# 程序运行中的错误

- 程序运行中出错的两种基本情况：
  - 程序中某些部分 (编写的) 有错，但有错部分在一些特殊情况下才会执行，只有某些特殊的数据组合才会暴露错误，但由于程序复杂，测试中没发现这种错误
  - 程序对正确数据 (来自用户输入/文件/其他数据源) 能正确工作，但不能处理运行中遇到的错误数据 (值/类型等不合要求)
- 对于这两种情况，通常会希望
  - 程序 (应用系统) 不会因此终止执行或者产生严重后果
  - 一部分程序出错，不应该把错误传导到系统的其他部分，不应该影响其他部分的正常行为
  - 出错造成的影响应尽可能局部，整个系统还应该能 (在某种合理的意义下) 继续工作

# 错误的检查

- 处理错误的前提：发现错误
  - 发现错误的基本方式检查程序中处理的数据
  - 系统在内部定义的操作中会自动完成一些检查
  - 编程序时，需要在适当的地方加入检查数据错误的代码
- 不做检查，出错后状态非法，错误可能传到其他地方，引发更严重后果
  - 在可能出错的地方写检查代码称为防御性程序设计 (一种编程理念)
  - 尽可能让各代码单元保护自己，可以提高程序的可靠性
- 几类典型的检查
  - 读入数据应该检查得到的数据是否满足需要
  - 对参数有要求的函数，可以考虑检查参数的情况
  - 执行有可能出错的操作 (例如除法) 之前，检查操作数

# Python 程序中的异常和异常处理

- 对运行中错误的处理，是编程序时应该考虑和处理的问题
  - 语言需要提供尽可能好用的支持机制
  - 允许编程人员灵活定义具体的处理方法
- 异常处理机制：Python 系统对程序运行中错误的报告和处理
  - 在执行程序过程中 Python 解释器检查发现的错误均报告异常
  - 自己编写的程序在检查数据中发现问题时，也可以报告异常
  - 语言提供特殊结构，用于描述发生异常之后的处理和如何继续
- 如果发生异常，程序中没有处理，则该异常变成未处理异常
  - 未处理异常最终表现为致命错误，导致程序的执行终止
  - 在 IDLE 里，致命错误使系统回到交互状态，输出错误信息
  - 在直接执行的程序，致命错误导致程序结束，可能输出一些信息

# 捕捉和处理异常

- **try 语句**：Python 专用于捕捉和处理异常的语言结构
  - 用于描述程序执行中发生异常后的处理操作和流程
  - 程序里，可以通过 try 语句控制发生异常后的处理方式
- 例： val = f(n) + 1/val 可能出现除数为 0，但不希望程序终止

```
try:                                # try 结构的开始
    val = f(n) + 1/val              # 这里 val 可能为 0 (发生除 0 错误)
except ZeroDivisionError:           # except 段
    print("variable val is 0 as a divider!")
    val = 1                         # 具体处理应根据程序的实际需要
```

  - 如果 try 块语句组执行中不出错，则继续执行 try 结构之后的代码
  - 如果 try 块执行中发生 ZeroDivisionError 异常，则执行 except 段的代码，之后继续执行 try 结构之后的代码

# 简单形式的 try 语句

- try 语句的最基本语法：顺序地包含以下几个段
  - try 段：一个，由 try 关键字引导，后跟一个语句组
  - except 段：一个或多个，每个 except 段称为一个异常处理器
- try 语句的执行：首先，执行 try 段的语句组
  - 如果执行中不出现异常，执行完这个语句组之后，整个 try 语句的执行结束
  - 如发生异常，立即终止语句组的执行，转去查找匹配的异常处理器
    - 顺序检查异常处理器，找到第一个匹配的处理器就转进去执行其语句块 (体)，后续的处理器不再被考虑
    - 异常处理器的代码正常执行完时，整个 try 语句结束
    - 异常处理器执行中又发生异常，则跳出该 try 结构向外围报告
  - 如果没找到匹配的处理器，异常将自动向外围传播

# 异常处理器

- 异常处理器的基本形式：

```
except 表达式:      # 表达式描述所要捕捉和处理的异常
    语句组          # 描述对所列异常的处理(以及后续控制)
```

  - 最简单情况，表达式是一个异常名
  - 也可以是多个描述异常的表达式的元组 (需要用圆括号括起)，表示处理一组异常
- try 语句中，如果有多个异常处理器，应正确排序
  - 相互无关的异常处理器可任意排列
  - 更一般的处理器放在后面
  - 不带表达式的处理器捕捉所有异常，显然应该 (也只能) 放在最后；用来描述前面专用处理器都不能处理的情况，提供一种最后的处理方式
- (演示)

# 预定义异常

- Python 有一组预定义异常，参见标准库手册第 5 节
  - 其中包括所有预定义异常和引发它们的情况，请自行查阅
- 编程中可能根据其意义参考和借用：
  - ZeroDivisionError，除数为 0 错误
  - FloatingPointError，在数值计算程序里发现数值有问题
  - ArithmeticError，涵盖所有算术计算错误的一般异常
  - ValueError，对象类型正确但值不符合需要
  - KeyError，字典访问时所给关键字不存在
  - TypeError，参与计算的对象类型不对
  - IndexError，非法索引
  - NameError，所引用的对象属性不存在
  - FileNotFoundError，没有找到具有给定名字的文件
  - IOError，输入/输出操作失败
- 还可以使用 Python 的面向对象功能自己定义特殊的异常

# 异常的传播

- try 结构可以任意嵌套，例如

```
try:                          # 外层 try 语句
    ... ...                   # 外层 try 语句组开始
    try:                      # 内层 try 语句
        ... ...               # 内层 try 语句组
    except xxxError:          # 内层异常处理器
        … …
    … ...                     # 外层 try 语句组结束
except yyyError:              # 外层异常处理器
    ... ...
```

  - 在内层 try 段的执行发生异常，首先在内层处理器中找匹配
  - 如果内层处理器都不能处理该异常，异常则会传到外层 (继续匹配)
- 在一个函数执行中发生的异常，如果在该函数内部没有被捕获和处理，该函数的执行立即结束，异常在调用函数的位置重新引发
- 异常处理器的查找可能导致一层层函数调用结束，甚至导致主程序终止

# raise 语句

- **raise 语句**：用来在程序里主动引发异常

  **raise 表达式**

  **raise**

  - 简单情况，**raise 内置异常名**

    如：**raise ValueError**

  - 也可 **raise 内置异常名(实参表)**，提供任意多个、任意类型的实参以说明异常的具体情况 (适用于大部分内置异常)

    如：**raise ValueError("Wrong arguments!")**

  - **不带表达式的 raise**：重新引发之前发生的最后一个异常 (一般用于异常处理器的语句组里)；如当时没有异常，则引发 **RuntimeError**

  - 对于执行 **raise** 语句主动引发的特定异常 (对象)，其后续处理与系统引发异常完全相同

# raise 语句的典型应用场景

1. 函数定义中，在检查函数参数之后，通过 **raise** 语句报告错误
   - 某些函数对参数 (类型/值等方面) 有限制，为保证函数调用有意义和安全执行，常需要在函数体开始处检查实际参数
   - 一旦发现函数实参错误，恰当的做法是引发适当的异常 (演示)
   - 通过 **raise** 语句所生成的异常对象，可以从异常引发点往异常处理点 (即异常处理器) 传递任意多项信息
2. 用异常控制程序的流程
   - 规范的流程控制：顺序结构、条件结构和循环结构，以及函数调用和退出
   - **continue, break, return** 语句能 (有限制地) 突破规范控制流的约束
   - 异常的引发 (和传播) 会中断当前执行流，也可作为流程控制机制 (演示)

# try 语句的完整结构

- 完整的 try 语句：可顺序地包含
  1. 一个 try 段，其中应该只包含抛出预期异常的语句 (组)
  2. 一个/多个 except 段，每段定义一个异常处理器
  3. 一个 else 段 (可缺)，由关键字 else 引导，后跟一个语句组
  4. 一个 finally 段 (可缺)，由关键字 finally 引导，后跟一个语句组
- 两个可选段的执行规则
  - 当有 else 段，try 语句组执行中没有发生异常时，则执行 else 段的语句组；else 段执行中发生的异常将向外传播
  - 当有 finally 段，无论 try 语句组的执行是否发生异常，结束这个 try 结构之前，最后都执行 finally 段的语句组
    - 用于描述 try 结构的清理动作 (无论如何都要做的操作)
    - 如果是异常进入，在执行完 finally 段后重新引发原来的异常
    - 如 finally 段的执行通过 break 或 return 结束，原有异常抛弃

# 实例：统计文件中的数据情况

- 需求：写程序统计一批文件里数值数据的情况，包括
  - 每个文件中合法数据 (可转换为浮点数) 的项数
  - 每个文件中错误数据 (不可转换为浮点数) 的项数
  - 所有文件中合法数据之和
- 思路：
  - 定义函数 **stat_datafile** 处理单个文件
    - 文件相关操作、数据转换和统计，结果输出和返回
  - 文件操作时可能出错，何处引发异常？何处处理？
  - 数据转换时可能出错，何处引发异常？何处处理？
- (演示)

# with 语句

- **with 语句：处理一类特定计算过程的特殊 Python 结构 (文件处理是这类计算过程的一种典型情况)**
  - 假设程序中要打开一个文件，读入文件内容，简单写法：

```
infile = open("datafile.txt")
… …                    # 读入文件内容和处理
infile.close()
```

  - 在文件读入和处理的过程中可能出错，应该考虑用 try 语句
    - 文件的打开和关闭需要配对：被打开的文件，无论后续操作是否正常进行，最后都应该关闭；即，合理的写法如下

```
infile = open("datafile.txt")
try:
    … …                # 读入和处理文件的过程可能出错
finally:               # finally 段里的语句总会被执行
    infile.close()
```

# with 语句

- 这类计算过程的共性：(1)一段操作需要被执行，但在处理之前需要执行某种进入操作，之后需要执行某种退出操作；(2) 无论具体处理是正常或者异常结束，最后的退出操作 (如释放重要的资源等) 都必须执行
  - 可以用 try-except-finally 模式描述
  - 也可使用专门的语言结构 — with 语句 — 来简化描述
    - 例如，上页的代码片段可改写为

```
with open("datafile.txt") as infile：
    … …                          # 通过 infile 读入文件内容并处理
# 无论文件读入过程如何结束，都会自动地执行 infile.close()
```

- with 语句的基本形式

```
with  表达式  [as  变量名]:        # with 语句被称为上下文 (context)
    语句组
```

# with 语句

- **with 语句的语义**
  - 执行时，解释器首先求表达式的值，所得对象是一个**上下文管理器 (context manager)**
    - 该类对象支持一对进入操作和退出操作，这对操作采用特殊名字，能被自动调用
      - **__enter__()** 描述进入上下文 **(with 块)** 的动作
      - **__exit__()** 描述退出上下文时的清理动作 **(简单情况调用时不需要参数)**
  - 解释器求出上下文管理器对象后，自动地执行其进入操作，并将操作的返回值约束于 **as** 关键字后的变量，然后执行语句块
  - 无论语句组的执行情况 **(正常结束或者发生异常)**，执行上下文管理器对象的退出操作后再退出上下文

# with 语句

- Python 系统和标准库中的一些类型定义了进入和退出操作，包括文件对象类型等，可以直接用于 with 语句
  - 对于文件对象：其进入操作是空操作，返回该文件对象自身 (self)；其退出操作是关闭文件 (演示)
  - (利用类定义可以实现自定义上下文管理器对象)
- with 语句的头部允许有多个 as 子句
  - 出现几个 as 子句相当于几个嵌套的 with 语句
    - 排在后面的，先退出
  - 如下面两段代码等价

```
with A( ) as a, B( ) as b:
    … …
```

```
with A( ) as a:
    with B( ) as b:
        … …
```