20. 面向对象编程-2

- 通过继承定义新类
- 人事信息管理

  一组相关类的定义及其层次结构
- 类层次结构
  - 方法查找
  - 动态方法约束
- 异常和标准异常类

类定义进阶：继承

- 类不仅可以用于定义数据抽象，还能用于定义相互有关的一组数据抽象（一组类型）
  - 这样定义的类型之间有一种层次关系。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
  - 在许多计算环境中需要这样的一组类型
  - **Python** 面向对象机制使人可以通过对已有类型扩充的方式，比较方便地定义出一组相关的类型
- **Python**允许基于继承的方式，从一个或几个已有类（称为基类）定义一个新类，称为指定基类的派生类
  - 派生类自动继承其基类的所有数据属性和方法属性
  - 只需按一定的方式编程，就可以使派生类的实例对象具有与基类的对象同样的行为，还可以根据需要扩充

类定义进阶：继承

- 通过继承方式定义新类的语法

  class derived-class(base-class, …):

  语句块

  在类名的括号里，可以列出一个或多个基类的名字

  如果不明确写出基类，所定义的类将自动继承系统基类 **object**

  object 里定义了一些所有对象都需要的设施

- 下面从一个例子出发，讨论定义这样一组类型的必要性和用途，定义中可以使用的 **Python** 机制，以及可能遇到的问题的解决方法（技术）

- 假定现在需要为某大学建立一个管理人事数据的系统，其中需要存储大学中各种人员的信息

人事数据和基本类

- 大学人事管理系统中需要管理很多类不同的人员
  - 教职工，包括教师/实验室人员/行政人员/辅助人员等
  - 学生，包括本科生/硕士生/博士生/博士后/函授生/进修生等
- 仔细分析会发现：
  - 每类不同人员，管理中需要掌握的有关信息是不同的

    但又有一些共性，而且有些覆盖所有人员的共性，有些类人员之间的共性更多，有些类之间共性较少
  - 不仅有关的信息可能不同，对这些人员可能的操作也有不同

    但也有些共同的操作，不同类别人员之间的共同操作可能较多，也可能较少
- 这种情况在实际中很常见，怎样设计一组合适的数据抽象？

人事数据和类

- 真实世界中处理这类问题的方法也是分门别类
  - 教职员工由人事部门管理
  - 学生由教务部门管理，下面再分
- 要统一到一个人事管理系统里，可以考虑分层的数据组织

  Python 的面向对象机制支持分层数据抽象，其一般模式是首先定义最一般的，覆盖面最广的数据类型

  而后在基本类型的基础上逐步扩充，做出各种专门的类型
- 下面考虑先定义一个表示个人信息的基本类，然后利用 **Python** 的继承机制，逐步定义出一些特殊的人员记录类
- 在这一工作中，将展示 **Python** 有关面向对象编程的其他机制和技术，并说明实例方法的查找过程等一般性问题

基本个人信息类

- 为描述一个基本的社会人，需要选择一组基本属性。下面考虑的类只选择了三项属性：性别、出生日期和姓名

  其中性别和出生日期是一个人的基本生物属性

  姓名是服务于社会交往的基本属性

- 显然，个人信息记录应该表示为一类可变对象。在这种对象的生存期中，其中的信息可能变动

  这时就需要确定哪些基本属性应该是可变属性，哪些不是

  例如，我们考虑人可以修改姓名，不能修改出生日期和性别

- 基于这些考虑，可以定义出一个表示个人信息的类
  - 这个类的对象提供了一些服务
  - 定义了一个计算人员年龄的操作，利用标准库包 **datetime**

基本个人信息类

```
import datetime  # 用于生日和年龄计算

class Person:
    def __init__(self, name, sex, birthday):
        if not (isinstance(name, str) and
                sex in ("女", "男") and
                isinstance(birthday, tuple) and
                len(birthday) == 3):
            raise ValueError(name, sex, birthday)
        birth = datetime.date(*birthday) # 拆分，生成日期对象
        self._name = name
        self._sex = sex
        self._birthday = birth

    def set_name(self, name): # 修改名字
        if not isinstance(name, str): raise ValueError
        self._name = name
```

基本个人信息类

```
def name(self): return self._name
def sex(self): return self._sex
def birthday(self): return self._birthday
# age 计算人的年龄
def age(self): return (datetime.date.today().year -
                       self._birthday.year)

def __lt__(self, another): # 定义比较操作，按姓名排序
    if not isinstance(another, Person):
        raise TypeError
    return self._name < another._name

def __str__(self):  # join 的参数是元组
    return " ".join((self._name, self._sex, str(self._birthday)))

def print(self):
    print(self._name, self._sex, self._birthday)
```

基本个人信息类

- 这个类定义的一些情况：
  - 构造函数
  - 几个解析函数，它们只是访问实例对象的数据

    几个方法直接取数据属性的值

    age 利用 **datetime** 的功能计算年龄

    __lt__ 被小于运算符调用，系统的内置排序函数和 **list** 的 **sort** 方法里都需要使用这个比较

    __str__ 中用 **str** 的 **join** 方法构造结果，参数是元组
  - 一个变动函数 **set_name**。只有这个函数修改对象状态
  - 类 **Person** 没有明确描述的基类，以 **object** 为基类

继承

- 基本个人信息类可以用在任何需要记录人员管理的场合
- 现在要进一步考虑表示大学师生员工
  - 大学中每个师生员工有一个学号或职工号
  - 其相关信息也包括个人的姓名、性别、出生年月，其操作也包括对基本个人信息记录的各种操作
- 如何定义一个类表示大学的师生员工？
  - 完全可以从空白出发定义相关的类
  - 但 **Python** 的类继承机制，使我们可以利用已有的类扩充，将表示大学人员的类定义为 **Person** 的派生类
  - 通过派生类技术，可以尽可能利用已有的定义，还可以改变对象的行为。这样有可能大大减少开发工作量

大学师生员工类

- 我们考虑在大学师生员工类里增加两项功能
  - 每个大学人员（类的实例）有一个学号/职工号
  - 希望增加一种人员记录计数功能，可以自动记录在这个系统开始之后定义过多少人员
- 先考虑第二项功能：
  - 显然这个记录不能放在实例对象里
  - 这个记录与本类有关，也不应作为全局变量
- **Python** 类定义中的数据属性可用于表示与整个类有关的信息
  - 可以通过类名访问（和修改）
  - 这里考虑用一个数据属性 **_num** 记录学校的人数。由于是需要记录创建的实例对象，应该在初始化函数里自动计数

大学一般人员类

- 要让派生类的实例具有基类实例的属性，需要建立这些属性

  可以在派生类的初始化函数里调用基类的初始化函数，这样既能避免重写代码，也能保证构造正确（满足基类要求）

  这是 **Python** 提倡的做法，也是人们常用而且最合理的做法

- 新类定义：

```
class UniPerson(Person):
    _num = 0

    def __init__(self, name, sex, birthday, ident):
        Person.__init__(self, name, sex, birthday)
        self._id = ident
        UniPerson._num += 1  # 创建实例时自动加一

    def id(self): return self._id
```

大学师生员工类

```
def __lt__(self, another):
    if not isinstance(another, UniPerson):
        raise TypeError
    return self._id < another._id

@classmethod
def num(cls): return UniPerson._num
```

- 有关说明：
  - 这里定义了一个类方法，通过它查询类中信息（检查学校里的人数）。类方法加 **@classmethod** 前缀声明，这种方法要求一个表示类的参数（常用**cls**），用类名加圆点形式调用
  - 重新定义了 **__lt__**，本类实例基于学号/职工号排序
  - 还可以重新定义 **__str__** 函数，改变输出形式。在重新定义的函数里，可以用类名加圆点的形式调用基类方法

一般学生类

- 在一般师生员工类里的学号/职工号是通过参数直接给定的。实际中应该按一定规则自动生成，学号和职工号的生成规则不同，所以应该在不同的地方分别考虑
- 生成学生编号的工作是一个类的“公事”，有关信息必须作为类的属性记录，如果有复杂生成规则，也需要定义为类函数
- 下面是学生类的定义

```
class Student(UniPerson):

    _id_num = 0

    @classmethod
    def _id_gen(cls):  # 实现学号生成规则
        Student._id_num += 1
        return "1{:09}".format(Student._id_num)
```

一般学生类

```
def __init__(self, name, sex, birthday, department):
    UniPerson.__init__(self, name, sex, birthday,
                       Student._id_gen())
    self._department = department
    self._enroll_date = datetime.date.today()  # 入校日期
    self._courses = {} # 一个空字典，表示所修课程

def set_course(self, course_name):
    self._courses[course_name] = "No-score"

def set_score(self, course_name, score):
    self._courses[course_name] = score

# 其他方法定义从略
```

- 学生学号是自动生成的，可以根据需要定义生成规则
- 下面考虑学生类的三个派生类，分别表示本科生/硕士生/博士生

具体学生类

```
class UGStudent(Student):
    def __init__(self, name, sex, birthday,
                       department, sclass):
        Student.__init__(self, name, sex, birthday,
                               department)
        self._class = sclass

class MasterStudent(Student):
    def __init__(self, name, sex, birthday, department):
        Student.__init__(self, name, sex, birthday,
                               department)
        self._id = self._id[:3] + "5" + self._id[4:] # 修改学号
        self._field = "Undetermined"      # 研究领域未定
        self._supervisor = "Undetermined" # 导师未定
```

学号第3位用5表示硕士生

具体学生类

```
class DoctorStudent(Student):
    def __init__(self, name, sex, birthday, department,
                 field, supervisor):
        Student.__init__(self, name, sex, birthday,
                         department)
        self._id = self._id[:3] + "8" + self._id[4:]
        self._field = field
        self._supervisor = supervisor
```

学号第3位用8表示博士生

- 无论生成哪个类的对象，继承的方法都可以使用，如果需要，可以在任意层次重新定义已有方法

  例如，UniPerson 类里定义的对象计数功能始终有效，每生成一个 UniPerson 的派生类的对象，计数值都会加一

教职员工类

- 下面考虑定义教职员工类

  由于教职员工也有很多不同的人员类别，先定义一个一般类，其中反映所有教职员工的共性，再由它派生出具体的类

- 教职员工的共性特征：
  - 入职日期
  - 职别
  - 工资
- 下面定义一般职工类

教职员工类

```
class Staff(UniPerson):

    _id_num = 0

    @classmethod
    def _id_gen(cls, birthday):  # 实现职工号生成规则
        Staff._id_num += 1
        brith_year = datetime.date(*birthday).year
        return "0{:04}{:05}".format(birth_year, Staff._id_num)

    def __init__(self, name, sex, birthday, entrydate=None):
        UniPerson.__init__(self, name, sex, birthday,
                                 Staff._id_gen(birthday))
        if entrydate:
            self._entrydate = datetime.date(*entrydate)
        else: self._entrydate = datetime.date.today()
        self._salary = 1720    # 默认设为最低工资, 可修改
        self._department = None  # 由具体类设定
        self._position = None    # 由具体类设定
```

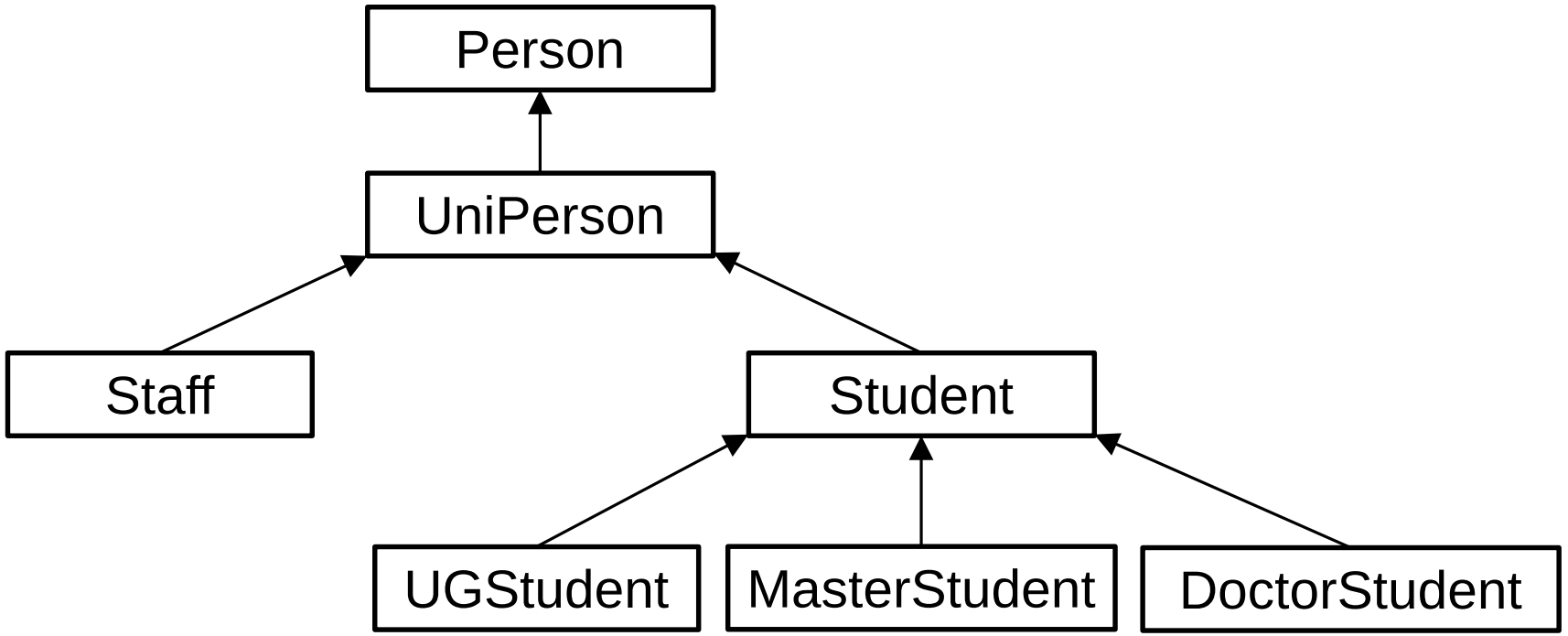
教职员工类

```
    def set_salary(self, amount):
        if not type(amount) is int:
            raise TypeError
        self._salary = amount

    def print(self):
        print(self.id(), self._name, self._sex,
                str(self.age()) + "-years")

    # 其他方法从略
```

- 下面可以考虑从 **Staff** 出发，派生出一批具体的职工类：如教师类，管理人员，辅助人员等，不再继续
- 至此，已经介绍了通过派生定义类时可能遇到的许多问题
  - 主要是方法的覆盖（重新定义），实例对象属性操作等
  - 顺便介绍了类的数据属性和类方法，及其使用

实例总结

- 前面工作定义的一组类形成了下面的分层次结构，其中箭头表示类之间的继承关系（从派生类到基类）



可以这样工作下去，通过适当的设计和改造，做出一批管理学校人事信息的类，基于它们开发出一个信息管理系统

类继承与子类型

- 现在介绍与类继承有关的一些情况
  - 如果类 D 是 C 的直接或间接派生类，也就是说，C 是 D 的直接或间接基类，issubclass(D, C) 都为真

    例如，issubclass(GUStudent, Person) 得到 True
  - 若 x 的值是类 D 的实例对象，则 type(x) is D 为真（虽然 type(x) == D 真但前一写法好），isinstance(x, D) 真，且
    - 对于 D 的任何基类 C，都有 isinstance(x, C) 为真，但显然 type(x) is C 不真
    - 也就是说，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
- 可知，isinstance(x, Person) 对 Person 及其所有子类的对象成立，isinstance(x, Student) 也一样，包括将来定义的派生类

方法查找

- 假设 x 的值是类 C 的一个对象，对于调用 x.m(...)，需要先找到 m 的定义，而后执行相应的函数，这个过程称为方法查找
- 方法都是在类中定义的，因此方法查找就是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：
  - 首先到该对象（称为调用对象）所属的类（C）中查找有没有名字为 m 的函数属性，如果有就确定了要调用的方法
  - 如没找到就转到 C 的直接基类，重复同样查找过程
  - 这一过程持续到在 C 的某基类找到 m 的定义；或者到达类 object 仍未找到 m 就是属性无定义，报告 AttributeError
- 查找是顺序进行的过程，一旦找到就结束。因此

  从 x 出发，实际调用的一定是 C 里定义的方法 m，或者其 C 的最近基类里定义的方法 m

方法查找

- 如果在派生类里重新定义了某个方法，例如 m，基类的同名定义就不会被正常查找过程找到

  所以前面定义子类的 __init__ 方法时，其中需要调用基类初始化函数时都写成 nnn.__init__(...)

- 看一个例子：

```
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？
```

基于对象查找所需方法，称为动态约束。

按方法查找规则结果必然：在 f 里调用 g 时应该根据 self 的对象的类型查找

方法查找

- 在 **Python** 里可以调整上述查找过程

  用内置函数 **super()** 可以要求从当前类的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

- 例如，前面 **Staff** 的 **__init__** 可以改写为

```
def __init__(self, name, sex, birthday, salary=1720):
    super().__init__(name, sex, birthday,  # 不写 self
                     Staff._id_gen(birthday))
    self._entrydate = datetime.date.today()
    self._salary = salary    # 默认设定为最低工资
    self._department = None  # 由具体类设定
    self._position = None    # 由具体类设定
```

- 采用 **super()** 的优点是不在派生类里直接写基类的名字，可以减少类之间的相互关联，可能更有利于程序的修改

替换原理

- 人们在面向对象编程中提出了一个“替换原理”，也就是说，要求派生类的对象能够用在要求其基类的对象的环境中
  - 具有同样的合理行为
  - 这实际上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，这称为是“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，通常用调用其初始化方法的方式完成
  - 覆盖（重新定义）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。同样可以通过调用基类原有方法的方式完成，例如通过 **super().m(...)** 形式
- 显然，不能要求基类对象能用到要求派生类对象的环境中

类和异常处理

- 实际上，Python的异常完全是基于类和对象的功能构造的
  - 每种异常是一个类，异常名就是类名
  - 运行中产生的异常（无论是系统引发还是通过 raise 语句引发），就是构造相应异常类的一个实例对象
  - 用 except *Ex_name* 描述捕捉的异常，就是要捕捉在相应 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，也就是说，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
  - 子句头部 except *Ex_name* as e 引入局部变量 e，如果运行进入这个异常处理器（也就是说，相应 try 块运行中发生异常，且该异常被这个处理器捕捉），e 的值就是由所引发异常对象带来的信息元组，成员是引发异常时的参数

类和异常处理

- 异常携带的信息用 **raise** 语句引发异常时的参数描述
- **Python** 语言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

```
+-- 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

- 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，从它派生出自己的异常类。例如：

```
class RationalZeroDivision(ZeroDivisionError):
    pass
```

- 最简单情况（很常见）就是希望定义一种特殊异常，以便区别处理，并不需要这种异常有什么特殊功能（如上即是）
  - 在这种情况下，选一个系统异常类派生自己的异常类，类体并不需要定义任何属性
  - 但需要语法完整，因此通常写一个**pass**语句
- 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

类和异常处理

- 由于异常是类，并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
- 下面是一个合理的例子：

```
try:
    … …
except RationalZeroDivision: # 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
    … …
```