第十一章

异常处理

异常处理（Exception Handling）是确保程序稳健性和可靠性的重要机制。通过捕获和处理运行时可能出现的错误，程序能够在面对意外情况时继续运行，避免崩溃。在商业数据分析领域，异常处理尤为关键。数据源可能存在缺失值、格式错误或其他异常情况，若不加以处理，可能导致分析结果失真或程序中断。因此，掌握 Python 的异常处理机制，有助于开发健壮的数据分析应用，确保分析过程的连续性和结果的准确性。

11.1 异常的概念

重要性：★★★★；　**难易度：**★

在 Python 编程中，**异常（Exception）**是指在程序执行过程中出现的错误或意外情况，导致程序无法按照预期继续运行。异常处理机制使程序能够捕获并处理这些错误，确保程序的稳健性和可靠性。

当 Python 解释器在执行代码时遇到错误，如试图除以零、访问不存在的变量或打开不存在的文件，便会引发异常。如果未对这些异常进行处理，程序将终止并显示错误信息。通过异常处理，程序可以捕获这些错误，执行相应的处理逻辑，从而避免程序崩溃。

以下示例演示了如何处理除零错误：

```
try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
    print(result)
except ZeroDivisionError:
    print("错误：除数不能为零。")
```

在上述代码中，`try` 块包含可能引发异常的代码。当执行到 `result = numerator / denominator` 时，由于 `denominator` 为零，会引发 `ZeroDivisionError` 异常。此时，程序跳转到对应的 `except` 块，输出提示信息“错误：除数不能为零。”。通过这种方式，程序避免了因未处理的异常而崩溃。

在 Python 编程中，异常处理机制通过 `try`、`except`、`else`、`finally` 和 `raise` 语句来管理程序运行时的错误。`try` 块包含可能引发异常的代码，`except` 块用于捕获并处理这些异常，`else` 块在未发生异常时执行，`finally` 块无论是否发生异常都会执行。此外，`raise` 语句允许程序员在特定条件下主动引发异常，以便在检测到错误或异常情况时中断正常的程序流程，并将控制权交给相应的异常处理器。

11.2 异常传播机制

异常传播机制指的是当异常在当前作用域未被捕获时，沿调用栈向上传递，直到被捕获或导致程序终止的过程。这种机制确保异常信息能够传递给适当的处理程序，以便采取相应的措施。

异常传播的工作原理：当代码块中发生异常时，Python 会检查该代码块是否有对应的 `except` 子句来处理该异常。如果存在匹配的处理程序，异常被捕获，程序继续执行。如果没有匹配的处理程序，异常会沿调用栈向上传递，直到找到合适的处理程序或到达程序的最顶层。如果在最顶层仍未捕获该异常，程序将终止，并输出未处理的异常信息。

以下示例演示了异常的传播过程：

```
def function_a():
    function_b()

def function_b():
    function_c()

def function_c():
    raise ValueError("发生了一个值错误")

try:
    function_a()
except ValueError as e:
    print(f"捕获到异常：{e}")
```

运行结果：

```
捕获到异常：发生了一个值错误
```

解析：

在上述代码中，`function_c` 中显式引发了 `ValueError` 异常。由于 `function_c` 内部没有处理该异常，异常被传播到调用它的 `function_b`。同样地，`function_b` 也未处理该异常，异常继续传播到 `function_a`。最终，异常传播到 `function_a` 的调用者，即 `try` 块。在此处，存在匹配的 `except` 子句来捕获 `ValueError` 异常，因此异常被成功捕获，程序输出相应的提示信息。

通过理解异常的传播机制，可以在适当的层次捕获和处理异常，确保程序的稳健性和可靠性。

11.3 Python 的内置异常类

异常处理是确保程序稳健性和可靠性的重要机制。Python 提供了丰富的内置异常类，用于捕获和处理各种错误情况。这些异常类均继承自 `BaseException`，并形成层次化的结构，方便开发者根据具体需求

进行捕获和处理。以下是一些常见的 Python 内置异常类及其使用示例：

1. ZeroDivisionError

当尝试除以零时引发。

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("错误：除数不能为零。")
```

2. ValueError

当函数接收到参数类型正确但值不合适时引发。

```
try:
    number = int("abc")
except ValueError:
    print("错误：无法将字符串转换为整数。")
```

3. TypeError

当操作或函数应用于不适当类型的对象时引发。

```
try:
    result = '2' + 2
except TypeError:
    print("错误：不能将字符串与整数相加。")
```

4. IndexError

当尝试访问序列中不存在的索引时引发。

```
try:
    numbers = [1, 2, 3]
    print(numbers[5])
except IndexError:
    print("错误：索引超出列表范围。")
```

5. KeyError

当在字典中使用一个不存在的键时引发。

```
try:
    data = {'name': 'Alice'}
    print(data['age'])
except KeyError:
    print("错误：键不存在于字典中。")
```

6. FileNotFoundError

当尝试打开一个不存在的文件时引发。

```
try:
    with open('nonexistent_file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("错误：文件未找到。")
```

7. AttributeError

当尝试访问对象中不存在的属性时引发。

```
try:
    obj = None
    obj.method()
except AttributeError:
    print("错误：对象没有该属性或方法。")
```

8. ImportError

当导入模块失败时引发。

```
try:
    import nonexistent_module
except ImportError:
    print("错误：模块导入失败。")
```

了解并正确处理这些内置异常，有助于编写健壮的 Python 程序，确保在各种错误情况下程序能够优雅地处理并继续运行。表11.1列出了常见的 Python 内置异常类及其含义。

11.4 自定义异常类

Python 内置的异常类已能处理大多数常见错误情况。然而，在特定的应用场景中，可能需要定义自定义异常类，以更准确地描述和处理特定的错误条件。

自定义异常类是通过继承内置的 `Exception` 类或其子类来实现的。通过这种方式，可以创建符合特定需求的异常类型，从而提供更精确的错误信息，提高代码的可读性和维护性。

以下示例展示了如何定义和使用自定义异常类：

```
class CustomError(Exception):
    """自定义异常类，继承自Exception"""
    def __init__(self, message, error_code):
        super().__init__(message)
        self.error_code = error_code

def process_data(data):
    if not isinstance(data, dict):
        raise CustomError("数据类型错误，期望为字典类型。", 1001)
    # 处理数据的逻辑
    print("数据处理成功。")

try:
    sample_data = ["这是一个列表，而非字典"]
    process_data(sample_data)
except CustomError as e:
    print(f"错误代码：{e.error_code} - 错误信息：{e}")
```

运行结果：

```
错误代码：1001 - 错误信息：数据类型错误，期望为字典类型。
```

代码解析：

在上述代码中，定义了一个名为 `CustomError` 的自定义异常类，继承自 `Exception`。该类的构造函数接受两个参数：`message` 和 `error_code`，并调用父类的构造函数初始化异常信息。在 `process_data` 函数中，首先检查传入的 `data` 是否为字典类型；如果不是，则引发 `CustomError` 异常，并提供相应的错误信息和错误代码。在 `try` 块中调用 `process_data` 函数，并在 `except` 块中捕获 `CustomError` 异常，输出错误代码和错误信息。

通过定义自定义异常类，可以更精确地描述特定的错误情况，增强代码的可读性和可维护性。

11.5 异常处理语句

11.5.1 raise 语句

`raise` 语句用于显式引发异常，以便在程序运行过程中处理特定的错误或异常情况。通过使用 `raise`，可以在代码中指定何时以及为何引发异常，从而提高程序的健壮性和可维护性。

raise 语句的基本用法：

1. 引发指定的异常：

```
raise Exception("这是一个自定义的异常消息")
```

上述代码将引发一个通用的 `Exception` 异常，并附带自定义的错误消息。

2. 引发特定类型的异常：

```
raise ValueError("无效的输入")
```

此示例引发一个 `ValueError` 异常，通常用于表示传入函数的参数具有有效类型但不在期望的值范围内。

以下示例演示了如何在函数中使用 `raise` 语句引发异常，并在调用该函数时捕获和处理异常：

```
def calculate_square_root(value):
    if value < 0:
        raise ValueError("输入值不能为负数")
    return value ** 0.5

try:
    result = calculate_square_root(-9)
except ValueError as e:
    print(f"错误：{e}")
```

运行结果：

```
错误：输入值不能为负数
```

解析：

在上述代码中，定义了一个名为 `calculate_square_root` 的函数，用于计算给定数值的平方根。函数首先检查输入值是否为负数；如果是，则使用 `raise` 语句引发 `ValueError` 异常，并提供相应的错

误消息。在调用该函数时，使用 `try` 块捕获可能引发的异常，并在 `except` 块中处理 `ValueError` 异常，输出错误信息。

通过这种方式，可以在程序中主动引发并处理异常，确保程序在遇到错误条件时能够优雅地处理，而不是直接崩溃。

11.5.2 try/except 语句捕获异常

`try/except` 语句用于捕获和处理程序运行时可能发生的异常，确保程序的稳健性和可靠性。其基本结构如下：

```
try:
    # 可能引发异常的代码
except 异常类型:
    # 处理异常的代码
```

在 `try` 块中编写可能引发异常的代码；如果发生指定类型的异常，程序将跳转到对应的 `except` 块执行处理代码。

以下示例演示了如何使用 `try/except` 语句处理除零错误：

```
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("错误：除数不能为零。")
        return None
    else:
        return result

# 测试代码
num1 = 10
num2 = 0
output = divide_numbers(num1, num2)
if output is not None:
    print(f"结果：{output}")
```

运行结果：

```
错误：除数不能为零。
```

解析：

在上述代码中，定义了一个名为 `divide_numbers` 的函数，用于执行两个数字的除法运算。在 `try` 块中，尝试执行除法操作；如果 `b` 为零，将引发 `ZeroDivisionError` 异常。此时，程序跳转到对应的 `except` 块，输出错误信息“错误：除数不能为零。”，并返回 `None`。如果未发生异常，`else` 块将执行，返回计算结果。

通过使用 `try/except` 语句，可以有效地捕获和处理程序运行时的异常，避免程序因未处理的错误而崩溃，提高代码的健壮性。

11.5.3 else 和 finally 子句

`else` 和 `finally` 子句用于增强代码的健壮性和可读性。`else` 子句在 `try` 块中代码成功执行且未引发任何异常时运行，而 `finally` 子句则无论是否发生异常，都会执行。

`else` 子句：

`else` 子句紧跟在所有 `except` 子句之后，仅在 `try` 块未引发任何异常时执行。这对于将正常执行路径与异常处理逻辑分开非常有用。

示例：

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("除数不能为零。")
else:
    print(f"结果是：{result}")
```

输出：

```
结果是：5.0
```

解析：

在上述代码中，`try` 块成功执行除法操作，未引发任何异常，因此执行 `else` 子句，输出计算结果。

`finally` 子句：

`finally` 子句无论是否发生异常，都会执行，通常用于释放资源或执行清理操作。

示例：

```
try:
    file = open('example.txt', 'r')
    content = file.read()
except FileNotFoundError:
    print("文件未找到。")
else:
    print(content)
finally:
    file.close()
    print("文件已关闭。")
```

输出（假设文件存在且内容为"Hello, World!"）：

```
Hello, World!
文件已关闭。
```

解析：

在上述代码中，`try` 块尝试打开并读取文件内容。如果文件存在且读取成功，执行 `else` 子句，打印文件内容。无论是否发生异常，`finally` 子句都会执行，确保文件被关闭。

通过结合使用 `else` 和 `finally` 子句，可以编写出更清晰、可靠的异常处理代码，确保资源的正确管理和释放。

11.5.4 try/except 语句捕获多个异常

try/except 语句可以捕获和处理程序运行时可能发生的异常。当需要捕获多个异常时，可以在单个 except 块中指定多个异常类型，或为每种异常类型定义独立的 except 块。

1. 在单个 except 块中捕获多个异常

可以在一个 except 块中通过元组指定多个异常类型。当 try 块中的代码引发这些异常之一时，程序将执行该 except 块。

```
try:
    # 可能引发异常的代码
except (TypeError, ValueError) as e:
    print(f"捕获到异常：{e}")
```

以下示例演示了如何在单个 except 块中捕获 TypeError 和 ValueError 异常：

```
def process_data(data):
    try:
        # 尝试将数据转换为整数
        number = int(data)
        # 执行除法操作
        result = 10 / number
    except (ValueError, ZeroDivisionError) as e:
        print(f"处理数据时发生错误：{e}")
    else:
        print(f"结果是：{result}")

# 测试代码
process_data("abc")  # 将引发ValueError
process_data("0")    # 将引发ZeroDivisionError
process_data("5")    # 正常处理
```

运行结果：

```
处理数据时发生错误：invalid literal for int() with base 10: 'abc'
处理数据时发生错误：division by zero
结果是：2.0
```

解析：

在上述代码中，process_data 函数尝试将输入数据转换为整数，并执行除法操作。如果输入无法转换为整数，将引发 ValueError；如果输入为零，将引发 ZeroDivisionError。通过在单个 except 块中捕获这两种异常，可以简化异常处理逻辑，提高代码的可读性。

2. 在多个 except 块中捕获不同的异常

如果需要对不同类型的异常执行不同的处理操作，可以为每种异常类型定义独立的 except 块。

```
try:
    # 可能引发异常的代码
```

```
except TypeError as e:
    print(f"捕获到TypeError：{e}")
except ValueError as e:
    print(f"捕获到ValueError：{e}")
```

以下示例演示了如何为 `TypeError` 和 `ValueError` 定义独立的 `except` 块：

```
def add_numbers(a, b):
    try:
        result = a + b
    except TypeError as e:
        print(f"类型错误：{e}")
    else:
        print(f"结果是：{result}")

# 测试代码
add_numbers(5, "10")  # 将引发TypeError
add_numbers(5, 10)    # 正常处理
```

运行结果：

```
类型错误：unsupported operand type(s) for +: 'int' and 'str'
结果是：15
```

解析：

在上述代码中，`add_numbers` 函数尝试将两个参数相加。如果参数类型不匹配，将引发 `TypeError`。通过为 `TypeError` 定义独立的 `except` 块，可以针对该异常类型执行特定的处理操作。

通过合理使用 `try/except` 语句，可以有效地捕获和处理多个异常，增强程序的健壮性和容错能力。

11.6 警告机制

在 Python 编程中，警告机制用于在不终止程序执行的情况下，提醒开发者注意可能存在的问题或不推荐的用法。这对于维护代码的健壮性和向后兼容性尤为重要。

警告的概念：

警告是一种非致命的通知，旨在提示开发者注意代码中的潜在问题，如使用了即将废弃的功能或存在可能导致错误的操作。与异常不同，警告不会中断程序的执行。

Python 中的 `warnings` 模块：

Python 提供了内置的 `warnings` 模块，用于发出、控制和过滤警告信息。该模块允许开发者根据需要定制警告的处理方式，如忽略特定警告、将警告转换为异常或自定义警告的显示格式。

发出警告：

要在代码中发出警告，可使用 `warnings.warn()` 函数。该函数接受警告消息和警告类别作为参数。

示例：

```
import warnings

def deprecated_function():
```

```
    warnings.warn("该函数已废弃，请使用新函数替代。", DeprecationWarning)
    # 函数的其他实现
```

在上述代码中，调用`deprecated_function()`时，将发出一条`DeprecationWarning`，提示该函数已废弃。

控制警告行为：

可以使用`warnings`模块中的过滤器函数来控制警告的处理方式。例如，使用`warnings.filterwarnings()`函数可以设置忽略特定类型的警告。

```
import warnings

# 忽略所有的DeprecationWarning
warnings.filterwarnings("ignore", category=DeprecationWarning)

def deprecated_function():
    warnings.warn("该函数已废弃，请使用新函数替代。", DeprecationWarning)
    # 函数的其他实现

deprecated_function()
```

在上例中，通过设置过滤器，程序将忽略所有的`DeprecationWarning`，因此即使调用了`deprecated_functio`也不会显示警告信息。

自定义警告类别：

开发者可以通过继承`Warning`类，定义自定义的警告类别，以满足特定需求。

示例：

```
import warnings

class CustomWarning(Warning):
    pass

def function_with_custom_warning():
    warnings.warn("这是一个自定义警告。", CustomWarning)
    # 函数的其他实现

function_with_custom_warning()
```

在上述代码中，定义了一个名为`CustomWarning`的自定义警告类别，并在函数中发出了该类型的警告。

通过合理使用 Python 的警告机制，开发者可以在不影响程序正常运行的情况下，提醒自己或他人注意代码中的潜在问题，从而提高代码的可维护性和可靠性。

11.7 异常处理在商业数据分析中的应用

11.7.1 会计数据分析中的异常处理

在会计数据分析过程中，可能会遇到数据缺失、格式错误或数值异常等问题。为确保数据处理的准确性和可靠性，需要有效地捕获和处理这些异常。Python 提供了丰富的内置异常类，可用于识别和处理不同类型的错误。

以下示例展示了使用 Python 内置异常类处理会计数据异常的代码：

```
def calculate_average_balance(transactions):
    """
    计算账户的平均余额。
    :param transactions: 包含交易金额的列表
    :return: 平均余额
    """
    try:
        if not transactions:
            raise ValueError("交易列表为空。")
        total_balance = sum(transactions)
        average_balance = total_balance / len(transactions)
        return average_balance
    except TypeError as e:
        print(f"类型错误：{e}。请确保所有交易金额均为数值类型。")
    except ZeroDivisionError as e:
        print(f"零除错误：{e}。交易列表可能为空。")
    except Exception as e:
        print(f"发生未知错误：{e}")
    return None

# 示例交易数据
transaction_data = [1000, 2000, '三千', 4000]

# 计算平均余额
average = calculate_average_balance(transaction_data)
if average is not None:
    print(f"平均余额为：{average}")
else:
    print("计算平均余额失败。")
```

运行结果：

```
类型错误：unsupported operand type(s) for +: 'int' and 'str'。请确保所有交易金额均为数值类型。
计算平均余额失败。
```

解析：

在上述代码中，定义了一个函数`calculate_average_balance`，用于计算账户的平均余额。该函数首先检查传入的交易列表是否为空；如果为空，则引发`ValueError`异常。然后，计算交易金额的总和，并除以交易数量以获得平均余额。在计算过程中，可能会遇到以下异常：

- `TypeError`：当交易列表中包含非数值类型的数据时，例如字符串’三千’，会引发类型错误。

- `ZeroDivisionError`：当交易列表为空时，计算平均值会导致零除错误。
- `Exception`：捕获其他未预见的异常。

通过使用 Python 的内置异常类，可以有效地捕获和处理不同类型的错误，确保会计数据分析过程的稳健性。

11.7.2 商品数据分析中的自定义异常处理

在商品数据分析过程中，可能会遇到数据缺失、格式错误或数值异常等问题。为有效捕获和处理这些特定错误，可定义自定义异常类，以提高代码的可读性和维护性。

以下示例展示了自定义异常类在商品数据验证中的应用：

```
class DataValidationError(Exception):
    """数据验证错误的基类"""
    pass

class MissingFieldError(DataValidationError):
    """缺少必要字段时引发的异常"""
    def __init__(self, field_name):
        super().__init__(f"缺少必要字段：{field_name}")
        self.field_name = field_name

    class InvalidValueError(DataValidationError):
    """字段值无效时引发的异常"""
    def __init__(self, field_name, value):
        super().__init__(f"字段 '{field_name}' 的值无效：{value}")
        self.field_name = field_name
        self.value = value

def validate_product_data(product):
    """验证商品数据的函数"""
    required_fields = ['id', 'name', 'price', 'quantity']
    for field in required_fields:
        if field not in product:
            raise MissingFieldError(field)
    if not isinstance(product['price'], (int, float)) or product['price'] < 0:
        raise InvalidValueError('price', product['price'])
    if not isinstance(product['quantity'], int) or product['quantity'] < 0:
        raise InvalidValueError('quantity', product['quantity'])
    # 其他验证逻辑
    print("商品数据验证通过。")

# 示例商品数据
product_data = {
    'id': 101,
    'name': '示例商品',
    'price': -50,  # 无效的价格
    'quantity': 20
}
```

```
try:
    validate_product_data(product_data)
except DataValidationError as e:
    print(f"数据验证错误：{e}")
```

运行结果：

```
数据验证错误：字段 'price' 的值无效：-50
```

解析：

在上述代码中，定义了一个基础异常类 `DataValidationError`，用于处理数据验证相关的错误。继承自该类的 `MissingFieldError` 和 `InvalidValueError` 分别用于处理缺少必要字段和字段值无效的情况。在 `validate_product_data` 函数中，首先检查商品数据是否包含所有必需字段；如果缺少某个字段，则引发 `MissingFieldError` 异常。随后，验证价格和数量字段的值是否有效；如果无效，则引发 `InvalidValueError` 异常。在 `try` 块中调用 `validate_product_data` 函数，并在 `except` 块中捕获 `DataValidationError` 异常，输出相应的错误信息。

通过这种方式，可以在商品数据分析过程中有效捕获和处理特定的错误情况，确保数据的准确性和可靠性。

11.7.3 国际贸易数据分析中的异常处理

在国际贸易数据分析中，处理数据时可能会遇到多种异常情况，如文件未找到、数据格式错误或计算过程中出现的数学错误。通过使用 Python 的 `try/except` 语句，可以有效地捕获和处理这些异常，确保程序的稳健性。

以下代码示例展示了如何在国际贸易数据分析中使用 `try/except` 语句处理多个可能的异常：

```
import csv

def analyze_trade_data(file_path):
    try:
        with open(file_path, mode='r') as file:
            reader = csv.reader(file)
            headers = next(reader)
            data = [row for row in reader]
            # 假设进行一些数据分析操作
            total_trade = sum(float(row[2]) for row in data)
            average_trade = total_trade / len(data)
            print(f"总贸易额：{total_trade}")
            print(f"平均贸易额：{average_trade}")
    except FileNotFoundError:
        print(f"错误：文件 '{file_path}' 未找到。")
    except ValueError:
        print("错误：数据格式不正确，无法转换为浮点数。")
    except ZeroDivisionError:
        print("错误：数据为空，无法计算平均值。")
    except Exception as e:
        print(f"发生未知错误：{e}")
```

```
# 调用函数进行分析
analyze_trade_data('trade_data.csv')
```

CSV 文件内容示例：

```
Country,Product,TradeAmount
USA,Electronics,12000.50
China,Textiles,8000
Germany,Machinery,15000
India,Pharmaceuticals,7000
```

代码解析：

- **导入必要模块**：使用 `csv` 模块读取 CSV 文件。
- **定义分析函数**：`analyze_trade_data` 函数接受文件路径作为参数，尝试打开并读取文件内容。
- **读取数据**：使用 `csv.reader` 读取文件内容，提取表头和数据行。
- **数据分析**：计算总贸易额和平均贸易额，并输出结果。
- **异常处理**：使用多个 `except` 块分别捕获可能的异常，包括：
  - `FileNotFoundError`：文件未找到。
  - `ValueError`：数据格式错误，无法转换为浮点数。
  - `ZeroDivisionError`：数据为空，导致除以零错误。
  - `Exception`：捕获其他未知错误。

通过这种方式，可以在国际贸易数据分析过程中有效地处理多种可能的异常情况，确保程序的稳健性和可靠性。

11.7.4 商品评论分析中的异常处理

在商品评论分析过程中，可能会遇到文件读取错误、数据处理异常等情况。通过使用 Python 的 `try`、`except`、`else` 和 `finally` 语句，可以有效地捕获和处理这些异常，确保程序的稳健性。

代码示例：

```
import json

def analyze_reviews(file_path):
    try:
        with open(file_path, 'r') as file:
            reviews = json.load(file)
    except FileNotFoundError:
        print(f"错误：文件 '{file_path}' 未找到。")
    except json.JSONDecodeError:
        print(f"错误：文件 '{file_path}' 不是有效的JSON格式。")
    else:
        # 假设进行一些评论分析操作
        total_reviews = len(reviews)
        print(f"总评论数：{total_reviews}")
    finally:
```

```
        print("评论分析过程结束。")

# 调用函数进行分析
analyze_reviews('product_reviews.json')
```

代码解析：

- **导入必要模块**：使用 `json` 模块处理 JSON 格式的数据。
- **定义分析函数**：`analyze_reviews` 函数接受文件路径作为参数，尝试打开并读取文件内容。
- **异常处理**：
  - 使用 `try` 块尝试打开并读取文件。
  - 使用 `except` 块捕获可能的异常：
    * `FileNotFoundError`：文件未找到。
    * `json.JSONDecodeError`：文件不是有效的 JSON 格式。
  - 使用 `else` 块在没有异常时执行评论分析操作，如计算总评论数。
  - 使用 `finally` 块在无论是否发生异常的情况下，执行结束语句。

通过上述代码，可以在商品评论分析过程中有效地处理多种可能的异常情况，确保程序的稳健性和可靠性。

表 11.1: 常见的 Python 内置异常类及其含义

异常类	含义
`Exception`	所有内置非系统退出异常的基类。
`ArithmeticError`	数学运算错误的基类，包括溢出、除零等错误。
`ZeroDivisionError`	除法或取模运算的第二个操作数为零时引发。
`OverflowError`	数值运算结果超出表示范围时引发。
`AssertionError`	`assert` 语句失败时引发。
`AttributeError`	尝试访问对象不存在的属性时引发。
`EOFError`	输入函数到达文件末尾条件（EOF）时引发。
`ImportError`	导入模块失败时引发。
`IndexError`	序列中索引超出范围时引发。
`KeyError`	在字典中使用不存在的键时引发。
`KeyboardInterrupt`	用户中断执行时（通常是按下 `Ctrl+C` ）引发。
`MemoryError`	操作耗尽内存时引发。
`NameError`	尝试访问未声明的变量时引发。
`NotImplementedError`	尚未实现的方法被调用时引发。
`OSError`	操作系统相关错误的基类，包括文件未找到、权限错误等。
`RuntimeError`	在检测到不属于其他类别的错误时引发。
`StopIteration`	迭代器没有更多值时引发。
`SyntaxError`	语法错误时引发。
`IndentationError`	缩进不正确时引发。
`TabError`	缩进中混合使用 Tab 和空格时引发。
`TypeError`	操作或函数应用于不适当类型的对象时引发。
`UnboundLocalError`	访问未在局部作用域中赋值的局部变量时引发。
`ValueError`	操作或函数接收到具有正确类型但不适当值的参数时引发。
`UnicodeError`	Unicode 相关编码或解码错误的基类。
`UnicodeEncodeError`	Unicode 编码时引发的错误。
`UnicodeDecodeError`	Unicode 解码时引发的错误。
`UnicodeTranslateError`	Unicode 翻译时引发的错误。