其他-3，错误和异常

- 模块和执行，执行Python程序
- 命令行和命令行参数
- 运行中的检查和错误处理
- 异常：
  - 概念，处理结构和实例
  - try, except, raise 和 finally
- 处理中文（将提供一些材料，不讲了）

模块和执行

- 一个定义好的 Python 模块有两种使用方式
  - 作为程序的主模块，直接启动运行
  - 作为辅助模块，被其他模块导入（import）和调用
- 一个大些的程序通常由一个主模块和若干辅助模块组成

  程序从主模块启动，主模块导入其他模块并使用其中的功能
- 对辅助模块，有时也需要独立考虑和检查其功能，例如在开发和修改测试阶段。利用 Python 的模块命名机制可以方便地处理这两种情况
  - 程序运行时总有一个全局变量 __name__，在程序执行中的每个时刻，这个变量都具有一个自动设置的字符串值
  - 如果通过导入的方式执行一个模块（py 文件），__name__ 将被设置为该模块的 py 文件的名字，当一个 py 文件被作为主模块启动时， __name__ 的值设置为特殊名字 "__main__"
  - 检查 __name__ 的值，就可以判断本模块是以什么方式运行

模块和执行

- 用 **Python** 编程的人有个习惯：为每个模块写一段服务于测试或其他在将其用作主模块时需要执行的代码（如测试驱动代码），形式是

```
if __name__ == "__main__"
    # 作为主模块时执行的代码
    ... ...
```

  - 作为主模块运行时，这段代码就会执行，因此很适合写测试代码
  - 如果本模块被其他模块导入执行，这段代码将被忽略
- 实例（**rational.py**，**comp18-words.py**）
- 在今后写程序时，如果是某种服务性的模块，其中定义了一些可供其他程序使用的功能，都应该采用这样的写法
- 显然，对于主程序模块，其中在定义了各种函数、全局变量之后，应该有一段实际启动程序的代码

  除非是在交互式调试的阶段

Python 程序的执行

- IDLE 是前面一直使用的 Python 编程环境，在其中可以方便地
  - 编写修改 Python 程序，运行、测试和调试
  - 也可以用其他编辑器或专门开发环境，如 PyCharm
  - 但开发完成后，Python 程序应该能脱离 IDLE 独立地运行
- 正常安装 Python 后，安装程序应该已经设置好程序路径，把 Python 解释器的相关目录加入可执行程序的查找路径，可以直接启动执行
- 命令行

  python abc.py　　启动 python 解释器运行程序 abc.py

  python words.py

  python /?　　查看启动 python 系统时可以用的参数

  可以看到许多启动参数，指明启动参数的情况或运行方式

命令行参数

- 在以命令行方式启动 **Python** 程序时，还可以为被执行的程序提供“参数”，供程序在运行中使用
  - **Python** 程序 **prog.py** 以命令行方式启动，在命令行中 **nnnn.py** 后面可以写任意多个空格分隔的“参数”，称为“命令行参数”
  - 需要通过标准库包 **sys** 取得和使用
- 在 **Python** 程序 **prog.py** 里取得命令行参数的方法
  - 在 **prog.py** 里导入标准包 **sys** 后使用 **sys.argv**，或者从 **sys** 导入 **argv** 后直接使用**argv**，这是一个字符串的表
  - 以命令行方式启动 **python prog.py … …**，在该程序运行时，表 **argv** 里的每个字符串是一个命令行参数
  - **argv[0]** 就是这个 **Python** 程序的名字 **"prog.py"**，**argv** 的其他成员顺序为命令行中 **prog.py** 之后的各命令行参数
- 看例子，调用程序文件 **python words1.py sun1.txt sun-stat.txt**

运行中的错误和处理

- 在程序运行中可能出错，例如
  - 数值计算或转换中出现除 0 或计算结果溢出，调用标准数学函数时参数不满足函数要求（log, sqrt, asin, 等）
  - 程序输入遇到不满足要求的数据（如超范围的浮点数），交互式程序，从网络或外设获取信息的程序可能遇到这类情况
  - （编程引起）偶发的运行错误，如访问表元素时的下标越界
  - 程序用完了可用资源（如建立新表时已无空闲内存等）
- 实例：
  - 美国约克顿号巡洋舰，操作员不当输入了一个 0
  - 阿丽亚娜5型火箭首次发射，加速度高导致数据转换溢出
  - 美国某火星飞船控制系统在途中周期性崩溃并重新启动

程序运行中出错

- 程序运行中出错，能随便允许它崩溃/停止运行吗？
  - 对简单的实验性程序，可以容忍（前面许多程序都是这样）
  - 对复杂的系统，通常不能允许
    - 编辑器？文字处理程序？Python 系统？游戏？
    - 各种实时控制系统？（交通工具/核电站/大型设备控制），与社会平稳安全运行有关的系统，如银行/股票交易系统？安全攸关的系统？
- 由于计算机的特点和威力（通用性和通过具体程序实现的专用性等），计算机越来越广泛地介入人类社会的各个领域
  - 程序运行中出错可能造成的威胁也越来越严重
  - 应设法保证程序不出错（正确性/可靠性）？或在出错时合理应对，使之不造成或少造成实质性的损害（容错）？

程序错误

- 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，但运行中遇到错误数据（值/类型/数量等不合要求）
- 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做某些“坏事”
  - 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

检查错误

- 处理错误的前提是发现错误
  - 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
- 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
- 不做检查，出错后状态非法，错误还可能传到其他地方，后果难以预料
  - 在可能出错的地方写检查代码称为防御性程序设计（一种理念）
  - 尽可能让各代码单元保护自己，可以提高程序的可靠性

处理错误

- 同学们编程中也遇到了一些情况
  - 程序输入需要转换到浮点数，数据不符合要求时转换出错
  - 统计文本时遇到了法语字母，不能转换到 ASCII 编码
- Python 系统执行程序中会检查一些错误
  - 如：ZeroDivisionError，IndexError，KeyError 等
  - 系统有一组预定义错误名，在一些操作中检查并可能报错
  - 如果出错后不处理，就会导致程序终止并显示错误信息
- 自己编写的程序也可能检查并发现错误，如参数/输入不符合需要等

  最简单的想法是报告错误（如用 print），问题是然后怎么办？
- 无论系统检查或自己检查，发现了错误之后都可能需要处理
  - 具体应用确定合适的处理办法，如文件打不开，是跳过还是终止
  - 需要有一种统一而强大的处理框架，允许编程中定制具体处理方式

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
- Python 把运行中错误的报告和处理统一到一套称为异常处理的机制下
  - Python 解释器系统检查发现的错误都报告异常
  - 自己编写的程序在检查中发现问题时，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，该异常就变成未处理异常
  - 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并输出一些信息

异常和异常处理

- Python 处理异常的结构是 try 语句
  - 用于描述程序执行中发生异常后的处理操作和流程
  - 人可以控制：发生异常后是让程序终止，还是适当处理后继续执行
  - 很多程序需要出现异常情况后还能继续工作，特别是很多重要程序
- 例：如果计算中可能出现除数为 0，但这时不希望程序终止，可以写：

```
try: # 表示一个 try 结构开始
    val = f(n) + 1/val
except ZeroDivisionError:
    print("0 divider occurs for variable val")
    val = 1  # 具体处理要根据程序的需要
```

  - 如果 try 的语句块执行中不出错，接着执行 try 结构之后的代码
  - try 块执行中发生 ZeroDivisionError 异常时，执行 except 段的代码，而后继续执行 try 结构之后的代码

简单的 try 语句

- try 语句的最基本形式是顺序的几个段：
  - 一个 try 段，由 try 关键字引导，后跟一个语句块
  - 一个或多个 except 段，一个这种段称为一个异常处理器
    - 每个异常处理器用一个或多个表达式描述自己处理的异常
    - 没写表达式的处理器处理一切异常，只能放在最后
- try 语句的执行（语义）：立即开始执行 try 段的语句块
  - 如果执行中不出现异常，执行完这个语句块后整个 try 语句结束
  - 执行中出现异常，转去查找匹配的异常处理器，顺序检查，找到匹配的处理器就转进去执行其语句块
    - 异常处理器的代码正常执行完时整个 try 语句结束
    - 异常处理器执行中发生异常，跳出该 try 结构向外层报告
  - 如果没找到匹配的处理器，异常将自动向外传播（后面详细介绍）

异常处理器

- 异常处理器的基本形式和意义：

  except 表达式 **...:**
  　语句块

  except 后的表达式可有多个，描述本异常处理器捕捉和处理的异常

- 异常处理器用于捕获异常并描述对异常的处理
  - 处理器头部的表达式描述本处理器捕捉的（那些）异常
  - 体（代码段）描述对异常的处理（和后续控制）
- **try** 基本语句段的执行中发生异常时，顺序检查后随的异常处理器
  - 执行第一个能捕捉发生的异常处理器，后面处理器不再考虑
  - 如果写多个处理器，应正确排序，更一般的处理器写在后面
  - 不写表达式的处理器捕捉所有异常，显然应该（也只能）写在最后，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

异常处理实例

- 实例，程序中要求必须输入一个整数：

```
while True:
    try:
        x = int(input("Please enter an integer: "))
        break
    except ValueError:
        print("Error! Input was not a valid integer. Try again ... ")
```

- 实例：需要一个数据文件，从用户获得文件名

```
while True :
    fname = input("Please give the file name: ")
    try :
        infile = open(fname)
        break
    except OSError:
        print("Cannot open " + fname + ".", "Another...")
```

异常处理实例

- 现在考虑一个“通用的” 带检查输入函数，引入两个参数：
  - 一个类型参数，用于类型转换和类型错误
  - 一个提示串参数，用于要求用户输入时的提示
- 函数定义：

```
def inputValue(valType, requestMsg):
    """Generic function for input a value of a given type."""
    while True:
        val = input(requestMsg + " ")
        try:
            val = valType(val) # 检查能否转换到所需类型
            return val
        except ValueError:
            print(val + " can't convert to " + str(type) + ". Another...")
```

- 调用实例：x = inputValue(int, "Please input an integer: ")

异常处理

- try 结构可以任意嵌套，例如

```
try:
    ... ...      # 外层语句块开始
    try:
        ... ...  # 内层语句块
    except xxxError:
        ... ...  # 内层异常处理器
    ... ...      # 外层语句块结束
except yyyError:
    ... ... # 外层异常处理器
```

  - 在内层 try 块执行发生异常，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，异常就会传到外层

- 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发生

异常处理实例

- 如果多处需要从用户得到数据文件名，可定义函数

```
def getDataFileName ():
    while True :
        fname = input("Please give the file name: ")
        try :
            infile = open(fname)
            return infile
        except OSError :
            print("Can't open " + fname + ".", "Another...")
```

这个函数一直循环，直至得到了一个能正常打开的数据文件

- 有可能用户需要有中断这种反复过程的机会（例如，确实没有文件了）
  - 这里遇到一个困难：怎么安排函数的返回值
  - 一种可能是返回 None，要求调用函数检查（但可能被忽视）
  - 另一方法是主动引发异常，要求调用函数的程序处理（下面介绍）

raise 语句

- 在程序里，可以用 **raise** 语句主动引发异常
- **raise** 语句，形式和语义：

  raise 表达式

  raise

  - 表达式给出引发的异常，常用内置异常名加一个字符串参数
  - **raise** 的执行引发指定异常，其后续处理与系统引发异常相同
  - 不带表达式的 **raise** 重新引发前面发生的最后一个异常，通常用在异常处理器的语句块里。当时没有异常时引发 **RuntimeError** 异常
- **raise** 语句的典型使用：函数检查参数之后报告错误
  - **Python** 函数定义对参数没有限制，而许多函数对参数有要求
  - 为保证函数调用有意义和安全执行，经常需要在函数体开始处检查实际参数。但一直存在的问题是：发现参数错误后怎么办？
  - 前面采用返回特殊值只是权宜之计，更好方法是引发适当的异常

参数检查和异常

- 前面的简单函数定义：

```
def triangle(a, b, c) :
    if a>0 and b>0 and c>0 and a+b > c and a+c > b and b+c > a :
        s = (a + b + c) / 2
        return sqrt(s * (s - a) * (s - b) * (s - c))
    else:
        return 0.0
```

错误结果将在计算中传播，有危险。返回 None 要求每个调用检查

- 用引发异常的方式处理：

```
def triangle(a, b, c) :
    if a>0 and b>0 and c>0 and a+b > c and a+c > b and b+c > a :
        s = (a + b + c) / 2
        return sqrt(s * (s - a) * (s - b) * (s - c))
    else:
        raise ValueError("wrong argument(s) for func triangle")
```

预定义异常

- Python 有一组预定义异常，详见标准库手册第 5 节

  那里列出了所有预定义异常和引发它们的情况

  请自己查阅

- 编程中可能使用（根据意义考虑，借用）：
  - ZeroDivisionError，例如用在 rational 的构造函数里
  - FloatingPointError，如在数值计算程序里发现数值有问题
  - ArithmeticError，更一般的情况
  - ValueError，例如发现值不符合需要
  - KeyError，如发现字典的关键码不符合需要
  - TypeError，如发现函数参数类型不对

  这都是建议，自己根据情况考虑。实际上还可以使用 Python 的面向对象功能自己定义异常，下面会介绍

try 语句

- 使用送给异常（名）的参数
  - 送给异常的参数将传到捕获异常的处理器
  - 在处理器头部描述异常的表达式之后可以加 **as var**

    var 应是一个变量，处理器捕捉到异常时，异常值传给这个变量

    在异常处理器的语句组里可以通过该变量使用异常的信息

```
except ValueError as msg :
    print("Find error:", msg)
```

- 完全可以通过异常传更多信息，例如对 **triangle** 函数

```
raise ValueError("wrong argument(s) for function triangle",
                 (a, b, c))  # 异常带有一个三元序对
```

返回的多个项被做成 **tuple**，在处理器里可以取用其中的信息

通过这种参数可以从检查错误处向处理错误的程序段传递任意信息

try 语句

- 一个完整的 try 语句顺序包含
  - 一个 try 段，由 关键字 try 引导
  - 一个或多个 except 段
  - （可选的）一个 else 段，由关键字 else 引导
  - （可选的）一个 finally 段，由关键字 finally 引导
- 两个可选段的执行
  - 如果有 else 段，try 语句组执行中不出现异常时，就执行 else 段的语句组。else 段执行中发生的异常将向外传播
  - 如果有 finally 段，无论是否发生异常，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（无论如何都要做的事情）
    - 如果是异常进入，执行完 finally 段后重新引发原来的异常
    - 如果通过 break 或 return 结束，原有异常抛弃

一个例子：统计文件里数据的情况

- 假设要定义一个函数，统计一个数据文件里的数据情况
  - 共计多少合法数据项（浮点数），有多少错误数据，计算总和
- 计划：
  - 打开文件（但文件可能打不开）
  - 统计数据，可能出现转换错误，处理异常并正确统计
- 所用技术：
  - 文件处理
  - 字符串处理
  - 异常处理

用异常作为控制机制

- 程序运行中发生异常，将终止当前执行流，去寻找异常处理器
  - 这也是一种“控制转移”，这种转移可能退出任意多层的程序结构，包括退出函数，可以在编程中利用
  - 例：迭代器结束引发**StopIteration**异常，**for**语句处理
- 例：退出多层循环，**Python** 没有直接处理机制
  - 可以自己组合各种机制，引进必要的控制变量，一层层检查情况并退出
  - 另一种可能性是利用异常处理机制
    - 在需要退出的内层循环中某些条件下用 **raise** 引发异常
    - 在相应外层用 **try-except** 结构捕捉异常
    - 注意，**StopIteration**无法穿过**for**循环

关键字

- 已经讨论过的关键字（31/33）

  and as assert break class continue def del
  else except elif False finally for from global
  if in import is lambda None nonlocal not or
  pass return raise True try while with yield

- 分类：
  - 特殊字面量 None, True, False
  - 运算符 and, or, not, in, is
  - 基本语句 assert, break, continue, pass, return, del, yield，模块导入 from-import, 变量声明 global 和 nonlocal
  - 控制语句 for-in, while, if-elif-else
  - 函数定义 def，描述函数的特殊表达式 lambda
  - 异常处理 except, finally, raise, try
  - 重命名 as（用在 import, except 等处）