19. 面向对象编程-I

- 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
- 类的继承（类层次结构，类功能的利用和修改，下次课的内容）

程序组织

- 随着程序越来越大，需要有更好的方法把程序组织起来
- 前面讨论的各种程序结构，其中最大的单位是函数，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，通过嵌套定义和一组函数和局部变量，可以把很复杂的（计算）功能包装在一个函数里
- 但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描述（定义）。例如，定义一个有理数功能包，能大大简化（规范）处理有理数的程序
  - 这种围绕着一类数据“对象”的程序实际上包含两个方面
    - 数据对象的表示。例如用一个二元组表示一个有理数
    - 一组操作。例如从有理数计算出新的有理数等

数据组织

- Python 提供了一批内置类型，每个类型是一种数据抽象
  - 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

  可以利用 Python 内置的组合类型实现程序所需的数据组织
- 前面讨论了组合的形式表示有理数的必要性，也借助于 Python 的数据组合机制设计了一种有理数实现，其中
  - 用两个元素的元组表示有理数，元素分别表示分子和分母
  - 定义了一组操作，定义了构造这种形式的“有理数”的函数，还有定义了基于这种元组实现各种有理数运算的函数
  - 定义这样一套函数，就像是在 Python 里实现了有理数

基于数据组合，有根本性的缺点

- 虽然上述技术可行，但也有一些明显的弱点（缺点）
  - 操作方式是通过下标访问元组元素，**r[0]**、**r[1]** 没有提示性。对于成分更多的对象，这样操作很容易写错
  - 这里建立和操作的只是规定形式的元组（两个元素都是整数的元组），而不是真正的有理数

    例：假如还需要表示整数格点，可能也会用二元组。这时无法区分两种不同数据，无法避免不正确的互操作
  - 相关的操作无法绑定于表示有理数的元组对象

    Python 的函数不描述参数类型，使问题更严重
  - 无法利用 **Python** 中各种与类型有关的机制
- 总结：虽然定义了有理数的表示，但不能将其区别于其他元组对象。这样的“有理数”只是一种表示形式，不是一种类型

用户定义类型

- 无论编程语言提供了多少内置类型，也不可能满足编程序的人的需要。解决问题的办法只能是允许编程序的人定义所需类型

  称为用户定义类型（**user-defined types**）

  定义新类型也是扩充语言，但这种定义比函数复杂，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

  进而，用户定义类型应该具有与内置类型同等地位，同样操作方式，从使用方式上无区分

- 总之，实现复杂程序时，经常需要围绕着一类数据组织程序模块，需要数据抽象机制。内置类型总不够用，需要用户定义类型
  - 实现数据抽象，就需要能把一类数据的表示方式（实现方式）和相关操作集合在一起，做成一个整体
  - 定义为类型，使之自然融入 **Python** 语言的编程环境中

数据抽象

- Python 支持数据抽象的编程概念是类（class）和对象

  整个 Python 语言也是围绕着类和对象的概念构造起来的

  Python 的所有内置类型都被看着是类，例如

  ```
  >>> type(1)
  <class 'int'>
  >>> type(print)
  <class 'builtin_function_or_method'>
  ```

  整数 1 的类型是名字为 int 的类，print 的类型也是类

- Python 语言被称为是一种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
  - 例如，int 是一个类型（类），0、1 等是这个类型的对象。下面考虑基于类和对象的编程，也就是面向对象的编程

类和抽象数据类型

- 类定义是 **Python** 语言里的一种程序结构，一个类（一个类定义）为程序引入了一个新的数据类型
  - 所定义的类具有给定的名字，可以通过名字使用
  - 程序里可以建立该类的实例，称为该类的对象或实例对象
  - 类定义中描述了其实例（对象）的表示方式和相关操作
- 类定义的形式很简单，基本形式是：

  class 类名:

  　　语句块

  其中语句块可以包含任何语句，最常见情况是一系列函数定义，也可以有一些赋值语句建立一组局部变量（称为属性）
- 与函数定义的情况类似，**Python** 把类定义也看成一种复合语句，其执行的效果就是建立起一个类对象，并将其与类名约束

类定义实例：初探

- 通过一个例子说明类定义的一些情况：有理数类
- 有理数类的实例应该是有理数，类定义应该描述有理数的行为
  - 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，即它们能做的各种运算
  - 需要为有理数类命名
- 选定 **Rational** 作为有理数类的名字，定义好的类可以：
  - 创建有理数：**Rational(3, 5)** 将创建一个有理数，建立起一个分子为 **3** 分母为 **5** 有理数对象
  - 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
  - 首先考虑一个朴素的实现

简单有理数类

- 类定义通常包含一些函数定义和一些用赋值描述的数据定义
  - 类里的函数定义描述该类的函数属性，定义的函数称为这个类的方法。最基本的是实例方法，可供本类的实例使用
  - 用赋值形式描述的是类的数据属性，有关情况后面介绍
- 定义有理数类是为了建立和使用有理数，考虑一些情况
  - 有理数对象需要创建和记录成分，为此需定义一个用特殊名字 __init__ 的初始化方法，建立有理数时自动调用这个方法
  - 有时需要取得有理数的分子或分母，为此需要定义两个方法，分别命名为 num 和 den
  - 需要做有理数计算，需要定义相应方法
- 在实例方法的定义中，总需要用第一个参数描述被操作的实例对象，这个参数通常取名 self。还可以有其他参数

简单有理数类

```
class Rational0:
    # 定义几个实例方法，其他类似
    def __init__(self, num, den=1):
        self._num = num  # 设置实例的数据属性
        self._den = den    # 用 _num 形式名字避免与方法名冲突

    def num(self): return self._num

    def den(self): return self._den

    def plus(self, another):
        n = (self._num * another._den +
                self._den * another._num)
        d = self._den * another._den
        return Rational0(n, d)

    def print(self):  # 输出有理数的方法，输出形式 (n/d)
        print("(" + str(self._num)+"/"+str(self._den) + ")")
```

简单有理数类

- 一些情况：
  - 从实例出发，通过圆点形式调用类里定义的实例方法
  - 实例方法里通过参数 **self** 以 **self.xxx** 形式访问实例的属性
  - 初始化方法为实例对象设置属性值，对象可以有任意多个属性，设置即建立。创建的每个有理数都有两个属性
  - 可以根据需要扩充其他操作（增加有理数运算）
- 这个简单有理数类可用，但也有些情况不令人满意
  - 建立的有理数不规范，数学中常要求分数的分母为正
  - 计算中没有做有理数化简。虽然Python可以表示任意大整数，计算结果正确。但表示大整数浪费资源，也不利于阅读
  - 有理数的输出最好能纳入内置 **print** 之中（可以考虑）

改造的有理数类

- 作为一个更好的有理数类，应该总建立规范的有理数
  - 需要检查分子/分母的符号，对其进行规范化
  - 需要考虑化简，保证创建的总是不可约简的有理数
- 化简有理数时需要求最大公约数的函数，这个函数在前面做过。现在的问题是，应该把这个函数定义在哪里？
- 两种可能：
  - 定义为全局函数，这样定义的函数与有理数类无关，可以随处使用，对于 **gcd** 这种通用函数，可以考虑
  - 定义在类里，但要注意：**gcd** 不以有理数对象作为操作对象，不应定义为以 **self** 作为第一个参数的实例方法
- 类定义中常需要一些在类内部使用，但不以本类的实例对象为操作对象的函数，**Python** 为处理这种问题提供了静态方法

改造的有理数类

- 现在考虑改造的有理数类的定义：

```
class Rational:
    @staticmethod  # 定义一个内部使用的静态函数
    def _gcd(m, n):  # 用 _ 开头的名字是 Python 惯例
        if n == 0:
            m, n = n, m
        while m != 0:
            m, n = n % m, m
        return n
```

- 在类的内部，静态函数应该通过类名以圆点形式调用
  - 静态函数没有 **self** 参数，与本类实例无关
  - 用 _ 开头的名字作为类的内部定义，在类外不应该使用。但这只是约定，**Python** 并不禁止在类外使用

改造的有理数类

- 新的初始化函数：

```
def __init__(self, num, den=1):
    if not isinstance(num, int) or not isinstance(den, int):
        raise TypeError
    if den == 0:
        raise ZeroDivisionError
    sign = 1  # 记录数的符号
    if num < 0:  # 记录分子符号
        num, sign = -num, -sign
    if den < 0:  # 记录分母符号
        den, sign = -den, -sign
    g = Rational._gcd(num, den) # 调用本类的 _gcd
    self._num = sign * (num // g)
    self._den = den // g
```

改造的有理数类

- 方法 num 和 den 的定义不变
- 现在考虑有理数运算
  - r1.plus(r2) 的形式不太令人满意
  - 有理数是数类型，最好能用内部数类型一样的 + - * / 运算符
- Python 规定了一套特殊方法名（函数名）
  - 完整的表见标准库手册 3.3 节 Special method names，包括 __init__，__str__ 等
  - 其中 3.3.7. Emulating numeric types 给出了与各种算术运算有关的特殊方法名，包括：

    +: __add__　　-: __sub__　　*: __mul__

    /: __truediv__　　//: __floordiv__　　**: __pow__

    %: __mod__　　还有各种逻辑运算符等

改造的有理数类

- 定义有理数的算术运算，加法：

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

用 Rational 构造，保证构造出有理数总具有正确的形式

- 除法，需要检查除数，可能抛出异常：

```
def __floordiv__(self, another):  # 模拟 // 运算符
    if another.num() == 0:
        raise ZeroDivisionError
    return Rational(self._num * another.den(),
                    self._den * another.num())
```

改造的有理数类

- 考虑定义有理数的比较运算，Python 也提供了特殊方法名

  ==: __eq__　　!=: __ne__　　<: __lt__

  <+: __le__　　>: __gt__　　>=: __ge__

- 方法定义：

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

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

  其他类似

改造的有理数类

- 实现有理数运算的方法都要求另一个参数 **another** 也是有理数，可以加检查，类型不正确时抛出 **TypeError** 异常

  可以考虑允许与 **int** 运算，在 **__add__** 里可以简单加一句：

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

- 为方便输出，可以利用到 **str** 的转换，特殊方法名为 **__str__**
  - 采用 **(num/den)** 的形式作为有理数的字符串表示
  - 定义下面转换方法：

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

改造的有理数类

- 有时可能希望把有理数转换到整数或者浮点数，定义：

```
def to_int(self):
    return self._num // self._den
def to_float(self):
    return self._num / self._den
```

混合运算时需要明确写出转换，如

```
x = 3 + (Rational(2, 3) + Rational(6, 5)).to_int()
```

- 注意：说“模拟”数值运算，这里不能做自动转换。如果希望实现 1 + Rational(2, 3)，需要另外定义一个方法（也有一组名字）

```
def __radd__(self, another):     #
    if not isinstance(another, int):
        raise TypeError
    another = Rational(another)
    ... ...
```

改造的有理数类

- 继续扩充，可以定义出一个完整的有理数类

  看看有理数计算的例子

- 目前定义的缺点：
  - 实现算术运算的实例方法中构造有理数，也像在类外构造有理数一样，做了很多无意义的检查
  - 这个事情也可以解决，方法是另外定义一个类内部使用的构造有理数的方法，其中省略不必要的检查

    方法的名字可以任意取，其中调用 **Rational.__new__()** 建立新的 **Rational** 对象

    技术和解释比较复杂，在代码里有例子，定义了一个名字是 **create** 的方法，这里不解释，有些情况后面介绍

类定义基本情况的总结

- 一个类定义的处理（计算）建立一个类对象
  - 就像函数定义建立起一个函数对象，而后可以调用执行。类定义建立的类对象主要用于创建类的实例对象
  - 可以通过类名以函数调用形式建立本类的实例对象
  - 类定义中可以包装一批属性，包括函数属性和数据属性
- 类定义里列出的定义（还可以有赋值）描述类的属性
  - **Python**处理类定义时为类对象建立这些属性
  - 可以通过类对象使用其属性，如 **Rational._gcd(…)**
- 注意区分由类定义创建的类对象和通过类实例化创建的实例对象

  如果变量 **x** 的值是类 **C** 的实例对象，**isinstance(x, C)** 成立

  可对任两个对象检查此关系，例如 **isinstance(None, int)**

类定义基本情况的总结

- 例如：
  - **Python** 处理 **Rational** 类定义，建立一个相应的类对象，并将其约束于类名 **Rational**
  - 通过表达式 **Rational(…)** 创建的是有理数类的实例对象
  - 前者可称为有理数类（有理数的类），后者可称为有理数
- 类定义里最重要的是一批表示实例方法的函数定义
  - 每个实例函数以 **self** 为第一个参数名，代表方法的调用对象
  - 实例方法从类实例对象出发，用圆点记法和参数表的形式调用，在调用时还需要为其他参数提供实参
  - 初始化方法以 **__init__** 为名，创建本类对象时自动调用。其 **self** 参数代表正在创建的对象。这个方法用于给创建的对象建立数据属性，设置初始状态（应建立良好的初始状态）

类定义基本情况的总结

- 通过一个类的实例对象，可以调用该类里定义的实例方法
  - 采用圆点调用形式，调用对象自动作为其第一个参数
  - 调用中应按需要提供其他参数
- 实例对象的属性通常都在初始化函数里设置
  - 一般采用 _ 开头的名字，表示仅供实例方法使用
  - 实例的属性可以在其他实例方法里修改
- 一般而言，类里定义的实例方法可以分为三类：
  1. 建立新对象的操作（创建操作或构造操作）
  2. 使用对象内部信息的操作（访问操作或解析操作）
  3. 修改已有对象状态的操作（变动操作）

类定义基本情况的总结

- 在有理数类里只有两类实例方法，创建和解析：
  - 这个类建立的对象，在创建之后的状态不会改变

    这种类型的对象就是不变对象
  - 这样的类定义出的类型即是不变类型（str 类等也是这样）
- 对象属性的值决定了对象的状态

  如果在一个类里定义了修改对象属性的方法，执行这种方法就会修改对象的状态

  修改实例对象属性的操作称为变动操作

  定义了对象变动操作的类实现的是可变类型（类似 list 等）

  可变类型的实例对象就是可变对象

类定义基本情况的总结

- 考虑一个简单的计数器类定义

```
class Counter:
    def __init__(self, init=0): # 初始化操作
        self._count = init
    def inc(self): self._count += 1          # 变动操作
    def dec(self): self._count -= 1          # 变动操作
    def value(self): return self._count      # 访问操作
    def reset(self): self._count = 0         # 变动操作
```

- 可以建立起任意多个计数器对象，每个维护一个计数状态
  - 建立起的计数器对象，通过操作可以改变其内部状态
  - 改变对象状态，并不改变其 id 函数的值（对象标识）

类定义基本情况的总结

- 类定义可以定义静态方法（前加 @staticmethod 修饰字）
  - 用于实现并不与本类实例有关的方法，例如一些用于实现实例方法的辅助功能函数
  - 内部使用的方法通常用下划线开头的名字
- Python 支持用特殊方法名模拟内部类型的操作，通过运算符或系统内置函数自动调用

  这种函数名有一大批，其中一些可用于模拟算术表达式

数据抽象与软件

- 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客观问题所在的真实世界
  - 人们对真实世界的研究和认识，总结为一些抽象或具体的概念，它们通常都具有信息内涵和可能的变化（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，具体例子如：
  - 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

数据抽象与软件

- 用计算中的对象模拟现实世界需要解决的问题中的对象，已证明是用计算机解决实际问题的一种有效方法，如此工作，需要
  - 总结需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，应用起来也更加灵活方便
  - 从技术上，既能关注计算的数据方面的性质，也能关注计算方面的性质
  - 从规模上，具有更好的适应性，可以定义或大或小的模块