异常处理

注意事项
避免滥用异常：不要用异常替代条件判断（如检查列表是否越界而非依赖 try）。

明确捕获类型：尽量指定具体异常类型（而非笼统的 except:），防止隐藏潜在问题。

- 异常的定义及作用
- 常见的异常类型
- 常见的异常处理方法
- 手动引发异常
- 自定义异常

异常的定义和作用

定义：异常（Exception） 是Python中表示程序运行期间错误或异常情况的对象。当程序遇到无法正常执行的状况（如逻辑错误、无效输入、资源不可用等），会抛出一个异常对象。

例子：除以零

```
yichang.py
1    print(10/0) # ZeroDivisionError
```

一个小问题：为什么需要异常处理？

程序运行时可能遇到意外，默认情况下，程序会直接崩溃并显示错误信息，让程序优雅地处理错误，继续运行或友好提示。

作用： 防止崩溃，提升用户体验， 错误分类处理，精准定位。

资源释放，保证稳定性。

常见的异常类型

常见异常类型	触发场景	例子	类比
ZeroDivisionError	除以零	10 / 0	数学老师最恨的除以零
FileNotFoundError	文件不存在	open("xxx.txt")	就像打开冰箱发现可乐没了
ValueError	类型转换失败	int("abc")	让数字‘abc’现原形的照妖镜
IndexError	列表索引越界	lst[10]（lst长度5）	超市储物柜只有50个，你非要开第51个
KeyError	访问字典不存在的键	dict["不存在的键"]	拿不存在的钥匙开保险箱

常见的异常处理方法

1. 基础try-except结构

这是最基本的异常处理方式。将你认为可能会抛出异常的代码放在 try 块中，并使用 except 块来捕获并处理这些异常。

适用场景：处理已知的特定错误类型

特点：精确捕获指定异常，避免误捕其他错误

```
try:
    #可能会引发异常的代码
    result = 10/0
except ZeroDivisionError:
    #处理特定的代码
    print("除数不能为零")
```

2. 多异常捕获

一个 try 块后面可以跟随多个 except 语句，以处理不同类型的异常。

适用场景：需要区分处理不同类型的错误

特点：类似医院分科室看病的逻辑

```
try:
    #可能会引发不同类型异常的代码
    data = int("not a number")
except ValueError:
    #处理特定的代码
    print("输入了非数字字符")
except TypeError:
    print("类型不匹配")
```

常见的异常处理方式

3.使用通用异常（Exception）：

如果你想要捕获所有类型的异常,可以使用 except Exception as e，它会捕获任何继承自 Exception 类的异常。

适用场景：需要记录未知错误

注意：慎用！可能会掩盖未预料到的错误

```
try:
    #可能发生任何异常的代码
    result = 10/0
except Exception as e:
    #处理所有异常的代码
    print(f"发生了异常:", {e})
```

4.Else 子句：

当 try 块中的代码没有抛出异常时，else 子句中的代码将会被执行。

适用场景：分离正常逻辑和错误处理逻辑。

特点：使代码更清晰，避免在try块中混杂过多代码。

```
try:
    #可能引发异常的代码
    number = int(input("请输入一个数字:"))
except ValueError:
    #处理特定的代码
    print("这不是一个有效的数字")
else:
    print(f"你输入的数字是:{number}")
```

常见的异常处理方式

5. finally资源清理

finally 块中的代码无论是否发生异常都会执行，通常用于清理操作，如关闭文件或释放资源。

适用场景：必须执行的操作（如关闭文件、释放网络连接）

特点：无论是否发生异常都会执行

```
file = None
try:
    file=open('test.txt','r')
    #可能引发异常的操作
except IOError:
    #处理特定的代码
    print("文件不存在或无法打开")
finally:
    if file:
        file.close()
#无论是否发生异常，都会执行finally中的代码
    print("文件已关闭")
```

手动引发异常

1.概念

手动引发异常指的是通过 raise 关键字，在特定条件下主动触发一个异常，这有助于在遇到不符合预期的情况时及时中断程序流程，并采取相应的措施。

2.语法

raise ExceptionType("异常信息")

其中，ExceptionType 是你想要抛出的异常类型（如 ValueError, TypeError 等），"异常信息"是可选的描述信息。

案例：

```python
def check_adult():
    try:
        #用户输入年龄
        age = int(input("请输入你的年龄:"))

        #判断是否成年
        if age < 18:
            raise ValueError("你还未成年")

    except ValueError as e:
        #捕获ValueError异常并处理（包括手动引发的）
        print(f"发生异常:{e}")
    else:
        #没有异常发生时执行的代码
            print("你已经成年了")
    finally:
        #无论是否发生异常都会执行的代码
        print("程序结束")

#调用函数
check_adult()
```

代码解析：
手动引发异常：
如果用户输入的年龄小于 18，使用 raise ValueError("你还未成年，无法继续操作") 主动抛出ValueError异常。
try 块：
包含可能引发异常的代码（如类型转换和条件检查）。
如果条件不满足，使用 raise 主动抛出异常。
except 块：
捕获 ValueError 异常（无论是由 int() 类型转换失败引发的，还是手动抛出的）。
输出对应的错误信息。
else 块：
如果 try 块中的代码没有抛出异常，则执行 else 块中的代码。
在这里，我们确认用户已成年，并输出提示信息。
finally 块：
无论是否发生异常，finally 块中的代码都会被执行。
在这个例子中，它用于打印一条消息，表示年龄检查结束。

自定义异常

1.概念

自定义异常是开发者根据需求自己定义的一种异常类型，它继承自 Python 的内置 Exception 类或其子类，通过自定义异常，可以更好地描述特定场景下的错误。

2.如何定义一个自定义异常？

创建一个类，并继承 Exception 或其子类，可以在类中添加构造函数（__init__），用于设置错误信息。

基本语法：

```
class MyCustomError(Exception):
    def __init__(self, message="这是一个默认的错误信息"):
        self.message = message
        super().__init__(self.message)
```

class MyCustomError(Exception)：定义了一个新的异常类，继承自 Exception。
def __init__(self, message)：为异常定义了初始化方法，接收错误信息。
super().__init__(self.message)：调用父类的构造函数，传递错误信息。

```python
# 自定义异常类
|解释|添加注释|×
class InvalidScoreError(Exception):
    |解释|添加注释|×
    def __init__(self, score, message="分数无效"):
        self.score = score
        self.message = f"{message}: 输入的分数为 {score}，必须在 0 到 100 之间"
        super().__init__(self.message)

# 使用自定义异常的函数
|解释|添加注释|×
def check_score(score):
    if score < 0 or score > 100:
        raise InvalidScoreError(score)  # 手动引发自定义异常
    print(f"分数有效: {score}")

# 主函数
try:
    user_input = float(input("请输入一个分数 (0-100): "))
    check_score(user_input)  # 调用检查分数的函数
except InvalidScoreError as e:
    print(f"捕获到异常: {e}")
finally:
    print("程序运行结束。\n")
```

1.InvalidScoreError是一个自定义异常类，它继承自内置的Exception类。通过定义___init___方法，可以传递自定义的错误消息。

2.check_score函数引发了InvalidScoreError异常，并提供了特定的错误消息。

3.在try块中尝试调用 check_score 函数，except 块捕获 InvalidScoreError 异常，并输出详细的错误信息。finally 块无论是否发生异常，都会执行 finally 块中的代码。

模块

- 标准库
- 常用标准库
- 标准库的导入和使用
- 第三方模块
- import导入的三种方式
- 创建自己的模块

标准库

1.模块概念：模块是一个包含Python代码的文件，通常以.py为后缀。它可以包含函数、类、变量等。

2.标准库概念：指在编程语言中，由该语言官方提供的预编写代码集合，这些代码实现了常见功能，供开发者直接使用。它们通常与编程语言的解释器或编译器一起分发，因此也被称作“内置库”或“核心库”。标准库的存在极大地提高了开发效率，因为它提供了丰富的基础功能和工具，避免了重复造轮子的工作。

tip：查阅标准库可以到Python标准库官方文档

常用的标准库

模块	作用	适用场景
os	提供了与操作系统交互的功能，包括文件路径操作、环境变量获取等。	文件管理、自动化脚本
sys	提供对Python运行时环境的访问，如命令行参数、解释器版本等。	获取命令行参数、强制退出程序
math	包含数学运算相关的函数和常量，比如三角函数、对数函数、圆周率等。	计算面积、体积、科学计算
datetime	处理日期和时间，支持日期计算、格式化输出等功能。	记录日志、计算时间差、定时任务
json	用于解析和生成JSON格式的数据，便于与Web服务进行数据交换。	网站接口、配置文件
random	生成伪随机数，支持从序列中随机选择元素等。	抽奖、游戏、随机测试数据

举例

1.random模块

```python
import random
# 随机整数（掷骰子）
print(random.randint(1, 6))  # 输出：1~6的随机数

# 随机选择（抽签）
colors = ["红", "蓝", "绿"]
print(random.choice(colors))  # 输出：随机选一个颜色

# 打乱列表（洗牌）
cards = ["A", "K", "Q", "J"]
random.shuffle(cards)
print(cards)  # 输出：["K", "A", "J", "Q"]（随机顺序）
```

2.os模块

```python
import os

# 获取当前工作目录
current_dir = os.getcwd()
print("当前工作目录:", current_dir)

# 创建新目录
new_dir = "example_dir"
os.mkdir(new_dir)
print(f"创建新目录: {new_dir}")

# 列出目录内容
files = os.listdir(current_dir)
print("当前目录下的文件和文件夹:", files)

# 删除目录
os.rmdir(new_dir)
print(f"删除目录: {new_dir}")
```

import的几种使用方法

- import 模块：通过模块名来访问其中的函数、类和变量
- from 模块 import 方法：直接导入模块中特定的函数，类或者变量
- import模块 as 别名：给导入的模块定义一个别名，在代码中使用这个别名来访问模块内容

tip：一般使用第一种和第三种比较多

标准库的导入和使用

1.import 模块

```
import math
print(math.sqrt(25))
```

2.from 模块 import 函数

```
from math import sqrt
print(sqrt(9))
```

3.import 模块 as 别名

```
import datetime as dt
now = dt.datetime.now()   # 用 dt 代替 datetime
print(now)
```

第三方模块

- 概念：第三方模块是指不是Python标准库自带的模块，而是由其他开发者创建并分享出来的。这些模块可以帮助我们更方便地完成特定的任务，比如处理数据、操作文件、构建Web应用等。

比如 requests等

requests是一个简单易用的HTTP库，用于发送HTTP请求

- 用pip或者pip3下载第三方模块：pip install 模块
- 导入第三方模块：import 模块
- 下载国外的需要用到清华镜像源：

```
pip install requests -i https://pypi.tuna.tsinghua.edu.cn/simple
```

- 删除安装好的包：pip uninstall 模块
- 查看安装好的包：pip list

创建自己的模块

案例：

```python
# calculator.py

|解释|添加注释|×
def add(a, b):
    """返回两个数的和"""
    return a + b

|解释|添加注释|×
def subtract(a, b):
    """返回两个数的差"""
    return a - b

|解释|添加注释|×
def multiply(a, b):
    """返回两个数的积"""
    return a * b

|解释|添加注释|×
def divide(a, b):
    """返回两个数的商"""
    if b == 0:
        raise ValueError("除数不能为零")
    return a / b
```

```python
# main.py

# 导入自定义模块
import calculator

# 使用模块中的函数
|解释|添加注释|×
def main():
    try:
        num1 = float(input("请输入第一个数字: "))
        num2 = float(input("请输入第二个数字: "))

        print(f"{num1} + {num2} = {calculator.add(num1, num2)}")
        print(f"{num1} - {num2} = {calculator.subtract(num1, num2)}")
        print(f"{num1} * {num2} = {calculator.multiply(num1, num2)}")
        print(f"{num1} / {num2} = {calculator.divide(num1, num2)}")
    except ValueError as e:
        print(f"输入错误: {e}")

# 主函数
if __name__ == "__main__":
    main()
```

面向对象编程

- 面向对象编程
- 面向对象编程的核心概念
- 继承和多态

面向对象编程

面向对象编程的核心思想是：把现实世界中的事物抽象成“对象”，并通过这些对象来组织代码。

对比过程式编程

过程式编程：以函数为中心，按顺序执行代码，适合解决简单问题。

面向对象编程：以对象为中心，将数据（属性）和操作数据的方法（行为）封装在一起，适合解决复杂问题。

面向对象编程

为什么使用面向对象编程？

1. 优点

模块化：代码结构清晰，易于维护和扩展。

复用性：通过继承和组合，减少重复代码。

灵活性：支持多态和动态绑定，适应复杂需求。

2. 应用场景

游戏开发：角色、道具等都可以抽象为对象。

Web 开发：用户、订单等业务逻辑可以用类表示。

数据科学：机器学习模型、数据集等常用类封装。

面向对象编程的核心概念

类的定义：在面向对象编程中，类是一种定义数据结构（属性）和行为（方法）的模板。它们是对象的蓝图，用于创建具有特定属性和方法的对象。在Python中，我们使用class关键字来定义一个类。

对象的实例化：对象实例化是指根据类创建具体对象的过程。每个对象都是类的一个实例，具有类中定义的属性和行为。在Python中，我们通过调用类名并传递任何必要的初始化参数来创建一个对象实例。

面向对象编程的核心概念

实例属性定义：是属于某个具体对象的变量，每个对象都有自己独立的实例属性，互不影响，实例属性通常在类的构造方法 __init__ 中定义。

语法：

```
class 类名:
    def __init__(self, 参数1, 参数2, ...):
        self.属性名1 = 参数1  # 定义实例属性
        self.属性名2 = 参数2
```

实例方法：是定义在类中的函数，用来操作实例属性或执行与对象相关的操作，实例方法的第一个参数必须是 self，表示调用该方法的对象本身。

语法：

```
class 类名:
    def 方法名(self, 参数1, 参数2, ...):
        # 方法体
        pass
```

```python
# 定义一个类
解释 | 添加注释 | ×
class Car:
    解释 | 添加注释 | ×
    def __init__(self, color, brand):
        self.color = color  # 属性：颜色
        self.brand = brand  # 属性：品牌

    解释 | 添加注释 | ×
    def start(self):  # 方法：启动
        print(f"{self.brand} 汽车已启动！")

# 创建对象
my_car = Car(color="红色", brand="宝马")
print(my_car.color)  # 输出：红色
my_car.start()       # 输出：宝马汽车已启动！
```

用class定义一个Car的类，__init__初始化Car的属性（colour和brand），再定义一个方法start。

继承和多态

继承定义：是面向对象编程的一个重要概念，它允许一个类从另一个类中继承属性和方法。通过继承，我们可以实现代码复用，并且能够建立类之间的层次关系。

父类（基类或超类）：被继承的类。

子类（派生类）：继承了其他类的类。

```python
# 定义一个父类
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")

# 定义一个子类
class Dog(Animal):  # Dog 继承自 Animal
    def speak(self):  # 重写父类的方法
        print(f"{self.name} says woof!")

# 使用
dog = Dog("Buddy")
dog.speak()  # 输出: Buddy says woof!
```

继承和多态

子类继承父类的属性和方法。

方法重写：子类可以重写父类的方法，以提供特定于该子类的功能。

super.()调用父类：子类可以通过super() 函数调用父类的构造函数来初始化从父类继承的属性。

```python
class Animal:
    def speak(self):
        print("This animal makes a sound.")

class Dog(Animal):
    def speak(self):  # 重写父类的 speak 方法
        print("The dog says woof!")

class Cat(Animal):
    def speak(self):  # 重写父类的 speak 方法
        print("The cat says meow!")

# 使用
dog = Dog()
cat = Cat()

dog.speak()  # 输出: The dog says woof!
cat.speak()  # 输出: The cat says meow!
```

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # 调用父类的 __init__ 方法
        self.breed = breed

    def speak(self):
        super().speak()  # 调用父类的 speak 方法
        print(f"{self.name} is a {self.breed} and says woof!")
```

继承和多态

多态定义：指的是不同类的对象通过相同的接口调用表现出不同的行为。换句话说，即使不知道变量引用的具体类型是什么，仍然可以通过统一的接口与它们交互，多态性通常基于继承，即多个类继承自同一个基类并各自实现了相同名称的方法，但这些方法的行为可能不同。

```python
class Animal:
    | 解释 | 添加注释 | ×
    def speak(self):
        pass

| 解释 | 添加注释 | ×
class Dog(Animal):
    | 解释 | 添加注释 | ×
    def speak(self):
        print("汪汪汪")

| 解释 | 添加注释 | ×
class Cat(Animal):
    | 解释 | 添加注释 | ×
    def speak(self):
        print("喵喵喵")

| 解释 | 添加注释 | ×
def animal_sound(animal: Animal):
    animal.speak()

dog = Dog()
cat = Cat()

animal_sound(dog)  # 输出: "汪汪汪"
animal_sound(cat)  # 输出: "喵喵喵"
```