象时，计数加 1；当引用被销毁时，计数减 1。当计数变为 0 时，对象所占用的内存会被立即回收。
   - **优点**: 简单、高效，垃圾可以被及时回收。
   - **缺点**: 无法解决**循环引用**的问题（例如 `a.x = b; b.y = a`）。
2. **标记-清除 (Mark-Sweep)**:
   - **原理**: 这是为了解决循环引用问题而生的辅助机制。它将所有对象分为"可达"和"不可达"两类。从根对象（如全局变量、调用栈）开始，遍历所有可达对象并进行标记（Mark）；遍历结束后，所有未被标记的对象都是不可达的垃圾，将被统一回收（Sweep）。
   - **作用**: 专门用于回收由循环引用导致的、引用计数不为 0 的垃圾。
3. **分代回收 (Generational Collection)**:

- **原理**: 这是一种基于统计学假设的优化策略，假设是“绝大多数对象都是朝生夕灭的”。Python 将对象分为三代（0, 1, 2）。新创建的对象都在第 0 代。当第 0 代对象数量达到阈值时，Python 会触发一次 0 代的垃圾回收。经过回收后仍然存活的对象会被提升到第 1 代。第 1 代和第 2 代的回收频率远低于第 0 代。
- **作用**: 通过专注于回收最可能产生垃圾的年轻代对象，大大提高了 GC 的效率。

协同工作: 引用计数处理了绝大部分对象的回收。分代回收机制会定期运行，对各个代执行标记-清除算法，从而高效地回收循环引用的垃圾。

Q43: (★★★★☆) 解释 Python 代码的执行过程（从.py 文件到 PVM 执行.pyc 字节码）。

1. **源代码 (`.py`)**: 程序员编写的 Python 代码。
2. **词法分析与解析**: 解释器读取源代码，进行词法分析（将代码分解成 token），然后进行语法分析，构建一个**抽象语法树 (AST)**。
3. **编译**: 解释器遍历 AST，将其编译成 Python 特有的**字节码 (Bytecode)**。字节码是比机器码更抽象、与平台无关的中间指令集。
4. **缓存 (`.pyc` 文件)**: 编译后的字节码会被缓存到 `.pyc` 文件中。下次运行程序时，如果 `.py` 文件没有被修改，解释器会直接加载 `.pyc` 文件，跳过解析和编译步骤，加快启动速度。
5. **执行**: **Python 虚拟机 (PVM)** 加载并执行字节码。PVM 是一个循环，它逐条读取字节码指令，并在其内部的 C 函数中模拟指令的行为，最终完成程序的执行。

Q44: (★★★★☆) `weakref` 模块的作用是什么？请举一个必须使用弱引用的例子（例如缓存、观察者模式）。

- **作用**: `weakref` 模块允许你创建对对象的**弱引用 (Weak Reference)**。一个弱引用不会增加对象的引用计数。当一个对象只剩下弱引用时，垃圾回收器可以随时回收它。
- **必须使用的例子 (缓存)**:
  - **场景**: 假设我们要实现一个缓存，用于存储大对象的计算结果。缓存的键是对象本身，值是其计算结果。
  - **问题**: 如果使用普通字典作为缓存 (`cache = {obj: result}`), 这会形成一个对 `obj` 的强引用。即使程序其他地方已经不再需要 `obj`，由于缓存还引用着它，`obj` 将永远不会被回收，导致内存泄漏。
  - **解决方案**: 使用 `weakref.WeakKeyDictionary`。

    ```
    import weakref

    # WeakKeyDictionary 的键是弱引用
    cache = weakref.WeakKeyDictionary()

    class BigObject:
        pass

    obj1 = BigObject()
    cache[obj1] = "some computed result"

    # 当 obj1 不再被其他地方使用时，它会被垃圾回收
    # cache 中对应的条目也会被自动移除
    del obj1
    # 此时 cache 会变为空
    ```

Q45: (★★★★☆) 什么是 Python 的命名空间和作用域？LEGB 规则的具体查找顺序是怎样的？

- **命名空间 (Namespace)**: 是一个从**名字到对象的映射**。Python 中有多种命名空间，如内置命名空间、全局命名空间和局部命名空间。它就像一个字典，键是变量名，值是对象。
- **作用域 (Scope)**: 是 Python 程序中一个命名空间可以直接访问的文本区域。
- **LEGB 规则**: 这是 Python 查找一个变量名的顺序：
    1. **L (Local)**: 局部作用域，即当前函数或类的内部。
    2. **E (Enclosing)**: 闭包函数外的函数（即嵌套函数的外部函数）的局部作用域。
    3. **G (Global)**: 全局作用域，即模块的顶层。
    4. **B (Built-in)**: 内置作用域，包含了 Python 的内置函数和异常名（如 `len`, `print`, `Exception`）。

Q46: (★★★☆☆) 变量 `a=1` 和 `a=2`，`id(a)` 会变化吗？`a=[1]` 和 `a.append(2)`，`id(a)` 会变化吗？这背后体现了什么？

- `a=1` 和 `a=2`: `id(a)` **会变化。**
    - **原因**: 整数是**不可变类型。** `a=2` 这个操作不是修改 `1` 这个对象，而是创建了一个新的值为 `2` 的整数对象，然后让变量 `a` 指向这个新对象。
- `a=[1]` 和 `a.append(2)`: `id(a)` **不会变化。**
    - **原因**: 列表是**可变类型。** `a.append(2)` 是一个**就地修改 (in-place)** 操作，它直接在 `a` 所指向的那个列表对象内部追加了一个元素，并没有创建新的列表对象。
- **背后体现**: 体现了 Python 中**可变对象 (Mutable)** 与**不可变对象 (Immutable)** 的核心区别。

Q47: (★★★☆☆) CPython 中小整数对象池和短字符串驻留是如何工作的？

- **小整数对象池**: CPython 为了性能优化，会预先创建并缓存一个范围内的整数对象，默认范围是 `[-5, 256]`。当代码中使用这个范围内的整数时，不会创建新对象，而是直接返回缓存中对象的引用。
- **短字符串驻留 (String Interning)**: 对于一些符合特定规则（如只含字母、数字、下划线）的短字符串，CPython 也会将其缓存到一个字符串池中。当创建新的符合规则的字符串时，如果池中已存在相同内容的字符串，则直接返回其引用。这节省了内存并加快了字符串比较的速度（可以用 `is` 代替 `==`）。

Q48: (★★★☆☆) 什么是 Python 的帧对象 (Frame Object)？如何访问它？

- **帧对象**: 帧对象是 Python 虚拟机（PVM）在执行字节码时使用的数据结构。每当一个函数被调用，就会创建一个新的帧。这个帧对象包含了执行该函数所需的所有信息，如局部变量、全局变量的引用、代码对象、指令指针等。
- **如何访问**:
    - `sys._getframe()`: `sys` 模块的 `_getframe(depth)` 函数可以获取当前调用栈的帧对象。`depth=0` 表示当前帧，`depth=1` 表示调用者的帧。
    - `inspect` **模块**: `inspect.currentframe()` 提供了更安全、更推荐的访问方式。

Q49: (★★★☆☆) `sys.getrefcount()` 的返回值为什么通常比你预期的要大 1？

- 因为当你调用 `sys.getrefcount(my_object)` 时，你将 `my_object` 作为参数传递给了 `getrefcount` 函数。这个传递过程本身就创建了一个对 `my_object` 的**临时引用。**因此，函数在计算引用计数时，这个临时引用也被计算在内，导致结果比实际多 1。

第五部分：迭代、生成与异步编程 (★★★★☆)

Q50: (★★★★★) 迭代器 (Iterator) 和可迭代对象 (Iterable) 的区别是什么？for 循环的工作原理是什么?

- **区别**:
  - **可迭代对象 (Iterable)**: 任何实现了 `__iter__()` 方法的对象。这个方法返回一个迭代器。常见的可迭代对象有列表、元组、字符串、字典等。
  - **迭代器 (Iterator)**: 任何实现了 `__next__()` 方法的对象。`__next__()` 方法返回序列中的下一个元素，当没有元素时，则抛出 `StopIteration` 异常。迭代器自身也实现了 `__iter__()` 方法，该方法返回它自己。
- `for` **循环工作原理**:
  1. `for` 循环首先调用可迭代对象的 `__iter__()` 方法，获取一个迭代器。
  2. 然后，它在一个无限循环中不断调用这个迭代器的 `__next__()` 方法，将返回值赋给循环变量。
  3. 当 `__next__()` 抛出 `StopIteration` 异常时，`for` 循环会捕获这个异常并优雅地退出循环。

Q51: (★★★★★) 生成器 (Generator) 是如何工作的？它相比普通函数返回列表有什么核心优势?

- **工作原理**: 生成器是一种特殊的迭代器，它不需要用类来实现。任何包含 `yield` 关键字的函数都会变成一个**生成器函数**。调用生成器函数不会立即执行，而是返回一个**生成器对象**。每次对这个生成器对象调用 `next()` 时，函数会从上次离开的地方继续执行，直到遇到下一个 `yield` 语句，然后它会交出（yield）一个值并再次暂停。
- **核心优势**:
  1. **内存效率 (惰性求值)**: 生成器一次只在内存中生成一个值，而不是一次性创建整个列表。这对于处理大型数据集或无限序列至关重要，可以极大地节省内存。
  2. **代码简洁**: 用函数语法就能实现一个迭代器，比用类实现 `__iter__` 和 `__next__` 要简洁得多。

Q52: (★★★★☆) `yield` **和** `yield from` **有什么本质区别?**

- `yield`: 用于交出一个值。它暂停生成器的执行，并将 `yield` 后面的表达式作为结果返回给调用者。
- `yield from <iterable>`: 用于**委托**给另一个可迭代对象（通常是另一个生成器）。它会建立一个通道，将子生成器的产出直接传递给最外层的调用者，同时也将调用者发送给主生成器的值传递给子生成器。它极大地简化了嵌套生成器的代码。

Q53: (★★★★☆) 什么是协程 (Coroutine)？它与线程的根本区别是什么?

- **协程**: 协程是一种用户级的轻量级"线程"，通过 `async def` 定义。它可以在执行过程中被暂停，以便让其他协程运行，之后再从暂停处恢复。
- **根本区别**:
  - **调度方式**: 线程是由**操作系统内核**进行**抢占式调度**的，线程的切换时机由操作系统决定。协程是由**程序自身（事件循环）进行协作式调度**的，协程的切换必须由协程本身通过 `await` 关键字主动让出控制权。
  - **资源消耗**: 协程非常轻量，创建和切换的开销极小，可以在单线程内轻松管理成千上万个协程。线程则相对重量级。

- **并发模型**: 线程在多核上可以实现并行，但受 GIL 限制。协程在单线程上实现并发，非常适合 I/O 密集型任务。

Q54: (★★★★☆) `asyncio` 的事件循环 (Event Loop) 是什么？它是如何调度协程的?

- **事件循环**: 事件循环是 `asyncio` 的核心。它是一个在单线程中运行的循环，负责管理和执行所有的异步任务（协程）。
- **调度方式**:
  1. 你通过 `asyncio.create_task()` 或 `asyncio.run()` 将协程注册到事件循环中。
  2. 事件循环将任务放入一个队列，并开始执行第一个任务。
  3. 当一个正在运行的协程遇到 `await` 表达式（例如 `await asyncio.sleep(1)` 或一次网络请求）时，它会暂停自己，并将控制权交还给事件循环。
  4. 事件循环会继续执行队列中的下一个就绪任务。
  5. 同时，事件循环会监听 I/O 事件。当一个被等待的 I/O 操作完成时，事件循环会唤醒之前暂停的那个协程，让它从 `await` 语句后继续执行。

Q55: (★★★☆☆) 列表推导式、集合推导式和生成器表达式在性能和内存使用上有何不同?

- **列表/集合推导式**:
  - **行为**: 立即求值（Eager Evaluation）。它们会一次性在内存中创建并返回一个**完整的列表或集合**。
  - **适用场景**: 当结果集不大，或者你需要多次访问结果时。
- **生成器表达式**:
  - **行为**: 惰性求值（Lazy Evaluation）。它返回一个**生成器对象**（一种迭代器），只有在你迭代它时，才会逐个计算和产出值。
  - **适用场景**: 处理非常大的数据集或无限序列，因为它极大地节省了内存。

Q56: (★★★☆☆) `async/await` 和 `yield from` 在底层实现上有什么联系?

- `async/await` 是在 `yield from` 基础上发展而来的**语法糖**。
- 在 Python 3.5 之前，`asyncio` 的协程是基于生成器实现的，使用 `@asyncio.coroutine` 装饰器，并用 `yield from` 来等待另一个协程或 Future。
- `async/await` 的引入，使得协程的定义和使用在语法上与普通函数更加区分，意图更明确，代码也更易读。但其底层机制仍然依赖于生成器和 `yield` 的暂停/恢复能力。

Q57: (★★★☆☆) `async for` 和 `async with` 是做什么用的?

- `async for`: 用于迭代一个**异步可迭代对象**（实现了 `__aiter__` 和 `__anext__` 方法的对象）。它允许在循环的每次迭代之间发生异步操作（即 `await`）。
  - **场景**: 从数据库游标或网络流中逐行异步读取数据。
- `async with`: 用于处理**异步上下文管理器**（实现了 `__aenter__` 和 `__aexit__` 方法的对象）。它允许在进入和退出上下文的代码块时执行 `await` 操作。
  - **场景**: 异步地获取和释放资源，如数据库连接、网络会话等。

Q58: (★★★☆☆) `asyncio.Future` 和 `asyncio.Task` 的区别是什么?

- `Future`: 是一个**低级别的可等待对象**，代表一个异步操作的**最终结果**。它像一个占位符，你可以给它设置结果或异常，但它本身不绑定任何执行逻辑。通常由库的底层使用。

- `Task`: 是 `Future` 的一个**子类**。它专门用于**包装和运行一个协程**。当你使用 `asyncio.create_task(coro)` 时，它会安排协程 `coro` 在事件循环中执行，并返回一个 `Task` 对象。这个 `Task` 对象既可以被 `await`，也可以用于取消协程。
- **关系**: `Task` 是一个带有执行逻辑的 `Future`。

第六部分：并发与并行 (★★★★☆)

Q59: (★★★★★) 多线程、多进程、协程三者的优缺点和适用场景分别是什么？ (I/O密集型 vs CPU密集型)

- **多进程 (multiprocessing)**:
  - **优点**: 能利用多核实现**真正并行**，绕过 GIL，适合 **CPU 密集型**任务。进程间相互独立，稳定性高。
  - **缺点**: 资源消耗大，创建和切换开销大，进程间通信（IPC）复杂。
- **多线程 (threading)**:
  - **优点**: 资源消耗比进程小，启动快，线程间共享内存，通信方便。适合 **I/O 密集型**任务。
  - **缺点**: 在 CPython 中受 GIL 限制，无法利用多核进行并行计算。需要处理线程同步问题（如锁），容易出错。
- **协程 (asyncio)**:
  - **优点**: 资源消耗**极小**，切换开销极低，单线程内实现高并发。非常适合**高并发 I/O 密集型**任务（如网络服务器）。无需处理锁。
  - **缺点**: 无法利用多核。代码需要异步化设计（`async/await`），有一定学习成本。如果遇到阻塞操作会阻塞整个事件循环。

Q60: (★★★★☆) `multiprocessing` 模块中 `fork`，`spawn`，`forkserver` 三种启动方式有什么区别?

- `fork` **(仅限 Unix)**: 子进程几乎是父进程的完整拷贝，包括内存数据。启动速度快，但可能导致文件描述符或锁状态的混乱（“copy-on-write”）。
- `spawn` **(Windows 和 macOS 默认)**: 子进程从头开始启动一个全新的 Python 解释器，然后导入主模块。启动慢，但环境干净，避免了 `fork` 的问题。
- `forkserver` **(仅限 Unix)**: 一种折中方案。主进程先启动一个“服务器进程”，之后所有新的子进程都由这个服务器进程 `fork` 出来。避免了每次都重新导入模块的开销。

Q61: (★★★★☆) `threading.Lock`，`RLock`，`Semaphore`，`Event` 的区别和使用场景是什么?

- `Lock` **(互斥锁)**: 最基本的锁，保证同一时间只有一个线程能进入临界区。一个线程不能在释放前再次获取它。
- `RLock` **(可重入锁)**: 允许**同一个线程**多次获取锁，内部维护一个计数器。每次获取加 1，每次释放减 1，直到计数为 0 才真正释放锁。主要用于解决递归函数中需要加锁的场景。
- `Semaphore` **(信号量)**: 允许一个固定数量的线程同时访问资源。它内部维护一个计数器，获取时减 1，释放时加 1。当计数器为 0 时，其他线程必须等待。用于限制对有限资源的并发访问（如连接池）。
- `Event` **(事件)**: 最简单的线程间通信机制。一个线程可以发出一个“事件”信号 (`event.set()`)，其他一个或多个线程可以等待这个信号 (`event.wait()`)。

Q62: (★★★★☆) `multiprocessing.Queue` 和 `queue.Queue` 有什么区别?

- `queue.Queue`: 用于**同一进程内**的多个**线程**之间进行通信。它是线程安全的，数据直接在内存中传递。

- `multiprocessing.Queue`: 用于**多个进程**之间进行通信。它是进程安全的。放入队列的对象会被**序列化 (pickle)**，在另一端再**反序列化**，这涉及到管道和锁等更复杂的底层机制。

Q63: (★★★☆☆) `ThreadPoolExecutor` 和 `ProcessPoolExecutor` 有什么区别？如何选择？

- 它们都来自 `concurrent.futures` 模块，提供了统一的高级接口来管理线程池和进程池。
- `ThreadPoolExecutor`: 使用**线程池**来执行任务。
- `ProcessPoolExecutor`: 使用**进程池**来执行任务。
- **如何选择**:
  - 对于 **I/O 密集型**任务，选择 `ThreadPoolExecutor`。
  - 对于 **CPU 密集型**任务，选择 `ProcessPoolExecutor` 以绕开 GIL，利用多核优势。

Q64: (★★★☆☆) 什么是守护线程/守护进程 (Daemon)?

- 守护线程/进程是一种在**后台**运行的线程/进程。
- 它的核心特性是：当一个程序的所有**非守护**线程/进程都退出后，整个程序就会退出，而不会等待守护线程/进程执行完毕。它们会被直接终止。
- **用途**: 适合执行一些不那么重要的后台任务，如心跳检测、日志记录、监控等。

第七部分：模块、包与异常处理 (★★★★☆)

Q65: (★★★★☆) 深入解释 Python 的导入系统 (sys.path, sys.modules, finder, loader)。

Python 的导入系统由几个核心组件协同工作：

1. `sys.modules`: 一个**缓存字典**。当一个模块被第一次导入时，它的模块对象会被存储在这里。后续对同一模块的导入会直接从这个字典中获取，从而避免了重复执行模块代码。这是实现模块单例模式的基础。
2. `sys.path`: 一个字符串**列表**，指定了模块的搜索路径。当导入一个模块时，Python 会按顺序遍历 `sys.path` 中的每个路径来查找模块。
3. **查找器 (Finder)** 和 **加载器 (Loader)**: 这是更底层的导入协议。
   - **Finder**: 它的任务是在 `sys.path` 的路径中**找到**模块。如果找到，它会返回一个包含加载器（Loader）的模块规格（spec）。
   - **Loader**: 它的任务是**加载**模块。它会执行模块的源代码，创建一个模块对象，并将其放入 `sys.modules`。

完整流程: `import my_module`

1. 检查 `sys.modules` 是否已有 `my_module`，如果有则直接返回。
2. 如果没有，遍历 `sys.meta_path` 中的每个 Finder 对象。
3. Finder 尝试在 `sys.path` 中找到 `my_module`。
4. 如果找到，Finder 返回一个包含 Loader 的 spec。
5. 调用 Loader 的 `exec_module` 方法，执行模块代码，创建模块对象。
6. 将新的模块对象存入 `sys.modules`。

Q66: (★★★★☆) 什么是循环导入？如何诊断并解决循环导入问题？

- **什么是循环导入**: 当两个或多个模块相互导入时，就会发生循环导入。例如，`module_a.py` 导入 `module_b.py`，而 `module_b.py` 同时又导入 `module_a.py`。

- **诊断**: 循环导入通常会导致 `ImportError` 或 `AttributeError`。`AttributeError` 是因为在循环中，一个模块试图访问另一个尚未完全加载的模块中的属性。
- **解决方法**:
    1. **重构代码**: 这是最好的方法。重新设计模块的依赖关系，将共享的功能提取到一个独立的第三方模块中，打破循环。
    2. **延迟导入/局部导入**: 将 `import` 语句移到需要使用该模块的函数或方法内部。这会延迟导入操作，直到函数被调用时才发生，此时所有模块通常都已加载完毕。
    3. **使用接口文件或重构依赖**: 将一个模块中的部分类或函数移动到另一个模块，或者使用一个接口模块来解决依赖问题。

Q67: (★★★★☆) `try...except...else...finally` 四个代码块的完整执行逻辑是怎样的？`else` 块有什么用？

- **完整执行逻辑**:
    1. 首先执行 `try` 块中的代码。
    2. **如果没有异常发生**: 执行 `else` 块（如果存在），然后执行 `finally` 块（如果存在）。
    3. **如果发生异常**:
        - Python 查找匹配的 `except` 块并执行它。
        - 执行完 `except` 块后，执行 `finally` 块。
        - 如果没有找到匹配的 `except` 块，异常会向上传播，但在传播之前，`finally` 块仍然会被执行。
- `else` **块的作用**: `else` 块中的代码仅在 `try` 块**没有发生任何异常**时执行。它的主要好处是**提高代码可读性**，可以将“正常流程”的代码与“异常处理”的代码清晰地分离开。那些你希望只在成功时才执行，且其本身可能抛出异常但你不希望被同一个 `try...except` 捕获的代码，就应该放在 `else` 块里。

Q68: (★★★☆☆) `raise` 和 `raise from` 语句有什么区别？

- `raise`: 抛出一个新的异常。如果在 `except` 块中使用 `raise` 而不带任何参数，它会**重新抛出**当前正在处理的那个异常，保留其原始的追溯信息。
- `raise NewException from OriginalException`: 用于**异常链 (Exception Chaining)**。它允许你捕获一个异常，然后抛出一个新的、更具上下文意义的异常，同时**保留原始异常**作为新异常的 `__cause__` 属性。
- **核心区别**: `raise from` 清晰地表明新异常是由原始异常直接引起的，这在调试时非常有用，因为它提供了完整的异常上下文链。

Q69: (★★★☆☆) `__init__.py` 文件有什么作用？`__all__` 变量是做什么用的？

- `__init__.py` **的作用**:
    1. **标记包**: 它的主要作用是告诉 Python，包含它的目录应该被视为一个**包 (Package)**。在 Python 3.3 之前，这是必需的。
    2. **初始化包**: 当包被导入时，`__init__.py` 文件会被自动执行。你可以在这里编写包级别的初始化代码，例如设置包级别的变量。
    3. **简化导入**: 可以在 `__init__.py` 中导入子模块的特定属性，从而让用户可以直接从包导入，如 `from my_package import my_class` 而不是 `from my_package.my_module import my_class`。
- `__all__` **的作用**:

- `__all__` 是一个定义在模块或 `__init__.py` 中的字符串列表。
- 它用于控制当执行 `from <module> import *` 时，**哪些名字会被导入**。如果定义了 `__all__`，只有列表中的名字会被导入。如果没有定义，`import *` 会导入所有不以下划线开头的公共名称。

Q70: (★★★☆☆) 绝对导入和相对导入的区别是什么?

- **绝对导入**: 从项目的根目录开始指定完整的模块路径。
  - **语法**: `from my_app.utils import helper`
  - **优点**: 路径清晰明确，不会有歧义。推荐在大多数情况下使用。
- **相对导入**: 使用点号 `.` 来表示相对于当前模块位置的路径。
  - **语法**: `from . import utils` (从同级目录导入 `utils`) 或 `from ..models import User` (从上级目录的 `models` 模块导入 `User`)。
  - **优点**: 在重构包名时更方便。
  - **限制**: 相对导入只能在包内部使用，不能在顶层脚本中使用。

Q71: (★★★☆☆) 常规包 (有 `__init__.py`) 和命名空间包 (无 `__init__.py`) 有什么区别和应用场景?

- **常规包**: 包含 `__init__.py` 文件的目录。它是一个物理上集中的包。
- **命名空间包 (Python 3.3+)**: 不包含 `__init__.py` 文件的目录。
- **核心区别与应用场景**: 命名空间包允许一个包的子模块**分散在文件系统的多个不同位置**。多个目录可以共同构成同一个命名空间。这对于大型库或插件系统非常有用，因为它允许不同的第三方包向同一个命名空间（如 `requests.plugins`）贡献模块，而无需修改原始库的代码。

Q72: (★★★☆☆) 如何设计一个健壮的自定义异常体系?

1. **创建基础异常类**: 为你的库或应用创建一个专属的基础异常类，让它继承自 Python 内置的 `Exception`。

```
class MyAppBaseException(Exception):
    pass
```

2. **创建具体异常类**: 让所有其他具体的自定义异常都继承自这个基础异常类。

```
class DatabaseError(MyAppBaseException):
    pass

class ValidationError(MyAppBaseException):
    pass
```

3. **提供有用信息**: 在异常中包含有用的上下文信息，帮助调试。
4. **使用**: 用户可以通过捕获你的基础异常类 `except MyAppBaseException:` 来捕获所有来自你库的预期异常。

第八部分：标准库与常用模式 (★★★☆☆)

Q73: (★★★★☆) `itertools` 和 `functools` 模块中，你最常用的是哪几个函数？请举例说明其应用场景。

- `itertools` **(高效迭代)**:

    - `chain`: 将多个可迭代对象连接成一个单一的迭代器。`itertools.chain('ABC', 'DEF')` -> A B C D E F。
    - `cycle`: 无限重复一个可迭代对象中的元素。`itertools.cycle('ABC')` -> A B C A B C ...。
    - `islice`: 对迭代器进行切片操作，返回一个新的迭代器。`itertools.islice(range(10), 2, 8, 2)` -> 2 4 6。
- `functools` **(高阶函数)**:
    - `partial`: (见 Q26) 冻结函数的部分参数，创建一个新函数。
    - `lru_cache`: 一个装饰器，为函数提供一个**最近最少使用 (Least Recently Used)** 的内存缓存。对于计算开销大且同样参数会重复调用的函数（如斐波那契数列）能极大地提升性能。
    - `wraps`: (见 Q20) 在创建装饰器时，用于保留原始函数的元信息。

Q74: (★★★☆☆) `re` **模块中** `match()`, `search()`, `findall()` **的区别是什么?**

- `re.match(pattern, string)`: 从字符串的**起始位置**开始匹配。如果起始位置不匹配，则返回 `None`。
- `re.search(pattern, string)`: 扫描整个字符串，查找**第一个**匹配的位置。
- `re.findall(pattern, string)`: 扫描整个字符串，以**列表**形式返回所有不重叠的匹配项。

Q75: (★★★☆☆) `with` **语句和上下文管理器协议是如何工作的？如何用生成器实现一个上下文管理器?**

- **上下文管理器协议**: 任何一个类，只要实现了 `__enter__` 和 `__exit__` 两个方法，就遵循了上下文管理器协议。
- `with` **语句工作原理**:
    1. `with` 语句首先调用上下文管理器对象的 `__enter__` 方法。`__enter__` 的返回值（如果有）会被赋给 `as` 后面的变量。
    2. 然后，执行 `with` 代码块中的语句。
    3. 无论代码块是正常结束还是发生异常，`__exit__(exc_type, exc_val, exc_tb)` 方法都一定会被调用。它的三个参数用于接收异常信息，如果没有异常则都为 `None`。
- **用生成器实现**: 使用 `contextlib` 模块的 `@contextlib.contextmanager` 装饰器。

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwargs):
    # __enter__ 部分：yield 之前
    resource = acquire_resource(*args, **kwargs)
    try:
        yield resource  # yield 的值会赋给 as 后面的变量
    finally:
        # __exit__ 部分：yield 之后
        release_resource(resource)
```

Q76: (★★★☆☆) `datetime` **模块中 naive 和 aware datetime 对象有什么区别？如何正确处理时区问题?**

- **区别**:
    - **Naive (天真) 对象**: 不包含时区信息。它只是一个时间数字，无法与另一个不同时区的 naive 对象进行有意义的比较。

    - **Aware (感知) 对象**: 包含时区信息（通过 `tzinfo` 属性）。它代表一个明确的、无歧义的时刻。
- **如何正确处理**:
    1. **使用 `zoneinfo` (Python 3.9+) 或 `pytz` 库** 来创建 aware 对象。
    2. **最佳实践**: 在程序内部，尽量将所有时间都转换为 **UTC** (协调世界时) 进行存储和计算。
    3. 仅在与用户交互（输入/输出）时，才将 UTC 时间转换为用户所在的本地时区。

Q77: (★★★☆☆) `pathlib` 模块相比 `os.path` 有什么优势?

- **面向对象**: `pathlib` 使用 `Path` 对象来表示文件系统路径，而 `os.path` 使用纯字符串。
- **优势**:
    1. **可读性与易用性**: 代码更直观。`path / 'subdir' / 'file.txt'` 比 `os.path.join(path, 'subdir', 'file.txt')` 更简洁。
    2. **方法丰富**: `Path` 对象自带了大量有用的方法，如 `.exists()`, `.is_dir()`, `.read_text()`, `.glob()` 等。
    3. **跨平台一致性**: `pathlib` 能更好地处理不同操作系统的路径差异。
    4. **类型安全**: 传递的是对象而不是字符串，减少了错误。

Q78: (★★★☆☆) `logging` 模块的基本组件（Logger, Handler, Formatter, Filter）是什么？它们是如何协同工作的?

- **基本组件**:
    - **Logger**: 程序员与之交互的入口。提供 `debug()`, `info()`, `warning()`, `error()` 等方法。
    - **Handler**: 决定日志的去向，如输出到控制台 (`StreamHandler`)、文件 (`FileHandler`) 等。
    - **Formatter**: 定义最终日志记录的格式（如时间、级别、消息内容）。
    - **Filter**: 提供更精细的日志过滤控制。
- **协同工作**:
    1. 代码调用 `Logger` 实例的方法（如 `logger.info("message")`）。
    2. `Logger` 根据其级别和 `Filter` 判断是否要处理这条日志。
    3. 如果处理，`Logger` 将日志消息传递给所有关联的 `Handler`。
    4. 每个 `Handler` 也根据自己的级别和 `Filter` 判断是否处理。
    5. 如果处理，`Handler` 使用其关联的 `Formatter` 将日志记录格式化成字符串。
    6. 最后，`Handler` 将格式化后的字符串发送到指定的目标（控制台、文件等）。

Q79: (★★★☆☆) `pickle` 模块是做什么用的？它有什么安全风险?

- **作用**: `pickle` 模块用于**序列化 (Serialization)** 和**反序列化 (Deserialization)** Python 对象。
    - 序列化（`pickle.dump()`）: 将 Python 对象转换为一个字节流，以便存储到文件或通过网络传输。
    - 反序列化（`pickle.load()`）: 从字节流中重建原始的 Python 对象。
- **安全风险**: **绝对不要反序列化来自不可信或未经验证来源的数据。** `pickle` 格式的字节流可以被恶意构造，在反序列化时执行任意代码，这可能导致远程代码执行漏洞。

第九部分：Python 内部与C API (高阶) (★★☆☆☆)

Q80: (★★☆☆☆) 什么是 `PyObject` 和 `PyVarObject`？它们在CPython内部是什么角色?

- 它们是 CPython 解释器内部用来表示所有 Python 对象的 C 语言结构体。
- `PyObject`: 是所有 Python 对象的**基础结构体**。它包含两个最基本的字段：
  - `ob_refcnt`: 对象的引用计数。
  - `ob_type`: 一个指向类型对象的指针，决定了该对象是什么类型（如整数、列表）。
- `PyVarObject`: 是 `PyObject` 的一个变体，用于表示**可变长度**的对象，如列表、字符串、字典等。它在 `PyObject` 的基础上增加了一个字段：
  - `ob_size`: 表示对象包含的元素数量。

Q81: (★★☆☆☆) 在C扩展中，如何正确地管理Python对象的引用计数？（`Py_INCREF`, `Py_DECREF`）

- 在 C 扩展中，必须手动管理 Python 对象的引用计数以防止内存泄漏。
- `Py_INCREF(obj)`: 当你创建了一个对 `obj` 的新引用（即"拥有"了这个引用）时，必须调用它来增加引用计数。
- `Py_DECREF(obj)`: 当你不再需要一个对象的引用（即"释放"所有权）时，必须调用它来减少引用计数。当计数变为 0 时，对象会被销毁。
- **核心原则 (所有权)**:
  - 当你从其他函数接收到一个对象时，通常是**借用引用 (Borrowed Reference)**，你不需要管理它。
  - 当你创建一个新对象（如 `PyLong_FromLong`）或需要长期持有某个对象的引用时，你就获得了**新的所有权 (New Reference)**，你必须在未来某个时刻负责 `DECREF` 它。

Q82: (★★☆☆☆) CPython 的 GIL 在C代码层面是如何实现加锁和释放的?

- GIL 本质上是一个**互斥锁 (mutex)**，并带有一个线程切换的计时器。
- 在 C 代码层面，主要通过一对宏来操作：
  - `Py_BEGIN_ALLOW_THREADS`: 这个宏会**释放 GIL**。它通常用在即将进入一个阻塞的 C 语言 I/O 操作（如 `read`, `write`）之前。
  - `Py_END_ALLOW_THREADS`: 这个宏会**重新获取 GIL**。它用在阻塞的 C 操作完成之后，返回 Python 解释器之前。
- 这种机制使得 C 扩展能够在执行耗时的、非 Python 相关的操作时，允许其他 Python 线程运行，从而实现了对 I/O 密集型任务的并发支持。

第十部分：测试、工具与工程化 (★★★★☆)

Q83: (★★★★☆) `unittest` 和 `pytest` 有什么主要区别？为什么 `pytest` 在社区中更受欢迎?

- **主要区别**:
  - **用例编写**: `unittest` 采用类继承的方式（继承 `unittest.TestCase`），遵循 xUnit 风格。`pytest` 使用更简洁的函数式风格，普通的函数即可作为测试用例。
  - **断言方式**: `unittest` 需要使用 `self.assertEqual()`, `self.assertTrue()` 等一系列断言方法。`pytest` 直接使用 Python 原生的 `assert` 语句，失败时会提供详细的上下文信息。
  - **Fixture 机制**: `pytest` 拥有强大而灵活的 Fixture 机制，用于测试前置/后置处理和依赖注入，而 `unittest` 主要依赖 `setUp/tearDown` 方法，功能相对简单。

    - **插件生态**: `pytest` 拥有一个极其丰富的插件生态系统，可以轻松扩展其功能（如 `pytest-cov` 用于覆盖率，`pytest-xdist` 用于并行测试）。
- `pytest` **更受欢迎的原因**:
    1. **代码更简洁**: 测试用例编写简单直观，减少了大量模板代码。
    2. **断言更强大**: `assert` 语句易于编写和阅读，且失败信息非常详细。
    3. **Fixture 功能强大**: Fixture 机制是其杀手级特性，能优雅地管理测试依赖和状态。
    4. **插件丰富**: 强大的插件生态极大地提升了测试效率和能力。

Q84: (★★★★☆) 什么是 `pytest` 的 fixture？请解释它的作用、生命周期（scope）以及 `yield` 的用法。

- **什么是 Fixture**: Fixture 是 `pytest` 中用于测试准备和清理工作的函数。它们为测试用例提供了一个固定的、可靠的基线环境，如数据库连接、临时文件、测试数据等。
- **作用**:
    1. **准备前置条件 (Setup)**: 在测试运行前执行代码，准备所需的环境和数据。
    2. **执行清理工作 (Teardown)**: 在测试结束后执行代码，清理资源。
    3. **依赖注入**: 测试函数可以直接通过参数名来请求使用一个 fixture，`pytest` 会自动将其返回值注入。
- **生命周期 (Scope)**: 通过 `@pytest.fixture(scope=...)` 设置。
    - `function` (默认): 每个测试**函数**执行一次。
    - `class`: 每个测试**类**执行一次。
    - `module`: 每个**模块**执行一次。
    - `package`: 每个**包**执行一次。
    - `session`: 整个测试**会话**只执行一次。
- `yield` **的用法**: 在 fixture 中使用 `yield` 可以将 fixture 分为两部分。`yield` 之前的代码是 **Setup** 部分，`yield` 语句本身返回 fixture 的值。`yield` 之后的代码是 **Teardown** 部分，会在使用该 fixture 的测试结束后执行。

Q85: (★★★★☆) 什么是 Mock？在 `unittest.mock` 中，`Mock` 和 `MagicMock` 的区别是什么？`patch` 的常用方式有哪些？

- **什么是 Mock**: Mock（模拟对象）是一种在测试中替代真实对象的测试替身（Test Double）。它的目的是隔离被测试单元，使其不依赖于外部系统（如数据库、API）或复杂的内部组件，从而让测试变得简单、快速和可控。
- `Mock` **vs** `MagicMock`:
    - `MagicMock` 是 `Mock` 的一个子类。
    - 它预先实现了大部分 Python 的魔法方法（如 `__str__`, `__len__`, `__iter__` 等）。
    - **核心区别**: 当你需要模拟一个行为像列表或字典的对象，或者需要对魔法方法进行断言时，应该使用 `MagicMock`。在其他大多数情况下，`Mock` 就足够了。
- `patch` **的常用方式**: `patch` 用于在测试期间动态地替换模块或类中的对象。
    1. **作为装饰器**: `@patch('module.ClassName')`，最常用的方式。
    2. **作为上下文管理器**: `with patch('module.ClassName') as MockClass:`，用于在代码块内进行替换。

3. **手动启动和停止**: `patcher = patch(...)`, `mock_obj = patcher.start()`, `patcher.stop()`，提供了最精细的控制。

Q86: (★★★☆☆) Python 的环境管理工具有哪些？`venv` 和 `virtualenv` 的区别是什么？`conda` 又适用于什么场景？

- **主要工具**: `venv`, `virtualenv`, `conda`, `pipenv`, `poetry`。
- `venv` **vs** `virtualenv`:
  - `venv`: 是 Python 3.3+ **内置**的标准库。功能相对基础。
  - `virtualenv`: 是一个第三方库，需要单独安装。功能更强大，支持更早的 Python 版本，且创建环境的速度通常更快。
- `conda` **的适用场景**:
  - `conda` 是一个跨平台的包管理器和环境管理器，主要来自于 Anaconda 发行版。
  - **核心优势**: 它不仅能管理 Python 包，还能管理**非 Python 的软件包**（如 C/C++ 库、CUDA 等）。
  - **适用场景**: 特别适用于**数据科学和机器学习**领域，因为这些领域常常依赖于复杂的非 Python 库。

Q87: (★★★☆☆) `requirements.txt` 和 `pyproject.toml` / `setup.py` 在项目依赖管理中分别扮演什么角色？

- `requirements.txt`:
  - **角色**: 用于定义一个**应用 (Application)** 的具体依赖。
  - **目的**: 确保在不同环境中部署应用时，能够复现一个**确切的、固定的**依赖环境。通常包含具体的版本号（如 `requests==2.28.1`），可以使用 `pip freeze` 生成。
- `pyproject.toml` / `setup.py`:
  - **角色**: 用于定义一个**库 (Library)** 的元数据和依赖。
  - **目的**: 当其他人要安装你的库时，告诉打包工具（如 `pip`）这个库需要哪些依赖。它通常定义一个**宽松的版本范围**（如 `requests>=2.25.0`），以保证最大的兼容性。`pyproject.toml` 是 PEP 518 引入的现代化标准，正在逐渐取代 `setup.py`。

Q88: (★★★☆☆) 什么是 Linting 和 Formatting？请列举至少两种常用的工具并说明其作用。

- **Linting (代码检查)**:
  - **作用**: 对代码进行**静态分析**，检查代码中的语法错误、风格问题、潜在的 bug 和不符合规范的代码。它的目的是**发现问题**。
  - **工具**:
    - `Flake8`: 结合了 `PyFlakes` (检查错误) 和 `pycodestyle` (检查风格) 的工具，配置灵活。
    - `Pylint`: 功能非常强大，检查项非常全面，但配置也相对复杂。
- **Formatting (代码格式化)**:
  - **作用**: 自动**重写**代码，使其符合一个固定的、统一的风格规范。它的目的是**解决风格问题**。
  - **工具**:
    - `Black`: 被称为"不妥协的代码格式化工具"，风格强制统一，几乎没有配置项。
    - `isort`: 专门用于对 `import` 语句进行排序和格式化。

Q89: (★★★☆☆) `mypy` 是做什么用的？在 Python 中引入静态类型检查有什么好处和潜在的成本？

- **`mypy` 的作用**: `mypy` 是一个**静态类型检查器**。它会读取你在代码中添加的类型提示（Type Hints），并在不运行代码的情况下分析代码，找出潜在的类型不匹配错误。
- **好处**:
  1. **提前发现 Bug**: 可以在编码阶段就发现由于类型错误导致的 bug。
  2. **提高代码可读性和可维护性**: 类型提示就像一种文档，让其他人更容易理解函数期望的输入和输出。
  3. **改善 IDE 支持**: IDE 可以利用类型提示提供更精准的代码补全和重构建议。
- **潜在成本**:
  1. **增加工作量**: 需要花时间为代码添加和维护类型提示。
  2. **学习成本**: 需要学习类型系统的相关知识。
  3. **处理第三方库**: 对于没有提供类型提示的第三方库，可能需要为其编写存根文件 (`.pyi`)。

第十一部分：Web 后端与网络 (以 Flask & FastAPI 为核心) (★★★★☆)

Q90: (★★★★★) 深入解释 WSGI (Flask) 和 ASGI (FastAPI) 的区别。它们如何将请求传递给应用，以及这种差异如何影响性能和功能（例如 WebSocket）？

- **核心区别**:
  - **WSGI (Web Server Gateway Interface)**: 是**同步**网关接口，专为同步框架（如 Flask, Django）设计。它遵循"请求-响应"模型，一次只能处理一个请求。
  - **ASGI (Asynchronous Server Gateway Interface)**: 是**异步**网关接口，是 WSGI 的超集，专为异步框架（如 FastAPI, Starlette）设计。它支持长时间运行的连接，如 WebSocket 和 HTTP/2。
- **请求传递**:
  - **WSGI**: 应用服务器（如 Gunicorn）接收到一个完整的 HTTP 请求后，调用一个可调用对象（即 Flask 应用），并将请求信息作为 `environ` 字典和 `start_response` 函数传入。
  - **ASGI**: 应用服务器（如 Uvicorn）与应用通过一个包含 `scope`, `receive`, `send` 三个参数的异步可调用对象进行通信。`receive` 和 `send` 是异步函数，允许在单个请求的生命周期内进行多次双向通信。
- **影响**:
  - **性能**: ASGI 应用可以通过事件循环在等待 I/O 时处理其他请求，从而在 I/O 密集型场景下实现更高的并发性能。
  - **功能**: WSGI 无法原生支持 WebSocket 等需要持久连接的协议。ASGI 的双向通信能力使其可以轻松实现 WebSocket、服务器发送事件（SSE）等现代 Web 功能。

Q91: (★★★★☆) 对比 Flask 和 FastAPI 的请求生命周期。Flask 的应用上下文和请求上下文是如何工作的？FastAPI 的依赖注入系统又是如何实现类似功能的？

- **Flask 上下文**:
  - Flask 使用两个线程隔离的**上下文**来管理请求数据：
    - **请求上下文 (Request Context)**: 封装了当次 HTTP 请求的信息，包含 `request` 和 `session` 对象。

        - **应用上下文 (Application Context)**: 封装了应用级别的信息，包含 `current_app` (当前应用实例) 和 `g` (请求级别的临时存储)。
    - **工作方式**: 当一个请求进入时，Flask 会**推送**相应的上下文到栈顶，使其成为全局可访问的代理对象。请求结束时，上下文被**弹出**。
- **FastAPI 依赖注入 (Dependency Injection)**:
    - FastAPI 不使用全局上下文，而是通过一个强大的**依赖注入系统**来管理请求数据和资源。
    - **工作方式**: 你在路径操作函数（视图函数）的参数中用 `Depends()` 声明依赖。FastAPI 会在处理请求时，自动调用这些依赖函数，并将它们的返回值**注入**到你的路径操作函数中。每个依赖项在同一次请求中只会被计算一次并缓存结果。
- **对比**: Flask 的上下文是**隐式**的，全局可用。FastAPI 的依赖注入是**显式**的，通过函数签名声明，代码更清晰，耦合度更低。

Q92: (★★★★☆) Flask 的 `g` 对象和 FastAPI 的 `Depends` 有什么异同？请举例说明它们在处理请求级别的状态时的应用场景。

- **相同点**: 两者都可用于在单次请求的生命周期内共享数据或状态，例如存储当前认证的用户信息或数据库会话。
- **不同点**:
    - **Flask `g` 对象**: 是一个简单的**全局命名空间对象**，其生命周期与请求上下文绑定。你需要**手动**在请求开始时（如使用 `before_request` 钩子）向 `g` 中存入数据，然后在视图函数中直接访问 `g.user`。
    - **FastAPI `Depends`**: 是一个**声明式的依赖注入系统**。你将获取数据的逻辑封装在一个依赖函数中，然后在需要它的地方用 `Depends` 声明。FastAPI 会自动管理依赖的执行和结果注入。
- **应用场景举例 (获取当前用户)**:
    - **Flask**:

```
@app.before_request
def load_logged_in_user():
    g.user = get_user_from_db(...)

@app.route('/profile')
def profile():
    return f"Hello, {g.user.name}"
```

    - **FastAPI**:

```
def get_current_user(token: str = Depends(oauth2_scheme)):
    return get_user_from_db(token)

@app.get("/profile")
def profile(current_user: User = Depends(get_current_user)):
    return f"Hello, {current_user.name}"
```

Q93: (★★★☆☆) FastAPI 如何利用 Pydantic 进行数据验证和序列化？这与 Flask 中常用的 Marshmallow 或 WTForms 有何不同?

- **FastAPI & Pydantic**:
    - FastAPI 与 Pydantic **深度集成**。你只需要使用 Python 的类型提示来定义一个继承自 `pydantic.BaseModel` 的模型。

    - FastAPI 会自动使用这个模型来：
        1. **数据验证**: 对传入的请求体（JSON）进行类型检查和规则验证。验证失败会自动返回 422 错误。
        2. **数据序列化**: 对返回的数据进行格式化。
        3. **API 文档**: 自动生成符合 OpenAPI 规范的交互式 API 文档 (Swagger UI)。
- **Flask & Marshmallow/WTForms**:
    - 在 Flask 中，数据验证和序列化通常由**第三方库**（如 Marshmallow, WTForms）完成。
    - 你需要**手动**定义一个 Schema，然后在视图函数中**显式地**调用它来加载（验证）输入数据和导出（序列化）输出数据。
- **核心不同**: FastAPI 的方式是**声明式**和**自动化**的，代码更少，且与文档生成无缝集成。Flask 的方式是**命令式**和**手动**的，需要编写更多的模板代码。

Q94: (★★★☆☆) 在 Flask 和 FastAPI 中，你会如何实现用户认证？请对比基于 Session 的认证和基于 JWT 的无状态认证，并说明它们各自的优缺点和适用场景。

- **Session-based 认证 (有状态)**:
    - **原理**: 用户登录后，服务器创建一个 Session 并将其 ID 存储在客户端的 Cookie 中。后续请求，服务器通过 Cookie 中的 Session ID 查找对应的 Session 数据来验证用户。
    - **优点**: 数据存储在服务端，相对安全；可以方便地在服务端撤销 Session。
    - **缺点**: **有状态**，对服务器有存储开销；不利于水平扩展和分布式部署。
    - **适用场景**: 传统的单体 Web 应用。
- **JWT-based 认证 (无状态)**:
    - **原理**: 用户登录后，服务器生成一个包含用户信息的、经过签名的 JSON Web Token (JWT) 并返回给客户端。客户端将 JWT 存储起来（如 Local Storage），并在后续请求的 `Authorization` 头中携带它。服务器只需验证 JWT 的签名即可，无需查询数据库或缓存。
    - **优点**: **无状态**，服务器无需存储 Session，易于水平扩展；天然适用于微服务和移动应用。
    - **缺点**: Token 一旦签发，在过期前难以撤销；Payload 暴露（虽然无法篡改），不应存放敏感信息。
    - **适用场景**: API、微服务架构、单页应用 (SPA)。

第十二部分：数据处理与分析 (以 NumPy & Pandas 为核心) (★★★★☆)

NumPy 核心

Q95: (★★★★☆) NumPy 的 `ndarray` 相比 Python 的 `list` 有什么核心优势？请从内存布局、数据类型和计算性能三个方面解释。

1. **内存布局**: `ndarray` 对象在内存中是一块**连续的、固定大小**的区域。而 Python 的 `list` 存储的是指向各个对象的指针，这些对象在内存中是分散的。连续内存布局使得 `ndarray` 的缓存命中率更高。
2. **数据类型**: `ndarray` 要求所有元素都是**同一种数据类型**（如 `int32`, `float64`），这消除了类型检查的开销。`list` 可以存储不同类型的对象。
3. **计算性能**: 基于以上两点，NumPy 可以将计算任务委托给底层的高度优化的 C 或 Fortran 代码，并利用 **SIMD (单指令多数据流)** 指令集并行处理数据，实现所谓的**向量化运算 (Vectorization)**。这比 Python 层面使用 `for` 循环进行计算要快几个数量级。

Q96: (★★★★☆) 什么是 NumPy 的广播 (Broadcasting) 机制？请举一个具体的例子，说明两个不同形状的数组是如何通过广播进行计算的。

- **什么是广播**: 广播是 NumPy 在处理不同形状的数组进行算术运算时的一套规则。它可以在不实际复制数据的情况下，将较小的数组“拉伸”或“复制”，使其形状与较大的数组兼容，从而实现元素级的运算。
- **规则**: 从两个数组的尾部维度开始逐个比较：
  1. 如果维度大小相等，或其中一个为 1，则兼容。
  2. 如果不满足以上条件，则不兼容，抛出错误。
- **例子**: 计算一个 `(3, 3)` 数组和一个 `(3,)` 数组的和。

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  # Shape (3, 3)
b = np.array([10, 20, 30])                       # Shape (3,)

# 广播过程:
# a.shape -> (3, 3)
# b.shape ->    (3)
# 1. NumPy 将 b 的形状看作 (1, 3)。
# 2. 将 b 在第 0 轴上“拉伸”3次，使其形状变为 (3, 3)。
#    [[10, 20, 30],
#     [10, 20, 30],
#     [10, 20, 30]]
# 3. 然后进行元素级相加。
c = a + b
# c 的结果:
# [[11, 22, 33],
#  [14, 25, 36],
#  [17, 28, 39]]
```

Q97: (★★★☆☆) 解释 NumPy 中的轴 (Axis) 的概念。在 `np.sum()`, `np.mean()` 等聚合函数中，`axis=0` 和 `axis=1` 分别代表什么？

- **轴 (Axis)**: 轴就是数组的**维度**。对于一个二维数组（矩阵），你可以把它想象成一个表格：
  - `axis=0` 指的是**纵向**的轴，即沿着**行**的方向进行操作。
  - `axis=1` 指的是**横向**的轴，即沿着**列**的方向进行操作。
- **聚合函数中的含义**: `axis` 参数指定了**沿着哪个轴进行“压缩”或“折叠”**。
  - `np.sum(axis=0)`: 对**每一列**的元素求和。结果是一个一维数组，其长度等于列数。
  - `np.sum(axis=1)`: 对**每一行**的元素求和。结果是一个一维数组，其长度等于行数。

Pandas 核心

Q98: (★★★★☆) 什么是 Pandas 的 `Series` 和 `DataFrame`？它们与 NumPy 的 `ndarray` 有什么关系？

- `Series`: 一个带**标签**的一维数组。它由数据和与之关联的索引（Index）组成。可以看作是一个加强版的 `ndarray` 或一个有序的字典。
- `DataFrame`: 一个带**标签**的二维表格型数据结构，拥有行索引（Index）和列索引（Columns）。你可以把它看作是由多个 `Series` 共享同一个行索引组成的字典。
- **与 `ndarray` 的关系**:

- Pandas 是建立在 NumPy 之上的。
- `Series` 和 `DataFrame` 的底层数据都是由一个或多个 NumPy 的 `ndarray` 存储的。
- 你可以通过 `.values` 属性从 `Series` 或 `DataFrame` 中直接获取其底层的 NumPy 数组。

Q99: (★★★★☆) 详细解释 Pandas `groupby` 操作的"拆分-应用-合并" (Split-Apply-Combine) 模式。请举例说明如何使用 `groupby` 进行数据聚合。

- **"拆分-应用-合并"模式**:
  1. **拆分 (Split)**: 根据指定的键（如某一列的值），将 `DataFrame` 拆分成若干个子组。
  2. **应用 (Apply)**: 对每个子组独立地应用一个函数。这个函数可以是聚合函数（如 `sum`, `mean`）、转换函数（如 `transform`）或过滤函数（如 `filter`）。
  3. **合并 (Combine)**: 将应用函数后的结果合并成一个新的 `DataFrame` 或 `Series`。
- **举例 (数据聚合)**: 计算不同部门员工的平均薪水。

```
import pandas as pd
data = {'department': ['HR', 'IT', 'HR', 'IT', 'IT'],
        'salary': [5000, 8000, 5500, 9000, 8500]}
df = pd.DataFrame(data)

# 1. 拆分: 根据 'department' 列将 df 拆分成 HR 组和 IT 组。
# 2. 应用: 对每个组的 'salary' 列应用 mean() 函数。
# 3. 合并: 将结果合并成一个新的 Series。
avg_salary = df.groupby('department')['salary'].mean()
print(avg_salary)
# department
# HR    5250.0
# IT    8500.0
# Name: salary, dtype: float64
```

Q100: (★★★☆☆) Pandas 中 `loc` 和 `iloc` 的核心区别是什么？在哪些场景下应该优先使用其中一个?

- **核心区别**:
  - `df.loc[]`: 基于**标签 (Label)** 的索引。它使用行索引名和列名进行选择。切片时，**包含**结束标签。
  - `df.iloc[]`: 基于**整数位置 (Integer Location)** 的索引。它使用从 0 开始的整数位置进行选择，类似 Python 列表的切片。切片时，**不包含**结束位置。
- **优先使用场景**:
  - **优先使用 `loc`**: 当你的索引具有实际意义时（如日期、ID），使用 `loc` 可以让代码更具可读性，且不受行序变化的影响。
  - **优先使用 `iloc`**: 当你需要根据行的数字位置进行选择，或者处理没有有意义标签的 `DataFrame` 时，使用 `iloc`。在循环或需要与 NumPy 风格的整数索引交互时也很有用。

Q101: (★★★☆☆) 当处理一个远超内存大小的数据集时，你会采用哪些基于 Pandas 的策略和工具?

1. **分块读取**: 在使用 `pd.read_csv()` 时，设置 `chunksize` 参数。这会返回一个迭代器，你可以逐块（chunk）地处理数据，每次只将一小部分数据加载到内存中。
2. **指定数据类型**: 在读取数据时，使用 `dtype` 参数为列指定更节省内存的数据类型（如用 `float32` 代替 `float64`，用 `category` 类型处理低基数的字符串列）。

3. **只读取需要的列**: 使用 `usecols` 参数，只加载你实际需要的列。
4. **使用核外计算库**: 对于更复杂的计算，可以使用像 **Dask** 或 **Vaex** 这样的库。它们提供了类似 Pandas 的 API，但能在不将整个数据集加载到内存的情况下，并行地处理数据。