数据抽象和面向对象编程

- 为什么考虑数据抽象和面向对象编程
- **Python 类 (class)**
  - 类和对象
  - 创建对象
  - 对象方法
- 编程实例
- 类和对象基本情况的总结
- 通过继承定义新类

程序组织

- 随着程序越来越复杂，需要有更好的方法组织程序结构
- 前面讨论的各种程序结构，其中最大的功能单位是函数，用于构造计算过程的抽象 (过程抽象/函数抽象)
  - Python 允许函数定义嵌套，通过嵌套定义的一组函数和局部变量，可以用函数包装复杂的功能
- 但是，在实际应用中，仅有函数抽象可能不够，或者不够方便
  - 许多计算功能需要围绕一类数据对象描述 (定义)；例如，定义一个有理数功能模块，能简化 (规范化) 操作有理数的程序
  - 围绕一类数据“对象”的程序部件实际上包含两个方面特征
    - 一种数据对象的规范表示 (如用整数二元组表示有理数)
    - 一组与该类数据有关的操作，构成使用所定义数据抽象的接口 (例如，从有理数计算得到新的有理数等)

数据组织和用户定义类型

- **Python** 提供了一批内置类型，每个类型均是一种数据抽象
  - 表示一类数据 (如整数、表等)，有一组可用操作
  - 用一个名字 (类型名) 代表这类对象
  - 提供了一种 (或几种) 字面量描述方式
- 程序里可以利用 **Python** 内置的组合类型实现所需的数据组织
  - 实际情况：内置类型不可能满足编程序者的所有需求
- 合理的解决方法：允许编程序者定义所需的类型 (即，用户定义类型 **user-defined types**)
  - 定义新类型也是在扩充语言，但较函数定义复杂，因为一个类型不仅要关注一类数据的表示，还需要定义一组相关操作
  - 同时，用户定义类型应该具有与内置类型同等的地位，从使用方式上无区分

数据抽象和类、对象

- Python 支持数据抽象的编程概念是 类 (class) 和 对象
  - 整个 Python 语言围绕类和对象的概念构造
  - Python 的所有内置类型都是类，例如

```
>>> type(1)
<class 'int'>                    # 整数 1 的类型是名字为 int 的类
>>> type(print)
<class 'builtin_function_or_method'>
    # print 的类型是一个内部定义的、包含所有内置函数的类
```

- Python 语言是面向对象的编程语言
  - 类和对象是面向对象编程的两个最基本概念
  - Python 不仅以类和对象作为实现的基础，同时支持基于类和对象的编程

数据抽象和类定义

- **Python 定义新类型的语言结构：类定义**
  - 一个类定义：为程序引入了一个新的数据类型 (即类对象)，具有给定的名字，可以通过名字使用
    - 程序里，可以对类对象进行实例化 (这是定义类的最主要目的)，从而建立该类的实例 (称为该类的实例对象或对象)
  - 类定义中，一般描述了其实例 (对象) 的表示方式，以及相关的操作
- 类定义的基本形式：

```
class Class_Name:
    语句组          # 类的定义体 (简称 类体)

# 定义最简单的类 (空类)
class NullClass:
    pass
```

数据抽象和类定义

- **Python 定义新类型的语言结构：类定义**
  - 一个类定义：为程序引入了一个新的数据类型 (即类对象)，具有给定的名字，可以通过名字使用
    - 程序里，可以对类对象进行实例化 (这是定义类的最主要目的)，从而建立该类的实例 (称为该类的实例对象或对象)
  - 类定义中，一般描述了其实例 (对象) 的表示方式，以及相关的操作
- 类定义的基本形式：

```
class Class_Name:
    语句组          # 类的定义体 (简称 类体)
```

  - 类的定义体：位于类体表层，可以包含任何语句
    - 最常见的是一系列局部函数的定义 — 类的函数属性
    - 也可以用一些赋值语句建立一组局部变量 —类的数据属性 (或称类变量 class variable)

类定义的基本情况

- 一个类的函数属性称为本类的方法，其中最常见的是实例方法，用于操作本类的实例对象
  - 实例方法的调用：由本类的一个实例对象出发，用圆点记法
  - 实例方法的定义：第一个形参表示实际调用时的实例对象，通常取名 **self** (根据实际需求，还可以有其他参数)
  - 类的初始化方法：在一个类定义里，(通常有) 一个名为 **__init__** 的实例方法
- 类的实例化：用表达式 **Class_Name(…)**
  - 求值该表达式时，解释器将创建 **Class_Name** 类的一个新的实例对象，并自动地对该对象执行 **__init__** 方法
- 类定义是一种复合语句
  - 执行效果：建立起所描述的类对象，并将其与类名约束，表示一个数据类型

Python 类定义 (初步)：有理数类

- 有理数类的实例应该是有理数，类定义中描述有理数的有关操作
  - 需要为有理数类命名 (PEP8 建议类名采用大写字母开头的名字，需要时可以分段大写，如 BigData)
  - 需要考虑如何存储一个有理数的信息
  - 需要定义有理数的行为，包括该类对象有关的各种操作
- 首先考虑实现一个简单的有理数类 RationalSimple，用于：
  - 创建有理数对象：例如， RationalSimple(3, 5) 应创建一个分子为 3 分母为 5 的新有理数对象
  - 支持有理数运算：需要为有理数定义一组操作 (运算)，实现各种有用的有理数运算

  (演示)

简单有理数类

- 一些情况：
  - 初始化方法 __init__：为实例对象设置属性值
    - 实例对象可以有任意多个属性，赋值即创建 (例如，每个有理数对象都有两个属性 num, den)
  - 属性引用使用圆点记法
    - 实例方法的调用：由本类实例出发，使用圆点记法
    - 实例方法里，对实例对象的数据属性的访问，也通过圆点记法
- 上述简单有理数类的功能有限，且存在一些缺陷：
  - 初始化方法里没有检查参数，无法保证所建有理数对象合法
  - 创建的有理数对象没有标准化：数学中常要求分母为正；计算中也没有做有理数化简
  - 希望能将有理数的输出纳入内置函数 print 之中

改进的有理数类 (1)

- 建立有理数实例对象时，初始化方法 __init__
  - 需要检查参数的类型和值的合法性
  - 需要对有理数进行规范化，检查分子/分母的符号，并约简
- 问题：约简有理数时，需要用到求最大公约数的函数 gcd ，应该把 gcd 函数定义在哪里？
  - 定义为全局函数？所定义的函数与有理数类无关，可以随处使用
    - 但另一方面，gcd 作为实现有理数类的辅助函数，基于信息局部化原则，并不应该定义为全局函数
  - 定义在有理数类里？但注意：gcd 的计算不依赖于有理数类的实例，其参数也不是有理数类的实例，并不应该定义成以 self 作为第一个形参的实例方法

改进的有理数类 (2)

- 在类体里，可定义静态方法：仅在该类内部使用，但不以本类的实例为操作对象的局部普通函数
  - 描述形式：在函数定义的头部之前加修饰符 @staticmethod

```
class Rational:
    @staticmethod          # 定义一个仅限内部使用的静态函数
    def _gcd(m, n):        # Note：静态函数不应该有 self 形参
        if n == 0:
            m, n = n, m
        while m != 0:
            m, n = n % m, m
        return n
```

- 在类定义内部，静态函数应该通过类名或者本类的实例以圆点记法调用
- Python 约定：以 _ 开头的属性名都是类内部私有的名字，不应该在类外部访问 (一种信息隐藏的机制，虽然只是约定但应该严格遵循)

改进的有理数类 (3)

- 有理数实例对象的两个属性：也应该作为内部私有属性，不应该在类定义的外部被直接访问 (有理数应该是不变对象) ➔ 命名时应以 _ 开头

```
    def __init__(self, num, den=1):
        if not isinstance(num, int) or not isinstance(den, int):
            raise TypeError                    # 参数类型错，抛出异常
        if den == 0:
            raise ZeroDivisionError            # 分母为0，抛出异常
        sign = 1
        if num < 0:
            num, sign = -num, -sign
        if den < 0:
            den, sign = -den, -sign
        g = Rational._gcd(num, den)            # 调用本类的静态方法 _gcd
        self._num = sign * (num // g)          # 设置内部私有属性
        self._den = den // g
```

改进的有理数类 (4)

- 增加实例方法 num 和 den：提取有理数实例对象的分子、分母的接口

```
def num(self):  return self._num        # den 方法类似，略
```

  - 更符合 Python 风格的做法：用 @property 装饰器创建公开的、同名的只读特性 (见演示)

- 有理数是数学类型，希望能使用 + - * / 等运算符描述计算过程，从而沿用算术表达式的形式

- Python 规定了一套以双下划线开始且以双下划线结束的特殊方法名，以模拟内部类型的操作 (包括算术运算等)，通过运算符或系统内置函数自动调用

  - 完整的列表见Python 语言参考手册 3.3 节 Special method names，其中有与各种算术运算有关的特殊方法名，包括：

+: __add__	-: __sub__	*: __mul__
/: __truediv__	//: __floordiv__	**: __pow__
%: __mod__	(还有一元运算符、各种逻辑运算符等)	

改进的有理数类 (5)

- 定义特殊实例方法，实现有理数的算术运算

```
def __add__(self, another):          # 模拟/重载 + 运算符
    den = self._den * another.den()
    num = (self._num * another.den() +
           self._den * another.num())
    return Rational(num, den)
```

  - 定义中通过实例方法 sum 和 den 访问 another 对象的私有属性
  - 用类名 Rational 构造作为计算结果的对象，保证具有规范的形式

```
def __floordiv__(self, another):          # 模拟/重载 // 运算符
    if another.num() == 0:                # 检查除数
        raise ZeroDivisionError           # 可能抛出异常
    return Rational(self._num * another.den(),
                    self._den * another.num())
```

(其他运算类似)

改进的有理数类 (6)

- Python 也为各种关系运算符提供了特殊方法名

  ==: __eq__　　!=: __ne__　　<: __lt__

  <=: __le__　　>: __gt__　　>=: __ge__

- 定义特殊实例方法，实现有理数的比较运算：

```
def __eq__(self, another):          # 重载 == 运算符
    return (self._num * another.den() ==
            self._den * another.num())

def __lt__(self, another):          # 重载 < 运算符
    return (self._num * another.den() <
            self._den * another.num())
```

(其他运算类似)

改进的有理数类 (7)

- 有理数运算要求另一个参数 another 也是有理数对象，可以在实例方法增加参数检查，类型不正确时引发 TypeError 异常

```
if not isinstance(another, Rational):
    raise TypeError
```

  - 另外，也可以允许有理数与 int 作各种运算

```
if isinstance(another, int):
    another = Rational(another)
```

- 内置函数 str 对应的特殊方法名为 __str__，实现从类实例对象到字符串的转换

```
def __str__(self):
        return (str(self._num) + "/" + str(self._den))
```

  - 之后则可以用内置函数 print 输出有理数实例对象

- 继续完善扩充，则可以定义功能完善的有理数类 (参见演示)

类和对象基本情况的小结 (1)

- 执行一个类定义 ➔ 建立一个类对象 (c.f. 实例对象)
  - 类定义中包装了一批属性，解释器处理类定义时为类对象建立相应的属性，其中包括
    - 函数属性：可调用属性，由类定义表层的函数定义描述
    - 数据属性：由类定义表层的赋值语句描述
  - 对每个类对象，都有数据属性 __doc__，其值是该类体 (开头) 的文档串；如该类没有文档串，其默认值为空串
- 对于类对象，主要支持两类操作：
  - 属性访问：由类名出发取得/使用属性的值 (如 Rational._gcd(...))，或者给类对象的属性赋值 (但不常用)
  - 实例化：通过类名以函数调用形式建立本类的实例对象
- 当一个类对象建立之后，还可以通过属性赋值的方式为这个类对象增加新的属性，包括数据属性和函数属性

类和对象基本情况的小结 (2)

- 一个类定义确定一个作用域
  - 类体表层的定义是局部定义，在该类之外，需要通过圆点记法访问
  - 类体表层的局部名字 (数据/函数属性名) 的作用域不自动延伸到该类内部嵌套的作用域 (例如类内的函数定义，局部类定义)
- 类定义被执行时，解释器在创建类对象时建立一个新局部名字空间，类体中所有语句都在该空间产生效果
- 程序中的每个实例对象都有一个局部名字空间，其中包含该对象的所有属性及其约束值 (如实例对象为空，则其名字空间为空)
- Python 的面向对象机制 (包括类对象和实例对象的构造) 是动态的
  - 一个类对象创建后，可以通过属性赋值的方式增加或修改该类对象的数据/函数属性 (但这种情况不常见)
  - 允许先创建空实例对象，之后再逐步加入/扩充需要的属性 (但不建议这样操作，因为无法保证同一类型的不同对象之间的统一性)

类和对象基本情况的小结 (3)

- 类定义里，最常见的是各种实例方法的定义
  - 实例方法：都以 **self** 为第一个形参名，代表方法的调用对象
  - 实例方法的调用：从一个类实例对象出发，采用圆点记法和实参表
    - 调用对象被自动地约束到第一个形参 **self**
  - 初始化 **__init__** 方法：创建本类实例对象时被自动地调用
    - 其 **self** 参数代表所创建的实例对象
    - 功能：给所创建的实例对象建立数据属性，设置初始状态
  - 在一个方法定义里，如果需要访问同类的其他属性，必须基于类名或实例对象采用圆点记法，例如 **self.den()**
  - 需要时可以在类的函数定义里写 **global** 或 **nonlocal** 声明

类和对象基本情况的小结 (4)

- 例：设有一个类 C，其中定义了带三个形参的实例方法 m；并设变量 x 的值为类 C 的实例对象 o，即 x = C(…)
- 表达式 x.m：值为一个方法对象 (也称绑定方法 bound method)
  - 其中包含了两个成分：实例对象 o 和函数对象 m (类似于闭包)
  - 方法对象的最常见的用法：直接调用
    - 如 x.m(a, b)，调用时对象 o 将被作为 m 的第一个实参
  - 方法对象也是对象，可赋给变量，传入函数，作为函数返回值等
    - 如执行 p = x.m 后，可用 p(a, b) 来调用该方法对象
- 表达式 C.m：值为一个普通函数对象
  - C.m(x, a, b)：等价于 x.m(a, b)

类和对象基本情况的小结 (5)

- 类里可以定义静态方法 (前加 @staticmethod 修饰符)
  - 用途：实现与本类实例对象无关的方法 (没有 self 参数)，如一些用于实现实例方法的辅助功能函数
    - 本质上，静态方法就是在类定义体里的普通函数
- 类里还可以定义类方法 (前加 @classmethod 修饰符)
  - 用途：用于实现操作类对象，或者本类所有实例对象有关的方法
  - 类方法的第一个参数均表示其调用类，常用 cls 作为形参名；之后还可以有任意多个其他参数
  - 执行类方法时，调用类对象自动约束到 cls 形参
    - 类方法的定义体里，可通过 cls 参数访问该类的其他属性
- 静态方法和类方法的调用：由类名出发采用圆点记法 (也可以由类实例对象出发，但不建议)

类和对象基本情况的小结 (6)

- 例 (类方法)：要求定义一个带计数功能的类，其中包含一个计数器用来统计已创建的本类实例对象的数量
  - 注意：实例对象的个数与单个实例对象无关，是该类整体的性质，应该用类的数据属性记录 (而不能用实例对象的数据属性记录)

类和对象基本情况的小结 (6)

- 例 (类方法)：要求定义一个带计数功能的类，其中包含一个计数器用来统计已创建的本类实例对象的数量

```
class Countable:
    counter = 0                          # 数据属性，记录已创建本类实例的数量

    def __init__(self):
        Countable.counter += 1           # 每次创建实例，更新计算器

    @classmethod                         # 类方法
    def get_count(cls):
        return cls.counter               # 取得计数器当前值
```

```
if __name__ == "__main__":
    x = Countable()
    y = Countable()
    z = Countable()

    print(Countable.get_count())
```

类和对象基本情况的小结 (7)

- 类实例对象的属性，也称为实例属性/变量
  - 通常都在初始化 __init__ 方法里设置，以便维持新建实例对象 (初始状态) 的统一性
  - 可以在其他实例方法里修改实例对象的属性
  - 采用 _ 开头的属性，表示仅供实例方法使用，即私有属性
- 对于类定义中以双下划线开头 (但不以双下划线结尾) 的名字，在类定义之外不能进行属性引用
  - 解释器自动进行名称改写 (name mangling)，属于简单的保护措施
- 为实例对象设置属性时，如果所设置的属性与类定义中的 (数据或函数) 属性重名，将覆盖同名属性 (应注意避免名字冲突)
  - 例如，设类 C 中有实例方法 m，变量 x 的值为 C 的实例对象 o，即 x = C(...)，则属性赋值 x.m = 3 后，x.m 的值不再是方法对象而是 x 的属性引用 (但 C.m 仍是原函数对象)

类和对象基本情况的小结 (8)

- 类里定义的实例方法一般可以分为三类：
  1. 构造操作：基于一些已知信息创建新实例对象
  2. 解析操作：取得实例对象的信息/属性
  3. 变动操作：修改已有实例对象内部状态/属性
- 在有理数类里，只有两类实例方法：构造和解析
  - 该类的实例对象在创建之后状态不会改变，即是不变对象，相应的类型即是不变类型
- 实例对象属性的值决定了实例的状态
  - 如果在类里定义了变动操作，其执行时就会修改对象的状态
  - 定义了对象变动操作的类实现的是可变类型 (类似 list 等)，其实例对象就是可变对象
  - 演示：简单计数器类 (可变类型和对象)

类定义进阶：继承

- 继承是定义类的重要机制和技术
  - 作用一：可以基于已有的类定义新的类，重用已有类的功能，减少定义新类的工作量
  - 作用二：建立一组类型之间的继承 (层次) 关系，处于下层的类型是上层类型的子类型 (具有类似的行为，但可能有些扩充)，有利于更好地组织和构造复杂程序
- Python允许基于继承的类定义方式
  - 基类/父类/超类：被继承的已有类
  - 派生类/子类：通过继承定义出的新类
  - 派生类自动继承其基类的所有数据属性和方法属性
  - 派生类也可以根据需要修改基类中的某些属性 (即重新定义这些属性)，或者扩充新的功能 (即定义新的属性)

继承

- Python 中定义了一批内置类型，各种基本类型形成层次结构
  - 例如，内置异常都是类，之间的一般/特殊关系就是由继承构成的层次结构
  - 包含一个最基本的内置类型 **object**，其中定义了一些所有对象都需要的功能
    - 如果类定义的头部没有说明基类，该类自动以 **object** 为基类
    - 任何用户定义类都是 **object** 的直接或者间接派生类
- 定义派生类的语法形式：

```
class derived-class-name(base-class-name, …):
    语句组          # 定义体
```

  - 在括号里，可列出一个或多个指定的基类名

(演示：可循环移位表类型)

基类和派生类

- 在概念上，派生类是其基类的特殊情况，派生类的对象集合是基类的对象集合的子集
- 内置函数 **issubclass(cls1, cls2)** 检查两个类型是否具有继承关系，包括直接或间接的继承关系
  - 如 **cls2** 是 **cls1** 直接或间接的基类，返回 **True** ,否则返回 **False**
- 设变量 **x** 的值是类 **D** 的实例对象
  - **type(x) is D** 值为真， **isinstance(x, D)** 值为真
  - 对于 **D** 的任何基类 **C**
    - **isinstance(x, C)** 值为真， **type(x) is C** 为假

➔ 一个类的实例对象也看作其所有直接或间接基类的对象，派生类是基类的子类型

替换原理

- 在面向对象编程中有一个重要的设计规则 — 替换原理：要求派生类的实例对象能够用在要求其基类的实例对象的程序上下文中
  - 派生类和基类的实例对象具有同样的合理行为
  - 实际则要求派生类是基类的“保守”的功能扩充，原有的行为并不改变，称之为是“行为子类型”
- 在 **Python** 里定义派生类时，需要注意：
  - 初始化函数通常都需要重新定义，定义中必须正确初始化基类的所有数据属性，通常通过调用基类的初始化方法完成
  - 重新定义/覆盖基类已有方法时，首先要保证参数形式正确，还要保证支持基类原有的行为；也可以通过调用基类原有方法完成，即 **BaseClass.methodName(…)**
- **(显然，不能要求基类对象能用到要求派生类对象的上下文中)**

调用基类的方法

- 派生类的实例对象通常有更多的数据属性，定义时经常需要重新定义 __init__ 函数，完成该类实例的初始化

```
class DerivedClass(BaseClass):
    def __init__(self, ...) :
        BaseClass.__init__(self, ...)   # 首先调用基类的 __init__ 方法
        ... ...                         # 初始化函数的其他操作
    ... ...                             # 派生类的其他语句(和函数定义)
```

- 在派生类的方法定义里使用内置函数 super()，将要求从当前类的直接基类开始检索需要调用的方法，常在派生类覆盖基类方法时使用

```
class DerivedClass(BaseClass):
    def method(self, *args):
        super().method(*args)
        ... ...          # 派生类自己的附加操作
```

采用 super() 的好处是不用在派生类里直接写基类的名字，可减少类之间的相互关联，可能更有利于程序的修改；函数 super 还有其他的调用形式，请查阅手册

方法查找/解析 (1)

- 假设 x 的值是类 C 的一个实例对象，对于调用 x.m(...)，需要先找到 m 的定义，而后执行相应的函数 (相应的过程称为方法查找)
  - 即确定应该用哪个类里定义的名字正确 (名字为 m) 的方法
- Python 规定：
  - 首先，在该对象 (称为调用对象) 所属的类型 C 中查找是否有名字为 m 的函数属性，如果有就确定了要调用的方法
  - 如果没找到，就转到 C 的直接基类，重复同样查找过程
  - 上述查找过程持续到在 C 的某个基类找到 m 的定义；或者，到达类 object 仍未找到 m，则是属性无定义，报告 AttributeError
- 查找过程顺序进行，一旦找到就结束，从而
  - 从 x 出发，实际调用的一定是 C 里定义的方法 m，或者 C 的最近基类里定义的方法 m

方法查找/解析 (2)

- Python 允许在定义类的时候，指定任意多个基类
  - 如果一个类定义继承了不止一个基类，称为多重继承
  - 多重继承的派生类 (及其实例对象) 将继承所有基类的方法
    - 基类定义的类方法和实例方法，都可以通过派生类或者派生类的实例对象使用
  - 利用多重继承，可以构造集成多种功能的对象
  - 另一方面，多重继承也带来一些新问题，其中包括方法查找问题
- Python 设计了一种方法解析序 (method resolution order)
  - 对于类 C，根据继承关系，确定了一个方法解析序列，其中有序地排列着 C 类对象方法查找中可能涉及的所有类
  - C.mro() 返回类 C 的方法解析序列 (一个表)，C.__mro__ 返回一个内容相同的元组

方法查找/解析 (3)

- 如果在派生类里重新定义了某个方法，例如 m，由派生类对象出发调用 m 的时候，基类的同名定义不会被正常查找过程找到
  - 在定义子类方法时，如要调用基类的方法需要显式地从类名出发或通过 super 函数出发采用属性引用的形式
  - 对于方法 m 的查找，由调用对象的类型决定

```python
class B:
    def g(self):
        print("B.g is called.")

    def f(self):
        print("B.f is called.");
        self.g()

class C(B):
    def g(self):
        print("C.g is called.")


x = C()                 # 无初始化函数时生成空对象
x.f()                   # 输出什么？

print(C.mro())          # 检查属性查找的路径
print(C.__mro__)        # 也可以通过类对象的特殊属性 __mro__
```

动态约束：基于调用对象查找所需方法

按方法查找的规则，在 f 里调用 g 时应该根据 self 实参对象的类型查找

```
B.f is called.
C.g is called.
[<class '__main__.C'>, <class '__main__.B'>, <class 'object'>]
(<class '__main__.C'>, <class '__main__.B'>, <class 'object'>)
```

类和异常处理 (1)

- **Python** 里的异常基于类和对象的功能构造
  - 每种异常是一个类，异常名即是类名
  - 程序运行中产生的异常 **(**无论是系统引发还是 **raise** 语句引发**)**，就是构造相应异常类的一个实例对象
  - 异常处理器 **except** ***Ex_name*** 即是捕捉在相应 **try** 块的执行中引发的 ***Ex_name*** 异常类的实例对象
    - 注意“***Ex_name*** 的实例”的面向对象意义：这个处理器不仅捕捉 ***Ex_name*** 异常，还将捕捉 ***Ex_name*** 的所有子异常类的对象
  - 对于头部 **except** ***Ex_name*** **as e**，如果运行进入这个异常处理器，局部变量 **e** 自动约束到相应的异常对象 **(e.args** 则该异常对象所携带的信息元组，其成员是引发异常时的各参数**)**

类和异常处理 (2)

- Python 定义了一套内部异常类，它们构成一个树形结构
  - 所有异常类的基类是 BaseException
  - 其最主要的子类是 Exception，内置的异常类主要是这个类的直接或间接派生类

详细情况见标准库手册“第5节 标准异常”，5.4 节列出了所有标准异常及其类层次关系

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- ..
```

类和异常处理 (3)

```
+-- StopIteration
+-- ArithmeticError
|    +-- FloatingPointError
|    +-- OverflowError
|    +-- ZeroDivisionError
+-- AssertionError
+-- AttributeError
+-- BufferError
+-- EOFError
+-- ImportError
+-- LookupError
|    +-- IndexError
|    +-- KeyError
+-- NameError
|    +-- UnboundLocalError
+-- OSError
|    +-- FileExistsError
|    +-- FileNotFoundError
+-- ReferenceError
+-- RuntimeError
|    +-- NotImplementedError
+-- SyntaxError
|    +-- IndentationError
|         +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|         +-- UnicodeDecodeError
|         +-- UnicodeEncodeError
|         +-- UnicodeTranslateError
```

这里列出一些经常看到的异常，有关解释见标准库手册

类和异常处理 (4)

- 异常是类，且遵循类之间的子类型关系
  - 一个 try 语句后的一组异常处理，也应该按类继承关系进行组织

```
try:
    … …
except RationalError:            # 专门处理有理数除 0
    … …
except ZeroDivisionError:        # 处理一般除 0
    … …
except ArithematicError:         # 处理一切算术错误
    … …
except Exception:                # 处理用户和系统异常
    … …
except:                          # 处理所有问题
    … …
```

自定义异常

- 用户需要定义异常时，应该从系统异常类中选择一个合适的异常，从它派生出自定义异常类

```
class RationalError(ZeroDivisionError):
    pass
```

- 最简单/很常见情况：定义一种特殊异常以便区别处理，但并不需要这种异常有什么特殊功能
  - 简单处理：选一个系统异常类派生自定义异常类，类体并不需要定义任何属性
    - 为了语法完整，**pass** 语句作为类体
- 用户定义异常可以像系统异常一样引发和捕捉
  - 引发异常时，异常名后可以带一个实际参数表
  - 同样可以用 **except … as** 捕捉用户定义异常处理相关信息

容器和元素访问

- Python 中一些内置类型的对象可以包含多个元素，如 list/dict 等，可以取元素值或者给元素赋值 (如果是变动类型)
  - 采用下标表达式或下标赋值的形式，x = s[i] 或者 s[i] = y
  - 这类对象，常被称为容器对象 (container)
  - 如果某种自定义类型的对象也保存着一批元素，希望能采用上述形式来操作元素
- 特殊方法名： __getitem__ 和 __setitem__
  - 参数形式：

    obj.__getitem__(self, key)

    obj.__setitem__(self, key, value)
  - 如果一个类定义了这两个方法，其实例对象就能做下标访问
    - 实际操作由函数定义确定
    - 进一步结合 Python 的 slice 类型，可实现切片操作

容器的其他方法

- 对于容器，可能需要求其中的元素个数
  - 对标准容器，都可以用标准函数 len 求元素个数
  - 如果一个类定义中有实例方法 __len__，则对其实例对象调用 len 函数，系统就会自动调用该方法
- 对于标准容器类的对象，可以用 in 或 not in 运算符检查另一个对象是否其元素
  - 如果一个类定义中有实例方法 __contains__，则对该类的实例对象使用 in 或 not in 运算符时，会自动调用该方法
- 容器类还可以定义一个 __iter__ 方法，该方法的值应该是一个迭代器对象。例如，可以考虑定义一个生成器方法作为迭代器。复杂情况，也可以专门定义一个迭代器类 (只要它定义了 __next__ 方法，而且在迭代完成时引发 StopIteration 异常，在迭代完成后，再调用 __next__ 方法时总引发这个异常)
- (容器类演示)

数据抽象与软件 (1)

- 以数据为中心是目前广泛使用的软件开发理念，原因之一：
  - 数据更直接地对应于需要解决的实际问题
  - 人们对客观世界的研究和认识，总结为一些抽象或具体的概念，它们通常都具有信息内涵以及可能的变化 (操作)
  - 这些可以比较直接地映射到数据抽象，定义为抽象数据类型
- 客观知识体系都围绕着一批概念，例如：
  - 自然语言 (例如汉语) 处理中的汉字、词、短语、句子是不同层次的对象。各种对象有属性，例如动词/名词；相互搭配关系，等等
  - 商业流通领域的商品、价格、存量、利润、货架、仓库、卖场分区、供应商、客户、折扣、处理商品等等
  - 数学中的各种数、向量、矩阵、概念、定义、公理等
- 用程序中的对象模拟现实世界需要解决问题中的对象，已证明是用计算机解决实际问题的一种有效方法

数据抽象与软件 (2)

- 实现面向数据的程序设计需要
  - 总结所要解决的问题领域的一组最主要的概念，总结出其数据属性和行为特征
  - 设法建立相应的计算对象 (类/类型)，用这种对象的数据属性记录客观对象的信息，用相应操作反映客观对象的行为
- 采用基于数据抽象的设计，在计算机领域本身也有重要意义
  - 抽象数据类型是比函数更大的程序模块概念，应用起来更加灵活方便
  - 从技术上，既能关注实际问题在数据方面的性质，也能关注计算和操作方面的性质
  - 从规模上，具有更好的适应性，可以定义或大或小的模块
  - 从编程方法上，类定义严格区分对象的实现方式和使用接口，只通过类中定义的实例方法去操作对象
- (演示：客户管理)