冯楷翔期末课程设计——易错代码或调试经验

根据python上机经验，5处易错代码或调试经验

根据python上机经验，结合代码或库函数分析python语法特点，并列出5处易错代码或调试经验（调试经验最好结合图说明）。

易错1：可变默认参数

```
def append_to_list(value, my_list=[]):
    my_list.append(value)
    return my_list

print(append_to_list(1))
print(append_to_list(2))
```

- **问题描述**：在函数定义中使用可变对象（如列表、字典）作为默认参数，导致意外的行为。

```
E:\anaconda\python.exe D:\wpsfinished\大三上\python\期末课设\finalproject\yicuo.py
[1]
[1, 2]
```

- **调试经验**：

期望每次调用函数时 `my_list` 都是一个新的空列表，但实际上默认参数在函数定义时只初始化一次，导致多次调用共享同一个列表。

- **解决方案**：使用 `None` 作为默认参数，并在函数内部进行初始化。

```
def append_to_list(value, my_list=None):
    if my_list is None:
        my_list = []
    my_list.append(value)
    return my_list

```

易错2：不熟悉Python的内置数据结构

- **问题描述**：

C程序员可能习惯使用数组和结构体，但Python提供了列表、元组、字典、集合等丰富的数据结构。不了解或错误使用这些数据结构会导致代码效率低下或功能实现不当。

```
# 使用列表存储键值对
data = [["name", "Alice"], ["age", 30]]
# 查找值需要遍历列表
def get_value(key):
    for k, v in data:
        if k == key:
            return v
    return None

print(get_value("age"))  # 输出: 30

```

- **常见错误**：

不利用字典的键值对查找优势，导致查找效率低下

- **调试经验**：

使用内置数据结构直接完成

- **解决方案**：使用适当的数据结构，如字典进行键值对存储和快速查找。

```
data = {"name": "Alice", "age": 30}

def get_value(key):
    return data.get(key)

print(get_value("age"))  # 输出: 30

```

易错3：错误处理方式的差异

- **问题描述**：

C语言通常使用返回值或错误码来处理错误，而Python使用异常机制。C程序员可能会尝试在Python中模仿C的错误处理方式，导致未捕获的异常或错误处理不当。

```
def divide(a, b):
    if b == 0:
```

```
        return -1  # 模仿C的错误码
    return a / b

result = divide(10, 0)
if result == -1:
    print("Error: Division by zero")
```

- **常见错误**：

  能用，但忽略Python的异常机制，导致代码不够简洁且容易出错

- **调试经验**：

  使用try+except，可读性更高

- **解决方案**：

```
def divide(a, b):return a / b

try:
    result = divide(10, 0)
except ZeroDivisionError as e:print(f"Error: {e}")
```

易错4：异常捕获不当

- **问题描述**：

  过度捕获异常，使用裸 `except` 或捕获不具体的异常，可能掩盖真实错误。

```
try:
    result = 10 / 0
except:
    print("An error occurred")
```

- **常见错误**：

  捕获所有异常，无法得知具体的错误类型和位置，调试困难。

- **调试经验**：

  使用try时过于潦草，没有定义出现异常的情况

- **解决方案**：

  捕获具体的异常类型，并在处理时记录详细信息。

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
```

易错5：混淆 `is` 与 `==`

- **问题描述**：

  误用 `is` 操作符比较值，导致逻辑错误。`is` 比较对象的身份，而 `==` 比较值的相等性。

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a == b)  # 输出: True
print(a is b)  # 输出: False
```

- **常见错误**：

  期望 `a is b` 为 `True`，因为列表内容相同。

- **调试经验**：

  明确使用 `==` 与 `is` 区别。

- **解决方案**：

  使用 `==` 进行值比较，`is` 用于比较对象身份。