21. 面向对象编程-3

- 多重继承

  方法解析序（mro）
- 类的特殊方法名

  容器类和相关特殊方法名
- with 语句
- 数据持久性

  标准库包 pickle

多重继承

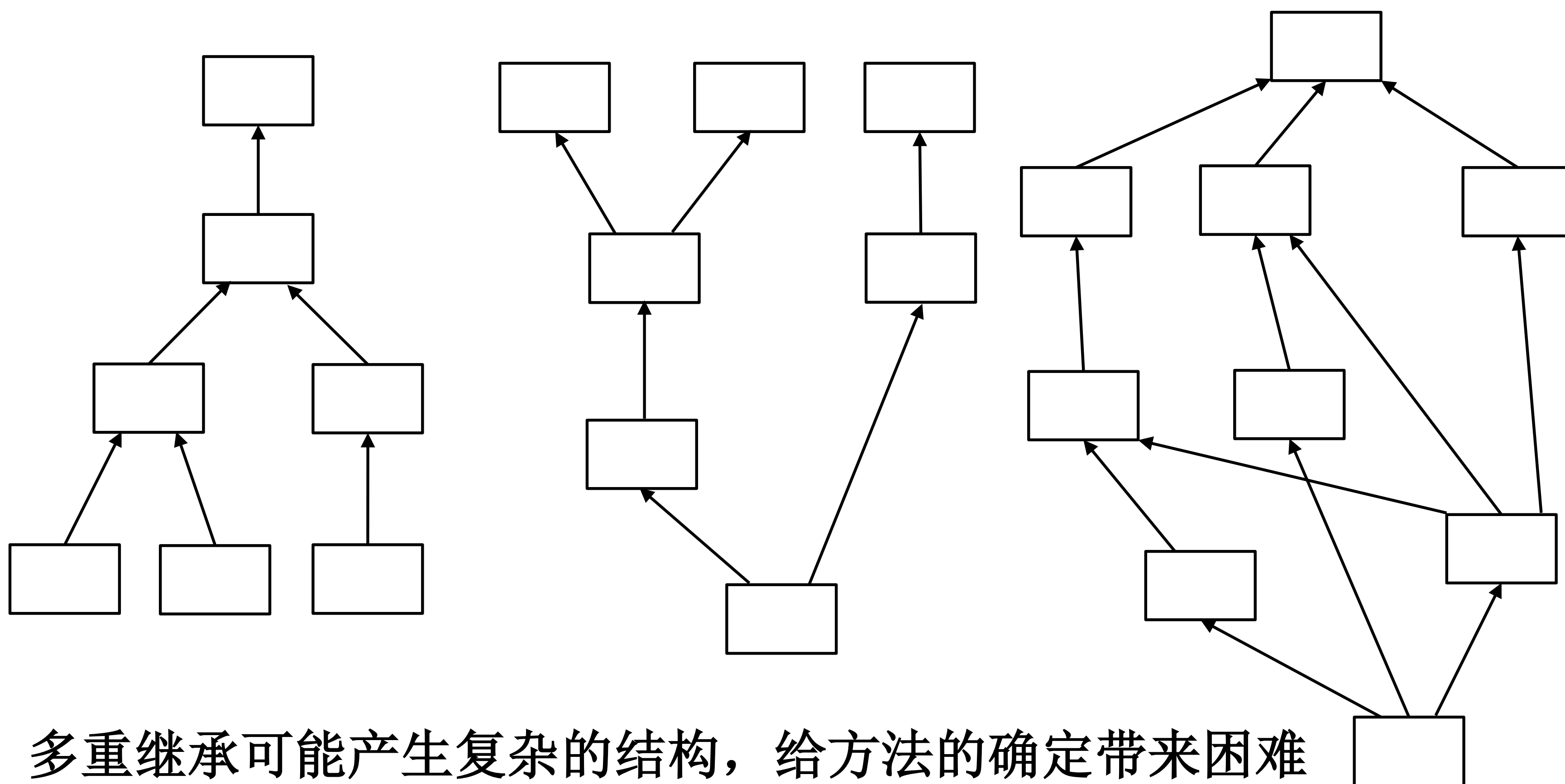
- 如果一个类定义继承了不止一个基类，就称为多重继承
  - **Python** 允许在定义类时指定任意多个基类
  - 多重继承时的派生类（及其对象）将继承所有基类的方法

    包括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
- 多重继承用于组合起不同功能，构造集成了多种功能的对象
- 例如，学校在在职硕士研究生
  - 这个人本身是学校的职工，又需要记录硕士研究生的相关信息（如课程完成情况，导师等），支持相关操作
  - 可以考虑从两个类（**Staff** 类和 **MasterStudent** 类）派生

多重继承

- 多重继承带来很多新问题，最重要的是方法查找问题

  如果多个基类里有同样名字的方法，怎样确定执行哪一个？



多重继承可能产生复杂的结构，给方法的确定带来困难

多重继承

- 方法查找（方法解析，method resolution）问题需要一个算法
  - 在一个任意复杂的类继承结构中确定一种顺序
  - 与简单情况（只有单继承的情况，只有简单多继承的情况）中的顺序相容，且易于理解
  - 是一个抽象的数学问题
- Python 把从一个类 C 出发检索方法的过程定义为一种序，用一个 list 表示，其中包含了 C 类对象方法检索中可能涉及的所有类
  - 默认定义的类方法（在 object 定义）mro() 给出这个表
  - mro 表示 method resolution order
- 基于类、继承、多重继承等数据组合技术，人们开发了许多组织和定义类的模式，称为面向对象的程序构造（开发）技术。有关情况有丰富的内容，是软件开发经验的总结。本课程不再深入

类的特殊方法名

- 前面介绍过的 __init__、__add__ 等特殊方法名
  - 实际上，Python 有一批这样的特殊名字，服务于各种目的
    - 一些方法名在特殊的场合调用，如 __init__
    - 一些用于模拟运算符，如 __add__
    - 一些被标准函数调用，如 __str__
  - 语言手册 3.3 节列出所有特殊方法名及其作用。如果在类里出现了特殊方法名的定义，在一定情况下就会调用它
  - 另一方面，具有这些特殊名字的方法也可以用常规形式调用
- 下面介绍几个（几组）特殊方法名，大致两类：
  - 模拟 Python 语言的某种表达形式，方便对象的使用
  - 服务于某种特殊用途

容器和元素访问

- Python 中一些内置类型的对象包含一些元素，如 list/dict 等，可以取元素或给元素赋值（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，能采用上述形式操作元素，也是很方便的描述形式
- 特殊方法名 __getitem__ 和 __setitem__ 服务于这种用途
  - 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，其对象就能做下标访问

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
  - 如果一个类定义了名字为 __len__ 的实例方法，如果对其对象调用 len 函数，系统就会自动调用上述方法
- 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

  如果一个类里定义了名字为 __contains__ 的实例方法，对该类的对象使用 in 或 not in 运算符时就会作用该方法
- 容器类还可以定义一个 __iter__ 方法，其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，也可以专门另外定义一个迭代器类（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）

一个简单容器类

- 考虑一个简单的容器类，其中用一个 list 保存元素，定义一组具有特殊名字的操作，可以像表一样使用
- 类定义梗概：

```
class Box:
    def __init__(self, elems=None): ...
    def __len__(self): ...
    def __getitem__(self, ind): ...
    def append(self, value): ...
    def __setitem__(self, ind, value): ...
    def __contains__(self, item): ...
    def __str__(self): ...
    def __iter__(self): ...
```

- 这只是一个简单示例，下面看一个更实际的例子

学生登记表类

- 现在考虑定义一个本科生登记表类

  计划给它定义一批有用操作，有些采用特殊的名字

  先考虑类的基本设计
- 需要记录一批学生的信息，采用什么结构？显然对象应能变动
  - 可以用 **list**，其中元素是学生信息记录
  - 可以用 **dict**，设某种检索字典的关键码，对应值是学生记录
  - 下面采用 **dict**，用学号作为关键码。用 **list** 的实现自己考虑
- 设计相应操作
  - 一批是通用的操作，如前面简单类的那些操作
  - 另一批是针对学生记录表的专门操作

学生登记表类

- 考虑实现下面常用方法（大多采用特殊方法名）：
  - add(self, student) 加入一学生记录
  - __len__(self) 给出登记表中的学生人数
  - __getitem__(self, sid) 返回学号为 sid 的学生记录，如果学号不存在就引发一个异常，不实现 __setitem__
  - __contains__(self, stu_id) 检查登记表有没有学号 sid 的学生，返回一个逻辑值
  - __iter__(self) 返回一个迭代器，该迭代器将按学号 sid 的顺序给出登记表里的学生记录
  - __init__(self) 建立一个空的登记表
  - __str__(self) 针对这个登记表做出一个“概要”字符串，对登记表里的每个学生，字符串里只有其学号和姓名

学生登记表类

- 再考虑一些处理学生登记表的专门操作
  - **display(self, sid)** 显示学号为 **sid** 的学生的所有信息，包括需要计算的平均成绩
  - **record(self, course, scores)** 登记一门课程的成绩。其中参数 **course** 是课程名串，**scores** 是 **(**学号**,** 成绩**)** 的表

    注意，学生的成绩也是用一个字典实现，以课程名为关键码，值是课程成绩
  - 一些与整个登记表有关的统计功能，可以适当考虑
- 基于学生类的已有功能，上述操作都不难实现
- 看看程序实现

with 语句

- 在进一步考虑学生登记表的设计和实现之前，先介绍 Python 的最后一种语句：with 语句

  with 语句是为了解决一类典型处理过程而引进的结构

  下面首先介绍这类问题，文件处理是一类典型的例子

- 假设现在要打开一个文件，读入文件内容，最简单的想法是：

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

- 但在文件读入和处理的过程中很可能出错
  - 因此需要考虑用 try 语句，把文件内容处理部分包装起来
  - 但文件打开和关闭应该配对，不用的文件应该关闭，尤其是在长期运行的程序中

with 语句

- 人们经常采用下面处理模式：

```
ifile = open("datafile.txt")
try:
    … … # 读入文件和处理
except …:
    … …
finally: # 写在 final 子句里的 close() 总会执行
    ifile.close()
```

- 在实际中有很多这类问题，其处理需要做一段计算，用一种操作进入这种处理片段，用另一个操作结束。其中的典型情况
  - 无论处理段执行中出现什么情况，能正常或者异常结束，都需要执行相应的结束处理操作
  - 这种情况可以用如上形式的 **try-except-final** 模式描述

with 语句

- Python 为这种有开始/结束操作的操作段引进了 with 语句

  前面文件处理可以简单写成

  ```
  with open("datafile.txt") as ifile
     … … # 通过 ifile 读入文件内容并处理
  ```

  文件对象的结束操作就是 close()

- 控制 with 语句执行过程的对象被称为 context manager，这种对象必须有明确定义的开始和结束方法
  - __enter__(self) 描述进入 with 块的动作，其返回值将约束于 as 关键字之后的给定变量名字
  - __exit__(self, type, value, trace) 描述退出 with 块时的动作。简单情况中后几个参数都不使用
  - 系统和标准库中的一些类型定义了这一对操作，包括文件对象类型等，可以直接用于 with 语句

with 语句

- 下面是一个简单的 with 语句的 context 管理器类：

```
class Context:
    def __enter__(self):
        print("__enter__() executed")
        return self

    def __exit__(self, type, value, trace):
        print("__exit__() executed")

    def action(self): # 根据需要定义
        print("action() executed")
        return "action ends"
```

- 调用代码段：

```
with Context() as con:
    print("Context:", con.action())
```

with 语句

- **with 语句的形式是：**

  with 表达式 as 变量, …:

  语句块

  可以有多个 expr as var 片段。出现几个 as 片段相当于几个嵌套的 with 语句，后面的先退出。如下面两段等价

```
with A() as a, B() as b:
    … …

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

- 下面考虑一个在实际应用中非常重要的场景

  其中使用 with 语句来描述非常方便

数据持久性

- 前面说过，计算机内存的数据无法持久保存
  - 断电即逝，再也找不到了
  - 程序里建立和使用的数据，即使用全局变量保存，程序结束也是这些数据的生命期结束（即使计算机还在运行）
  - 如果需要长期保存数据，就需要把它们存入外存介质，典型情况是存入外存的命名文件里
- 考虑前面的学生登记表，遇到了一些新麻烦：
  - 显然，这种登记表是在程序运行中逐步建立起来的。没人愿意每次启动程序总从空表开始重新建立新登记表，而希望继续使用原来已经建立的登记表。这说明需要保存到外存
  - 但现在的记录表不是简单数据，具有复杂结构。希望保存之后还能重新装入，恢复原样继续使用（并可以扩充和修改）

数据持久性

- 我们可以自己写两个操作函数，一个能把指定 **Register** 对象中的所有信息保存到文件，另一个能从文件读入这样一套信息，重新建立起和原对象等价的 **Register** 对象

  具体工作牵涉到文件中存储的数据及其形式问题，以及两个操作函数的实现。这件事比较麻烦，但一定能做到（自己考虑）

- 许多程序都需要持久性地保存构造出的复杂数据对象，而后需要把数据读进来，重新恢复原来的对象
  - 这是一种规范性的工作，语言系统需要支持
  - **Python** 标准库提供了几种标准的数据持久性库，见标准库手册第 **12** 章，**Data Persistence**
  - 各个库功能有些差异，可以自己阅读和比较
  - 下面只介绍其中使用最方便的一个库（但不是功能最强的），它已经足够应付我们面临的问题

数据持久性

- 下面要用的库名字是 **pickle**，其意有腌咸菜装坛等，使用它
  - 能很方便地把系统类型或自定义类型的对象存入文件
  - 能方便地重新装入并恢复保存的对象
- 假设 **math** 的值是程序里建立的 **Register** 对象，下面语句将这个对象转存到文件 **data.pickle**：

```
with open('data.pickle', 'wb') as f: # 二进制写模式打开文件
    # Pickle 用默认协议把 Register 对象 math 卸载到文件
    pickle.dump(math, f)
```

下面语句恢复文件 **data.pickle** 中保存的对象

```
with open('data.pickle', 'rb') as f: # 二进制读模式打开文件
    # 把装入重建的对象赋给变量 data，自动识别存储协议
    data = pickle.load(f)
```

数据持久性

- **Pickle** 以二进制形式保存内存对象，所建立的文件的内容不能用普通编辑器检查或修改
- 显然，**Pickle** 的 **dump** 和 **load** 命令必须相互配合，才能正确保存和恢复原来在内存的数据（可能具有很复杂的结构）
- 在计算机领域，两方或者多方需要相互配合完成一件共同工作的事情也非常多。各种网络服务是这方面的典型
  - 各户端和服务端需要按一定的规矩相互配合
  - 传输数据需要按一定格式安排内容，例如有特定形式的头部，否则使用方就不能正确分析和处理
- 为保证合作的双方（或多方）能正确工作，必须做出一套严格的规定，说明数据表示的方式或合作过程的步骤，这类严格定义的规定称为“协议”（**protocol**）。为正确保存和恢复，**Pickle** 也规定了若干套保存和恢复协议

数据持久性

- 在调用 **dump** 操作时，可以指定特殊的协议。不指定时 **Pickle** 默认使用功能最强的协议。在使用 **load** 操作时，**Pickle** 能够自动识别被装入的文件使用的协议
- **Pickle** 可以保存的对象：
  - 逻辑和数值对象，字符串等
  - 元素是可 **pickle** 对象的 **tuple, list, set, dict** 对象
  - 在模块顶层定义的类及其实例对象
  - 在模块顶层定义的函数，等等

  更多详情请查看系统标准库手册
- 可以把多个对象逐一 **dump** 到一个文件里，然后逐一恢复为对象赋值给一个个变量（见例子）

Python 关键字分类

- 关键字有如下大致分类：
  - 特殊字面量 None, True, False
  - 运算符 and, or, not, in, is （is not, not in）
  - 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, with (as)
  - 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, with 等处）
  - 类定义 class
  - for-in 还用在各种描述式中，……