学习目标：

1. 知道异常的概念
2. 会捕获异常
3. 熟悉异常的传递
4. 会抛出异常

异常的概念

异常是什么？

程序在运行时，如果 Python 解释器 **遇到** 到一个错误，**会停止程序的执行，并且提示一些错误信息**，这就是 **异常**。

程序停止执行并且提示错误信息 这个动作，我们通常称之为：**抛出（raise）异常**

例如：

- 数学错误：`1 / 0`（ZeroDivisionError）
- 类型错误：`"2" + 2`（Type Error）
- 索引错误：`[1,2][3]`（IndexError）

如果不处理以上的问题，那么Python 会打印错误信息并终止程序：

```
# ...

print(1 / 0)  # 运行后崩溃，提示：ZeroDivisionError: division by zero

# ...
```

程序开发时，很难将 **所有的特殊情况** 都处理的面面俱到，通过 **异常捕获** 可以针对突发事件做集中的处理，从而保证程序的 **稳定性和健壮性**。

捕获异常

通过`try-except`结构，可以“捕获”异常并处理，避免程序崩溃。

基本结构：

```
try:
    # 可能发生异常的代码（"尝试"执行）
     risky_code
except 异常类型1:
```

```
    # 若发生"异常类型1"，执行这里的处理逻辑
    handle_error1
except 异常类型2:
    # 若发生"异常类型2"，执行这里的处理逻辑
    handle_error2
except Exception as e:
    # 记录日志...
    print(e.__traceback__.tb_lineno)                                    # 获取异常发生的行数
    print(e.__traceback__.tb_frame.f_globals['__file__'])               # 获取异常发生的文件名
else:
    # 若没有异常，执行这里（可选）
    no_error_code
finally:
    # 无论是否有异常，都会执行这里（可选，常用于释放资源）
    always_execute_code
```

补充说明：当 Python 解释器 ***抛出异常*** *时，****最后一行错误信息的第一个单词，就是异常类型***

案例： 除以0的异常

```
def divide(a, b):
    try:
        result = a / b
        print("计算结果：", result)
    except ZeroDivisionError:
        # 捕获"除以零"异常
        print("错误：除数不能为0！")
    else:
        # 无异常时执行
        print("计算成功")
    finally:
        # 无论是否异常，都会执行
        print("除法运算结束\n")

# 测试
divide(6, 2)  # 无异常：计算结果：3.0 → 计算成功 → 除法运算结束
divide(6, 0)  # 有异常：错误：除数不能为0！ → 除法运算结束
```

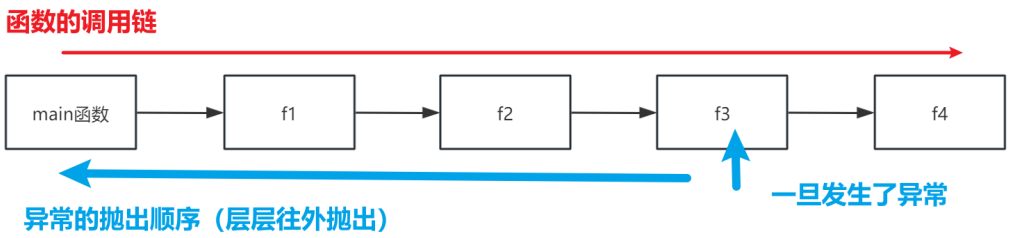
练一练： 要求用户输入整数

1 要求用户输入一个整数

2 使用8除以用户输入的整数并输出

异常的传递

1. **异常的传递** —— 当 **函数/方法** 执行 **出现异常**，会 **将异常传递** 给 函数/方法 的 **调用一方**
2. 如果 **传递到主程序**，仍然 **没有异常处理**，程序才会被终止
3. 在开发中，可以在主函数中增加 **异常捕获**
4. 而在主函数中调用的其他函数，只要出现异常，都会传递到主函数的 **异常捕获** 中
5. 这样就不需要在代码中，增加大量的 **异常捕获**，能够保证代码的整洁



举个例子：

- 定义函数 demo1() **提示用户输入一个整数并且返回**
- 定义函数 demo2() 调用 demo1()
- 在主程序中调用 demo2()

```python
def demo1():
    content = input("请输入一个整数:")
    num = int(content)
    return  num

def demo2():
    demo1()

if __name__ == '__main__':
    # 直接调用
    # demo2()

    # 也可以在主程序主动捕获异常
    try:
        demo2()
    except Exception as e:
        # 处理异常
```

传递路径解析：

1. demo1中发生ValueError异常，但是没有处理，于是传递给调用它的demo2
2. demo2中并没有处理异常，于是继续向外抛出，抛给调用它的main程序

3 main程序中如果没有处理异常，那么直接报错并终止程序继续执行

main程序中如果处理了异常，那么程序进入异常处理的流程并继续往下执行

异常传递的终止条件：

1 **被某个层级的 `try-except` 捕获**：如上述例子中，主程序的 `except` 捕获异常后，传递终止；

2 **到达程序顶层仍未被捕获**：此时程序会打印异常信息并崩溃。

抛出异常

在开发中，除了 **代码执行出错** Python 解释器会 **抛出** 异常之外， 还可以根据 **应用程序 特有的业务需求 主动抛出异常**

基本语法与使用

基本语法：`raise 异常对象`

示例：提示用户 **输入密码**，如果 **长度少于 8**，抛出 **异常**


```python
def get_password():
    password = input("请输入密码:")

    # 如果密码长度小于8，抛出异常
    if len(password) < 8:
        raise Exception("密码长度不正确")

    # 否则，返回密码
    return  password

if __name__ == '__main__':
    try:
        password = get_password()
```

```
        print(f"密码是:{password}")
    except Exception as e:
        print(e)
```

注意事项：

1. 使用raise关键字主动抛出异常
2. raise后面，是一个异常对象，这个对象是可以自定义的
3. 异常抛出后，仍然需要由外部的其他函数来捕获并处理异常

断言异常

断言（assert） 是一种用于调试的工具，当断言条件不满足时，会触发**AssertionError 异常**（断言异常）。它的核心作用是“在代码中嵌入检查点”，确保程序运行时的某些关键条件必须为真，否则直接暴露问题（方便调试）。

断言的语法非常简单，用于判断一个条件是否成立：

```
assert 条件表达式, 错误信息（可选）
```

执行逻辑：

- 如果条件表达式为True，那么断言不做任何事情，程序继续
- 如果条件表达式为False，则立即抛出AssertionError 异常，并 附带错误信息

案例： 计算列表的平均值

```
def calculate_average(numbers):
    # 断言：输入的列表不能为空（否则计算平均值无意义）
    assert len(numbers) > 0, "列表不能为空，无法计算平均值"
    return sum(numbers) / len(numbers)

# 正常情况：列表非空
print(calculate_average([1, 2, 3]))  # 输出：2.0

# 异常情况：列表为空，触发断言异常
calculate_average([])  # 抛出AssertionError: 列表不能为空，无法计算平均值
```

常见异常类型汇总

常见的异常如下：需要大家记住

编号	异常类型	触发场景	示例
1	`TypeError`	数据类型错误	`"2" + 2`（字符串 + 整数）
2	`ValueError`	数据值合法但不符合要求	`int("abc")`（字符串无法转整数）
3	`IndexError`	序列索引越界	`[1,2][3]`（列表索引 3 不存在）
4	`KeyError`	字典键不存在	`{"name": "张三"}["age"]`
5	`ZeroDivisionError`	除以零	`1 / 0`
6	`AttributeError`	访问对象不存在的属性 / 方法	`Student().score`（Student 类无 score 属性）
7	`NameError`	变量未定义	print(x)
8	`AssertionError`	断言失败	深度学习检查中常见
9	`FileNotFoundError`	打开的文件不存在	在后面文件章节中将会用到

所有的异常类型，可参考Python官方文档

自定义异常（重要性低）

我们之前见过的，不管是`Exception`，还是`ValueError`，或者是`KeyError`等等，都属于Python内置的异常类型。

那么我们能不能自己定义异常类型呢？

自定义异常类型很简单，让这个异常直接或者间接的继承`Exception`即可。

举个例子：模拟用户登录

需求：

1. 用户输入用户名，用户名长度必须在3-8位之间
2. 用户输入密码，密码长度必须在8-10位之间

```python
class UsernameError(Exception):
    def __init__(self):
        pass    # 空实现，使用默认的异常行为

class PasswordError(Exception):
    def __init__(self, length, min_len = 8, max_len = 10):
        self.length = length        # 保存错误的值
```

```
        self.min_len = min_len
        self.max_len = max_len

        # 调用父类的__init__，设置异常描述信息（通过super()传递）
        super().__init__(f'密码长度不合法,长度必须在{min_len} ~ {max_len} 之间')

def login():
    username = input("请输入用户名:")
    password = input("请输入密码:")

    # 长度必须在 3-8位之间
    if len(username) < 3 or len(username) > 8:
        raise UsernameError()

    if len(password) < 8 or len(password) > 10:
        raise PasswordError(len(password))



if __name__ == '__main__':
    try:
        login()
    except UsernameError:
        print("用户名不合法")
    except PasswordError:
        print("密码不合法")
    # 兜底
    except Exception as e:
        print(e)
    else:
        print("用户名和密码都合法，校验通过...")
```

注意： 在调用类的地方，用`except`捕获自定义异常，就可以根据不同异常类型执行针对性处理。