Modern Program Design

Class	
Created	@Dec 25, 2019 10:17 AM
Materials	
Reviewed	☑
Type	CS

还没有整理完善，蓝色的部分是要补充的内容，另外超了解（单击此处）等还没有加

一、基本语法整理

杂乱的整理一些新鲜的语法/注意事项（随想随补）

- 列表生成式
- 有步长的列表元素截取
- del删除元素
- 创建空集合必须用set

```
with #上下文管理器
print(hex(id(var))
zfill(width)
tup2 = (20,)#元组只有一个元素，要加逗号
repr(x) #返回对象 x 的字符串表达
eval(str) #用来计算在字符串中的有效Python表达 式,并返回一个对象
frozenset(s) #转换为不可变集合
chr(x) #将一个整数转换为一个字符

#随机数函数
choice(seq)#从序列中随机挑选一个元素
randrange ([start,] stop [,step])#从指定范围内，按指定基数递增的集合中获取一个随机数，基数默认值为 1
random()#随机生成下一个实数，在[0,1)范围内
seed([x])#改变随机数生成器的种子seed
shuffle(list)#将序列的所有元素随机排序
uniform(x, y)#随机生成下一个实数，在 [x,y]范围内
```

二、OOP

1.类的基本概念

（1）OO（objected oriented）

1.概念

- 一种程序设计范式
- 程序由对象组成，每个对象包含对用户公开的特定功能和隐藏的实现部分
- 对象是数据与相关行为的集合
- 不必关心对象的具体实现，只要能满足用户的需求即可

2.特点

- 将数据搁在第一位
- 更加适用于规模大的问题

3.三大特征

- 封装：将数据和行为组合，并对外隐藏数据的实现方式
- 继承：通过扩展一个类来建立另外一个类
- 多态：可以对不同类型的对象执行相同的操作

4.对象的特性

- 行为：通过可调用的方法定义
- 状态
  - 保存描述当前特征的信息
  - 对象状态的改变必须通过调用方法实现
- 标识：每个对象实例的标识应该是不同的

（2）类

1.概念

- 对象的类型，用来描述对象
- 构造对象的模板
- 定义了该集合中每个对象所共有的属性和方法
- 由类构造对象的过程称之为实例化，或创建类的实例

2.类与类之间的关系

- 依赖
  - 一个类的方法操纵另一个类的实例
- 聚合：类A包含类B的实例对象
- 继承
  - 类A由类B扩展而来，这时，类A不但包含从类B继承的方法，还有一些额外的功能

（3）代码详解

创建类

类数据属性和实例数据属性辨析

类的私有性

类方法与静态方法

特殊的类属性

类的常用专有方法

__new__

2.继承

（1）概念：通过扩展一个类来建立另外一个类

（2）代码详解：

继承

继承的检查

多继承

应用：运算符重载

3.多态

（1）概念：可以对不同类型的对象执行相同的操作

（2）代码示例

对内部类的继承contact.py

4.抽象类

（1）基本概念

- 抽象类
  - 特殊的类，只能被继承，不能被实例化
  - 从不同的类中抽取相同的属性和行为
- 元类：创建类的类
  - 使用metaclass创建
- 抽象类和普通类的区别
  - 抽象类中有抽象方法
  - 不能被实例化，只能被继承
  - 子类必须实现抽象方法

（2）代码实现

1.抽象类的实现（使用abc模块）

继承抽象类

注册抽象类

4.2接口Interface

> 💡 不重点要求掌握，这里只讲概念

（1）基本概念

- 抽象类是对根源的抽象，而接口是对动作的抽象
- 抽象类表示这个对象是什么，比如男人女人都是人，而接口则表示这个对象能做什么，比如人和长颈鹿都能吃东西
- 抽象类与接口使用中十大区别：
  - 抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象
  - 抽象类要被子类继承，接口要被类实现
  - 接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现
  - 接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量
  - 抽象类里的抽象方法必须全部被子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。同样，一个实现接口的时候，如不能全部实现接口方法，那么该类也只能为抽象类（想到了刘禹的课）
  - 抽象方法只能申明，不能实现，接口是设计的结果 ，抽象类是重构的结果
  - 抽象类里可以没有抽象方法
  - 如果一个类里有抽象方法，那么这个类只能是抽象类
  - 抽象方法要被实现，所以不能是静态的，也不能是私有的。
  - 接口可继承接口，并可多继承接口，但类只能单根继承

6.高级用法

（1）__slots__

（2）@property

三、多进程、多线程与协程

1.基本概念

（1）并行与并发

- 并行：在同一时刻，有多条指令在多个处理器上同时执行
- 并发：在同一时刻，只能有一条指令执行，但多个指令被快速轮换执行，使得在宏观上表现多个指令同时执行的效果

（2）进程和线程

- 进程（process）：操作系统分配资源的基本单位
- 线程（thread）：CPU调度和分派的基本单位
- 特点：
  - 应用程序至少有一个进程
  - 一个进程至少有一个线程
  - 同一进程的多个线程可以并发执行
  - 进程在执行过程中有独立的内存单元，而线程共享内存
  - 多进程编程需要考虑进程间的通信
  - 进程切换时，耗费资源较大
- 比较
  - 数据共享/同步
    - 多进程：数据共享复杂，需要IPC（Inter-Process Communication进程间通信）
    - 多线程：共享进程数据，简单，但同步复杂
  - 内存/CPU
    - 多进程：占用内存多，切换复杂，CPU利用率低
    - 多线程：占用内存少，切换简单，CPU利用率高
  - 创建销毁、切换
    - 多进程：复杂，速度慢
    - 多线程：简单，速度快
  - 编程、调试
    - 多进程：简单
    - 多线程：复杂
  - 可靠性
    - 多进程：进程间不会相互影响
    - 多线程：一个线程出错将导致整个线程终止
  - 分布式
    - 多进程：适用于多核、多机分布式
    - 多线程：适用于多核分布式
  - 适用条件
    - 多进程：计算密集型，如金融分析
    - 多线程：IO密集型，如socket，爬虫，web

（3）同步IO与异步IO

- 同步IO：在IO过程中当前线程被挂起，其它需要CPU计算的代码无法执行

——应用多线程可以解决该问题：计算和IO任务可以由不同线程负责

但会带来线程创建、切换的成本，而且线程不能无上限增加

- 异步IO：当前线程只发出IO指令，但不等待其执行结束，而是先执行其他代码，避免线程因IO操作而阻塞
- 事件驱动模型：一种编程范式，包含一个事件循环，当外部事件发生时使用回调机制来触发相应的处理
  - 实现机制
    - 每收到一个请求，创建一个新的进程来处理该请求
    - 每收到一个请求，创建一个新的线程来处理该请求
    - 每收到一个请求，放入一个事件列表让主进程通过非阻塞IO方式来处理请求
  - 应用场景：当程序中有许多任务，任务之间高度独立（不需要 互相通信或等待彼此等），并且在等待事件到来时， 某些任务会阻塞
- 事件列表模型
  - 实现机制
    - 主线程不断重复“读取请求-处理请求”这一过程
    - 进行IO操作时相关代码只发出IO请求，不等待IO结果，然后直接结束本轮事件处理，进入下一轮事件处理
    - 当IO操作完成后，将收到IO完成消息，在处理该消息时再获取IO操作结果
    - 在发出IO请求到收到IO完成消息期间，主线程并不阻塞，而是在循环中继续处理其他消息
  - 应用场景：对于大多数IO密集型的应用程序，使用异步IO 将大大提升系统的多任务处理能力
- 协程就是应用事件列表模型来进行异步IO的操作

2.多进程编程

（1）概念术语

1.进程号：通过`os.getpid()`得到

2.孤儿进程：子进程在父进程退出后仍在运行，会被init进程接 管，init以父进程的身份处理子进程运行完毕后遗 留的状态信息

3.僵尸进程：父进程创建子进程后，如果子进程退出，但父进程并没有获取子进程的状态信息， 那么子进程的进程描述符将一直保存于系统，对应的子进程称为僵尸进程

- 僵尸进程无法通过kill命令来清除

4.同步与异步

- 同步：
  - 按预定的顺序先后执行
  - 调用后需要等到返回结果
  - 同步是保证多进程安全访问竞争资源的一种手段
- 异步：不用阻塞当前进程来等待处理完成， 而是允许后续操作，并回调通知

5.临界区

- 临界资源：一次仅允许一个进（线）程使用的资源

- 临界区（Critical Section）：存取临界资源的代码片段
  - 多进程要求进入空闲的临界区时，一次仅允许 一个进（线）程进入
  - 如已有进（线）程进入临界区，则其它试图进 入临界区的进（线）程需要等待
  - 进入临界区的进（线）程要在有限时间内退出

6.互斥量：是一个仅处于两态之一的变量（解锁/加锁）

7.基于消息的IPC（inter-process communication）通信机制

- 队列
- 管道

8.进程池：进程开启过多导致效率下降（同步、切换带来成本），所以应固定工作进程的数目

- 由这些进程执行所有任务，而非开启更多的进程
- 与CPU的核数相关

9.分布式多进程（详见代码示例）

- 多机环境
- 跨设备数据交换

（2）函数详解

1.multiprocessing模块

- Process进程类
  - 创建进程类

```
Process([group [, target [, name [, args [, kwargs]]]]])
```

  - 实例对象表示一个子进程（尚未启动）
  - 使用关键字的方式来指定参数
    - `group`参数未使用，值始终为`None`
    - `target`表示调用对象，即子进程要执行的任务
    - `args`指定传给target函数的位置参数，元组形式，仅有一个 参数时要有逗号
    - `kwargs`表示调用对象的字典参数
    - `name`为子进程的名称
  - 在windows中`Process()`必须放到`if __name__ == '__main__'`中
  - Process类中方法简介
    - p.start()：启动进程，并调用该子进程中的p.run()
    - p.run()：进程启动时运行的方法，会调用target，自定义类一定要实现该方法
    - p.terminate()：强制终止进程p，可能产生僵尸进程，一般不建议使用

- p.is_alive()：如果p仍然运行，返回True
- p.join([timeout])
  - 主线程等待p终止（主线程处于等状态，而p处于运 行状态）
  - timeout是可选的超时时间
  - 只能join住start开启的进程，而不能join住run 开启的进程
- p.daemon
  - 默认值为False
  - 可以设置为True，但必须在p.start()之前设置，成为后台运行的守护进程，当p的父进程终止时，p也随之终止，且p不能创建新的子进程
- p.name：进程的名称
- p.pid：进程的pid
- p.exitcode：进程在运行时为None，如果为–N，表示被信号N结束
- p.authkey：进程的身份验证键

- Queue队列
  - `Queue([maxsize])`
  - `q.put()`
    - 插入数据到队列，参数blocked默认为True
    - 可能抛出Queue.Full异常
  - `q.get()`
    - 从队列读取并删除一个元素
    - 可能抛出Queue.Empty异常
- Pipe管道
  - `Pipe([duplex])`
  - 在进程之间创建一条管道，并返回元组`(conn1,conn2)`,其中`conn1, conn2`表示管道两端的连接对象
  - 必须在产生Process对象之前产生管道
  - dumplex：默认全双工，如果将duplex置为False，conn1只能用于接收，conn2只能用于发送
  - `conn1.recv()`:接收`conn2.send(obj)`发送的对象，如果没有消 息可接收，recv方法会一直阻塞;如果连接的另外一端已经关闭，那么recv方法会抛 出EOFError
  - `conn2.send(obj)`:通过连接发送对象，obj是与序列化兼容的任意对 象
  - `conn1.close()`:关闭连接

2.ProcessPoolExecutor

- 一种更简单、统一的接口(12-8)

（3）编程示例

3.多线程编程

（1）概念术语

1.GTL(Global Interpreter Lock)

- GIL本质上是互斥锁，控制一时间共享数据只能被一个任务修改，以保证数据安全
- GIL在解释器级保护共享数据，在用户编程层面保护数据则需要自行加锁处理
- Cpython解释器中，同一个进程下开启的多线程，同一时刻只能有一个线程执行，无法利用 多核优势
    - 可能需要先获取GIL

（2）函数详解

1.Threading

2.ThreadPoolExecutor

（3）编程示例

4.协程-14

（1）基本概念

1.英文：Coroutine, peusdo-thread, micro-thread（微线程）

2.执行过程

- 在一个线程中会有很多函数，一般将这些函数称为子程序，在子程序执行过程中可以中断去执行别的子程序，而别的子程序也可以中断回来继续执行之前的子程序，这个过程就称为协程
    - 执行函数A时，可以随时中断，进而执行函数B，然 后中断B并继续执行A，且上述切换是自主可控的
    - 但上述过程并非函数调用（没有调用语句）
- 表象上类似多线程，但协程本质上只有一个线 程在运行

（2）优劣分析

1.优点

- 无需线程上下文切换的开销，协程避免了无意 义的调度，由此可以提高性能
- 无需原子操作锁定及同步的开销
- 方便切换控制流，简化编程模型
    - 线程由操作系统调度，而协程则是在程序级别由程 序员自己调度
- 高并发+高扩展性+低成本
    - 一个CPU可以支持上万协程
    - 在高并发场景下的差异会更突出

2.缺点

- 程序员必须自己承担调度的责任
- 协程仅能提高IO密集型程序的效率，但对于CPU密集型程序无能为力

- 相关生态不成熟

3.在CPU密集型程序中要充分发挥CPU利用率需 要结合多进程和协程

（3）函数详解

1.通过yield关键字实现

2.通过greenlet实现

3.通过gevent实现

4.通过asyncio实现

（4）编程示例

四、异常处理-7

1.基本概念

（1）处理内容

- 向用户通告可能的错误
- 保存所有的已有结果
- 允许用户以妥善的形式退出程序

（2）常见异常

- 用户输入错误
  - 格式、语法
- 设备错误
  - 硬件故障、临时性下线
- 物理限制
  - 空间不足
- 代码错误
  - 对象为空、无效数据、除0

2.异常处理过程

（1）python内置异常

用的时候查就行了，懒得整理

（2）异常捕获

先给一个一般例子

```
try:
    print('try...')
    r = 10 / 0
    print('result:', r)
except ZeroDivisionError as e:
```

```
    print('except:', e)
finally:
    print('finally...')
print('END')
```

输出

```
try...
except: division by zero
finally...
END
```

- `try`执行try子句
  - 若无异常发生则except子句被忽略
  - 若发生异常则该子句其余部分被忽略
  - 如异常匹配except指定的异常类型，则执行对应的except子句
  - 如发生异常在except子句中没有与之匹配的分支，则传递到上一级 try 语句
  - 如终仍找不到对应的处理语句，则作为未处理异常，终止程序运行并显示提示信息
  - 包含多个except子句时多只有一个分支会被 执行
- `except ZeroDivisionError as e:`
  - 用这种方法将异常付给一个实例对象
- `finally`
  - 定义清理行为：无论在任何情况下都会执行
  - 在finally子句中应只做打印错误信息或 者关闭资源等操作
  - 避免在finally语句块中再次抛出异常

（3）处理捕获

- 异常处理并不仅仅处理那些直接发生在try子句中的异常，还能处理子句中调用的函数（甚至 间接调用的函数）里抛出的异常
- 但是不会处理其它处理程序中的异常（发生在except内部的异常（来自cxh））

（4）异常抛出

使用raise进行异常抛出

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input error!')
```

输出

```
Traceback (most recent call last):
  File "C:/Users/chunhui/Desktop/大三上/Modern Program Design/Code/Core_Practice.py", line 4, in <module>
```

```
    raise ValueError('input error!')
ValueError: input error!
```

- raise语句允许程序员强制抛出一个指定的异常
- raise语句如果不带参数，就会把当前错误原样抛出。此外，在except中raise一个Error，还可以把一种类型的错误转化成另一种类型

3.用户自定义异常类

（1）基本方法

- 直接或间接的从Exception类派生
- 异常类的命名多以 “Error”结尾
- 通常为了保持简单，异常类只包含属性信息， 以供异常处理
- 如果一个新建的模块中需要抛出几种不同的错 误，通常的作法是为该模块定义一个异常基类， 然后针对不同的错误类型派生出对应的异常子 类

五、高级特性

1.装饰器（Decorator）

（1）基本概念：在不修改原始代码的前提一下增强或扩展既有功能

- 在核心功能的基础上增加额外的功能如授权，日志等

简单例子：

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
@log
def now():
    print('2015-3-25')
now()
```

输出

```
call now():
2015-3-25
```

这里是把`@log`放到`now()`函数的定义处，相当于执行了语句：

```
now = log(now)
```

由于`log()`是一个decorator，返回一个函数，所以，原来的`now()`函数仍然存在，只是现在同名的`now`变量指向了新的函数，于是调用`now()`将执行新函数，即在`log()`函数中返回的`wrapper()`函数。

`wrapper()`函数的参数定义是`(*args, **kw)`，因此，`wrapper()`函数可以接受任意参数的调用。在`wrapper()`函数内，首先打印日志，再紧接着调用原始函数。

（2）应用

（其实在类中也有很多应用，比如@property，@profile等，这里就只介绍一个）

泛函数（9-14）（后面再补充）

2.生成器（generator）

（1）基本概念：一种一边循环一边计算元素的机制

（2）相对优势

- 直接生成列表：可能受到内存大小的限制，或者导致较高但不必要的时间成本
- 生成器
  - 惰性求值
  - 循环过程中不断返回后续元素
  - 避免一次创建完整的数据结构，从而节省大量的空间

（3）代码构建（10-7）

3.迭代器（Iterator）

（1）基本概念

1.迭代（Iterable）：通过for ... in等遍历数据结构如tuple, list， dict，字符串等

2.迭代器：可以被next()函数调用并不断返回下一个值的对象称为迭代器

- 迭代器可以记住遍历位置
- 迭代器只能往前不能后退

（2）特点

- 生成器都是迭代器，但list、dict、str 虽然可迭代，却不是迭代器（可以用iter()函数把其变成一个迭代器）
- 迭代器对象可以表示一个数据流
  - 被next()函数调用并不断返回下一个数据，直到没 有数据时抛出StopIteration错误
  - 可将该数据流看做是长度未知的有序序列，只能不 断通过next()函数实现按需计算下一个数据（惰性计算）
  - 可表示无限大数据流，例如全体自然数，而使用 list等不可能存储全体自然数

（3）代码示例

创建迭代器

迭代器相关工具

六、设计模式

0.设计原则

（1）开闭原则

- 对扩展开放，对修改关闭

（2）里氏代换原则

- 基类可被派生类替换

（3）依赖倒转原则

- 针对接口编程，依赖抽象而不依赖具体

（4）接口隔离原则

- 使用多个隔离的接口，比使用单个接口要好
- 降低类之间的耦合度

（5）最小知道原则

- 一个实体应当尽量少地与其他实体发生作用
- 系统功能模块应相对独立

（6）合成复用原则

- 尽量使用合成/聚合的方式，而不是使用继承

1.单例

（1）思路

- 确保某一个类只有一个实例存在

（2）代码示例

- 创建

```
class Singleton:

  def __init__(self):
    pass

  def __new__(cls,*args,**kwargs):
    if not hasattr(Singleton,'_instance'):
      Singleton._instance=object.__new__(cls)
    return Singleton._instance

slist=[Singleton() for i in range(10)]
for s in slist:
  print(hex(id(s)))
```

其中，__new__在__init__之前执行，其详细用法见OOP一节，hasattr判断类中有无所提及的属性，如果没有，则为其创建一个实例，如果已经存在，则直接返回

- 输出

```
0x227bb56ff28
0x227bb56ff28
0x227bb56ff28
0x227bb56ff28
0x227bb56ff28
0x227bb56ff28
0x227bb56ff28
0x227bb56ff28
0x227bb56ff28
0x227bb56ff28
```

可见循环创建出的实例地址都相同，其实质上只有一个实例

（3）应用场景

- 数据库连接读取配置文件，如果在程序运行期间，有很多地方都需要连接数据库，很多地方都需要创建数据库对象的实例，这就导致系统中存在多个 数据库实例对象，而这样会严重浪费内存资源，事实上，我们希望在程序运行期间只存在一个实例对象。

2.工厂模式

（1）思路

- 创建对象时不会对客户端暴露创建逻辑

（2）代码示例

```
class Fruit:
    pass

class Apple(Fruit):
    pass
    def pie(self):
        print("making apple pie")

class Orange(Fruit):
    pass
    def juice(self):
        print("making orange juice")

class FruitFactory:
    def generate_fruit(self, type):
        if type == 'a':
            return Apple()
        elif type == 'o':
            return Orange()
        else:
            return None

ff = FruitFactory()
apple = ff.generate_fruit('a')
orange = ff.generate_fruit('o')
apple.pie()
orange.juice()
```

- 这里，工厂类中只有一个创建类的方法，为了为不同的水果创建实例，建立Fruit父类，其它水果继承该类，其它水果不需要用__init__表示他们是怎么被实例化的，直接写其它的方法，在工厂类中return其实例，创建类时，直接使用工厂创建

- 输出

```
making apple pie
making orange juice
```

（3）特点

- 优点：隐藏了创建类的代码逻辑
- 缺点：

  比如上面水果类，如果后面再加一个水果，梨，我们需要在工厂类中也再加一个条件判断，这违背了“开放-封闭”原则：对扩展开放，对修改关闭

3.代理模式

（1）思路

- 使用代理对象在访问实际对象前执行重要操作

（2）类型

- 远程代理：实际存在于不同地址空间的对象在本地的代理者
- 虚拟代理：用于惰性求值，将一个大计算量对象的创建延迟到真正需要的时候进行
- 保护代理：控制对敏感对象的访问
- 引用代理

（3）代码实例（保护代理）

```
class Record:
  def read(self):
    pass

  def add(self,user):
    pass
class RecordError(Exception):
  def __init__(self):
    self.message='Access Record Failed'

class AddUserNotAllowedRecordError(RecordError):
  def __init__(self,user):
    self.message="Add user of {} failed due to no permission!".format(user)

class ReadUsersNotAllowedRecordError(RecordError):
  def __init__(self):
    self.message="read users not allowed due to no permission!"
class KeyRecords(Record):
  def __init__(self):
    self.users=['admin']

  def read(self):
    return ' '.join(self.users)

  def add(self,user):
    self.users.append(user)

class ProxyRecords(Record):
  def __init__(self):
    self.key_records=KeyRecords()
    self.secrect='test'
```

```
  def read(self,pwd):
    if self.secrect==pwd:
      return self.key_records.read()
    else:
      raise ReadUsersNotAllowedRecordError()

  def add(self,pwd,user):
    if self.secrect == pwd:
      self.key_records.add(user)
    else:
      raise AddUserNotAllowedRecordError(user)
    return 1

def main():
  pr=ProxyRecords()
  pwd=input("plz input the pwd:")
  try:
    print(pr.read(pwd))
    if pr.add(pwd,'zjc'):
      print("ADD Succeeded...")
  except ReadUsersNotAllowedRecordError as runare:
    print(runare.message)
  except AddUserNotAllowedRecordError as aunare:
    print(aunare.message)
  except RecordError as re:
    print(re.message)

if __name__=='__main__':
  main()
```

- 这里，应用场景是客户端读取和添加数据，这个过程我们要进行访问控制，如果在Record记录类中添加这个访问控制，那么这个类既要实现读取read和添加add的主功能，又要进行访问控制（即密码正确通过，密码错误报错），过于复杂，违背了软件设计中的单一职责原则
- 因此通过代理模式来执行这个访问控制，代码中Record作为父类，KeyRecords行使Record的主要功能，ProxyRecords作为代理类，键入密码，由代理类判断进入KeyRecords还是进行错误处理。

4.观察者模式

（1）思路

- 一种行为型设计模式，一个对象的变化可以通知其它对象
  - 行为型设计模式：设计对象和类的结构
  - 结构型设计模式：关注对象间职责，处理对象之间交互

（2）应用场景

- 新闻发布机构和它的订阅者
- 股票机构
- 分布式系统事件服务

（3）代码示例

```
import time


class Investor:
    def __init__(self, name, stock):
        self._name = name
        self._stock = stock
```

```
    @property
    def stock(self):
        return self._stock

    @stock.setter
    def stock(self, value):
        self._stock = value

    def update(self):
        print("{} invest on {} with price {}: sell it now!!!".format
              (self._name, self._stock.name, self._stock.price))


class Stock:
    def __init__(self, name, price):
        self._name = name
        self._price = price
        self._investors = []

    @property
    def name(self):
        return self._name

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        if self._price > value:
            self.notify()
        self._price = value

    def attach(self, investor):
        self._investors.append(investor)

    def notify(self):
        for investor in self._investors:
            investor.update()


def main():
    s = Stock('ABCD', 11.11)
    i1 = Investor('zjc', s)
    i2 = Investor('lys', s)
    s.attach(i1)
    s.attach(i2)
    s.price = 13
    time.sleep(1)
    s.price = 10


if __name__ == '__main__': main()
```

- @property：内置装饰器，负责把一个方法变成属性调用，详细用法点击此处
- 股票价格的变动会触发投资者里面的update推送消息
- 这个例子：当股票价值value上涨时（用sleep来模拟上涨需要的时间，当股票价值上涨时，价格等于价值；价值下跌时，立刻通知investor卖出股票）
- 可以随时添加删除观察者

5.适配器模式

（1）思路

- 为了解决接口不兼容的问题引进一种接口的兼容机制

（2）应用场景

- 比如通常情况下，原系统的代码要么无法获取——如库等、要么难以冒险重构——如运行5年以上的老旧系统牵一发而动全身。在设计中使用适配器模式，可以保证在不修改原系统代码的前提下，实现新需求与原系统的对接。
- 适配器模式是一种**结构型设计模式**，实现两个不兼容接口之间的兼容。以保证程序符合**开放/封闭原则**，保持新老代码间的兼容性。
- **结构型设计模式**处理一个系统中不同实体（类和对象）之间的关系，关注的是提供一种简单的对象组合方法来创造新功能。

（3）代码示例

- 第一种写法

```
class Computer:
  def __init__(self,name):
    self._name=name

  @property
  def name(self):
    return self._name

  @name.setter
  def name(self,value):
    self._name=value

  def __str__(self):
    return 'the {} computer'.format(self._name)

  def execute(self):
    print('execute a computer program')

class Human:
  def __init__(self,name):
    self._name=name
  def __str__(self):
    return '{} is an human'.format(self._name)
  def speak(self):
    print('an human speaks')

class Synthesizer:
  def __init__(self,name):
    self._name=name
  def __str__(self):
    return 'the {} synthesizer'.format(self._name)
  def play(self):
    print('play a synthesizer')

class Adapter:
  def __init__(self,adp_obj,adp_methods):
    self.obj=adp_obj
    self.__dict__.update(adp_methods)

  def __str__(self):
    return str(self.obj)
```

- 有一个问题：客户端仅知道如何调用execute()方法，并不知道 Synthesizer.play() 和 Human.speak() 。必须像调用 Computer.execute() 一样使用 Synthesizer.execute() 和 Human.execute() 来调用原系统中对象的执行函数。
- 适配器是救星！我们可以创建一个 Adapter 类专门用于统一接口。**init**()方法的obj参数是我们想要适配的对象，adapted_methods是一个字典，键值对中的键是客户端要调用的方法，值是应该被调用的方法。

- 接下来，只需要在调用时，对原有系统的类进行封装，即可实现统一使用 execute() 方法执行动作了，代码如第一种写法：

```
def main():
  objects=[Computer('mac')]
  syn=Synthesizer('yamaha')
  objects.append(Adapter(syn,dict(execute=syn.play)))
  h=Human('zjc')
  objects.append(Adapter(h,dict(execute=h.speak)))
  for obj in objects:
    obj.execute()
```

- 通过Adapter类中self.__dict__.update(字典)的方法，相当于为该类增添了一个属性self.execute，其值为syn.play(另一个类的方法)，并通过更改__str__方法，隐藏了该类的名字
- 有关__dict__，__str__的详细用法单击此处
- 个人思考：为什么不能直接在旧的源码上把方法名字改了，改成execute()不就行了？其实是很危险的，因为可能大程序中若干个地方都用到了原方法，这里只是想和computer类的execute进行适配，如果源码直接改动，可能虽然满足了这里的效果，但其它地方会报一堆错
- 结果如下：

```
execute a computer program
play a synthesizer
an human speaks
```

七、网络编程

1.基本概念

（1）数据交换

- 电路交换
  - 预分配
  - 链路独享
- 分组交换
  - 端到端时延变动，不可预测，不适合实时服务，但 按需分配，带来更好的带宽共享，实现成本更低
  - 使用报文（package）

（2）网络结构

- ISP
  - Internet Service Provider
  - 一个或多个分组交换机在多段通信链路组成的网络
  - 提供不同类型的网络接入
  - 独立运营
- ISP分层

    - 第一层：因特网主干
    - 第二层：具有区域性或国家性覆盖规模，引导流量通过第一层， 也可以跟其他第二层ISP交换流量
    - 较低层通过一个或多个第二层ISP与更大的因特网连接
- 接入点POP：ISP与ISP的连接点，由一个或多个路由器组成
- 网络接入点NAP：由第三方通信公司或因特网主干提供
- 接入方式：
    - 住宅接入：ADSL，光纤
    - 公司接入：以太网
    - 无线接入：无线局域网，广域无限接入网

（3）数据传输

- 物理层
- 数据链路层
- IP层
- 传输层（TCP/UDP 在这里）
- 应用层

2.TCP编程

（1）基本用法

- sock = socket(AF_INET,SOCK_STREAM)
- AF_INET对应IPv4协议，AF_INET6对应IPv6协议
- SOCK_STREAM指定使用面向流的TCP协议
- bind()绑定端口，端口号应不小于1024：因为0~1023为系统端口号，不能用
- listen()监听端口，并指定等待连接的最大数量
- accept()等待并返回一个客户端连接
- connect()主动连接服务端
- recv()接受TCP数据
- send()发送TCP数据

（2）特点

- 学网络我们知道TCP建立的可靠的连接，是要建立连接之后才进行网络传输，并通过TCP三报文握手保证其可靠性

（2）代码示例

1. Client-Server单向通信

Client端：

```
import os
import sys
import time
import random
from socket import *

class TCPClient:

  def __init__(self,server_ip,server_port):
    self._server_ip=server_ip
    self._server_port=server_port
    self._client=socket(AF_INET,SOCK_STREAM)

  def start(self):
    self._client.connect((self._server_ip,self._server_port))
    while True:
      msg=input("你好，我是Client?")
      self._client.send(msg.encode('utf-8'))
      data=self._client.recv(1024)
      if not data:
        continue
      if(data.decode('utf-8')=='再见 !'):
        print("再见！祝你好运！")
        break
      else:
        print("SERVER: %s" % data.decode('utf-8'))
    self._client.close()

def main():
  client=TCPClient(sys.argv[1],int(sys.argv[2]))
  client.start()

if __name__=='__main__':
  main()
```

- Client端初始化要知道服务器端的IP地址和通信端口
- 客户端启动start时，根据服务器端IP地址和端口主动connect服务器，并通过utf-8编码后向服务器端send信息，recv(1024)表示每次最多接受1024字节
- 客户端发送和接收数据在循环中进行，每进行一次循环则尝试一次数据接受，并将接收到的数据打印出来，若未接收到数据，通过continue继续循环，若接收到数据解码后为：“再见”，则循环终止，通过close()关闭

Server端：

```
import os
import sys
import time
import random
from socket import *

HOST = '127.0.0.1'
RSIZE = 1024 #设置接受数据最大值

class TCPServer:

    def __init__(self, port, maxconnections=5):
        self._port = port
        self._maxconnections = maxconnections
        self._server = socket(AF_INET, SOCK_STREAM)

    def start(self):
        self._server.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1) #设置给定套间字选项的值
        self._server.bind((HOST, self._port))
        self._server.listen(self._maxconnections)
        print("SERVER is listening on %s" % self._port)
```

```
        while True:
            conn, addr = self._server.accept()#接收客户端请求，完成了三次挥手，获取客户端对象和地址
            print(conn)
            print(addr)
            while True:
                try:#监测客户端突然退出异常
                    data = conn.recv(RSIZE)
                    if not data:#linux系统客户端退出不会报错，一直发空包，故实现跨平台
                        break
                    print("CLIENT: %s" % data.decode('utf-8'))
                    if data.decode('utf-8') == 'bye':
                        conn.send("再见 !".encode('utf-8'))
                        break
                    else:
                        conn.send('收到!'.encode('utf-8'))
                except Exception as e:
                    print("SERVER ERROR: %s" % e)
                    break
            conn.close()
        self._server.close()

def main():
    ser = TCPServer(int(sys.argv[1]))
    ser.start()


if __name__ == '__main__':
    main()
```

- Server端应首先开启，其IP地址和端口自己指定，在本机测试时，设定为127.0.0.1（环回地址），初始化时，除指定端口外，还需指定最大连接数maxconnections
- 在start中，先通过bind绑定IP地址和端口号，再通过listen()开启监听，使用两层循环，第一层循环遇到accept时会阻塞（阻塞的机理应该和写在死循环中也有关系），直到客户端进来才会继续进行（完成了三报文握手）
- 再第二层循环中接受客户端发来的数据并回复，用try处理其它异常情况
- 个人认为这里的max_connections没有作用

2. 一个Server，多个Client进行通信

Server端启用多线程，服务多个Client端

```
import sys
import os
from threading import Thread
from socket import *

BUFFERS = 1024
MAXC = 64


def speak(name, conn):
    print("{}已连接成功！".format(name))
    while True:
        try:
            msg = conn.recv(BUFFERS)
            if not msg:
                break
            print("{}:{}".format(name, msg.decode('utf-8')))
            conn.send("我收到了".encode('utf-8'))
            if msg.decode('utf-8') == 'byebye':
                print("你也再见...".format(name))
                break
        except Exception as e:
            print("server error %s" % e)
            break
```

```
    conn.close()


if __name__ == '__main__':
    ip = sys.argv[1]
    port = int(sys.argv[2])
    server = socket(AF_INET, SOCK_STREAM)
    server.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    server.bind((ip, port))
    server.listen(MAXC)

    while True:
        conn, addr = server.accept()
        ci, cp = addr
        # print(addr)
        t = Thread(target=speak, args=("client-" + ci + "-" + str(cp), conn))
        t.start()
    server.close()
```

- 相比第一种多了多线程的处理，即

  `t = Thread(target=speak, args=("client-" + ci + "-" + str(cp), conn))`
  `t.start()`

  这里Thread中target是线程运行的一个函数，args是传给函数的参数，在while循环中启动多线程

  关于多线程的相关知识，点击此处

Client端几乎不变

```
import sys
from socket import *

client=socket(AF_INET,SOCK_STREAM)
ip=sys.argv[1]
port=int(sys.argv[2])
client.connect((ip,port))
while True:
  msg=input(">>>:")
  if not msg:
    continue
  if msg=='exit':
    break
  client.send(msg.encode('utf-8'))
  data=client.recv(1024)
  print(data.decode('utf-8'))
client.close()
```

3. 互为Client-Server双向通信

- 这种方法每一端都即为客户端，也为服务器，因此两端相关的代码都要写
- 这个和之前的不同，可以互发消息（只能实现两方的收发），之前的只能Client发出，Server接受

```
from socket import *
from threading import Thread
import sys

BS = 2048

def recv(name, conn):
    while True:
        data = conn.recv(BS)
        if not data:
            break
        if data.decode('utf-8') == 'BYEBYE':
            #一方输入BYEBYE后，另一方显示下面的文字，但还可以发其他的，但输入方已经不能发东西了
            #当两方都输入BYEBYE时，连接才断开
```

```
            print("对方请求终止对话，按BYEBYE可结束对话......")
            break
        else:
            print("%s: %s" % (name, data.decode('utf-8')))


def send(conn):
    while True:
        # print('\n')
        msg = input("")
        if not msg:
            continue
        conn.send(msg.encode('utf-8'))
        if msg == 'BYEBYE':
            break


if __name__ == '__main__':
    ip = sys.argv[1]
    port = int(sys.argv[2])
    name = sys.argv[3]  # server, client
    server = socket(AF_INET, SOCK_STREAM)
    if name == 'server':
        server.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
        server.bind((ip, port))
        server.listen(5)
        conn, addr = server.accept()
    if name == 'client':
        server.connect((ip, port))
        addr = (ip, port)
        conn = server
    tr = Thread(target=recv, args=(str(addr), conn))
    tr.start()
    ts = Thread(target=send, args=(conn,))
    ts.start()
    tr.join()
    ts.join()
    conn.close()
    server.close()
```

- 这里只是启动了两个线程，分别负责接收数据和发送数据，（与上面为了接受多个Client发来的数据而启动多线程不同）

结果如下：

```
#一端
(venv4) C:\Users\chunhui\Desktop\大三上\Modern Program Design\Code>Core_Practice.py 127.0.0.1 8080 server
q
('127.0.0.1', 52587): w

#另一端
(venv4) C:\Users\chunhui\Desktop\大三上\Modern Program Design\Code>Partner_Practice.py 127.0.0.1 8080 client
('127.0.0.1', 8080): q
w
```

3.UDP编程

（1）特点

不同于TCP的可靠地采用三报文握手的连接方式，UDP的传输是更方便快捷但不保证可靠性的传输，只要给定ip和端口，不需要connect，即可直接发送，这种特点带来了许多优势

（2）优势

- 应用层能更好地控制要发送的数据和发送时间

- 无需连接建立
- 不需要任何准备即可进行数据传输
- 不引入建立连接的时延
- 无连接状态（联想tcp里面的connect）
- 无需维护一些状态参数
- 分组头部开销小（计算机网络用语）

（3）编程部分函数

- socket(AF_INET, SOCK_DGRAM)
- bind()绑定端口，端口号应不小于1024
- recvfrom()接收UDP数据
- sendto()发送UDP数据

（4）示例代码

服务器：

```
from socket import *
import sys

ipaddr=sys.argv[1]
port=int(sys.argv[2])
recv_size=1024

udp_server=socket(AF_INET, SOCK_DGRAM)
udp_server.bind((ipaddr, port))
print("bind udp on port %s" % port)
while True:
  data,addr=udp_server.recvfrom(recv_size)
  if not data:
    break
  else:
    print("%s: %s" % (addr,data.decode('utf-8')))
    udp_server.sendto("收到".encode('utf-8'),addr)
```

- 因为无需连接建立，只需一层while循环用于数据传输即可
- 相对于tcp连接其实简化了很多

4.邮件编程

（1）SMTP

1. 简介：
   - SMTP(Simple Mail Transfer Protocol)，是从发送方的邮件服务器向接收方的邮件服务器发送邮件
   - 运行在发送方邮件服务器的客户机端和接收方邮件服务器的服务器端（客户机端向服务器端发）
   - 在25端口建立与服务器的**TCP连接**
2. 编程函数
   - email模块负责邮件构建
   - smtplib模块负责邮件发送

3. 示例代码

简洁版

```
from email.header import Header
from email.mime.text import MIMEText
from smtplib import SMTP, SMTP_SSL

smtp_server='smtp.qq.com'
port=465#这个端口其实是取决于服务器的
#输入自己的Email地址和口令
from_addr='1183890660@qq.com'
passwd='*********'#此处不用密码，而是用qq邮箱客户端给出的授权码

#输入收件人的邮箱地址
to_addr='baozongbo@qq.com'

message=MIMEText('myxhaokeai', 'plain', 'utf-8')#第一个参数是邮件正文，plain表示纯文本
message['From']=Header(from_addr, 'utf-8')
message['To']= Header("bzb", 'utf-8')
message['Subject'] = Header('an test of SMTP', 'utf-8')#主题

server=SMTP_SSL(smtp_server,port)
server.login(from_addr, passwd) #登陆SMTP服务器
print('登陆成功')
print("准备发送邮件")
server.sendmail(from_addr, [to_addr], message.as_string())#发邮件，可以一次发给多人
server.quit()
print("发送完成！")
```

- 注意：465仅针对于qq邮箱服务器而言，如图

域名邮箱的POP3和SMTP服务器地址跟QQ邮箱是一样的，如下：

POP3服务器（端口110）	SMTP服务器（端口25）
pop.qq.com	smtp.qq.com

SMTP服务器需要身份验证。

域名邮箱的客户端收发也支持SSL加密方式，需要设置如下端口：
POP3服务器（端口995）；
SMTP服务器（端口465或587）。

- 若用其它邮箱服务器，应前往客户端的帮助中心进行查询

增强版

```
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.application import MIMEApplication
from email.header import Header
from email.utils import parseaddr, formataddr
from smtplib import SMTP_SSL

#用来格式化一个邮件地址
def _format_addr(s):
    name, addr = parseaddr(s)
    return formataddr((Header(name, 'utf-8').encode(), addr))
```

```
smtp_server='smtp.qq.com'
port=465
from_addr='jichang.zhao@qq.com'
passwd='jmxflofuugodbgga'
#to_addr='python_sem@163.com'
to_addr='liyashuai@buaa.edu.cn'

message=MIMEMultipart()#发送附件时，改为这个
message['From']=_format_addr('从这里发来：%s' % from_addr)
message['To']= _format_addr('给%s' % to_addr)
message['Subject']=Header('zjc', 'utf-8').encode()

#html
message.attach(MIMEText('<html><body><h1>Hello</h1>' +
    '<p>send by <a href="http://www.python.org">Python</a>...</p>' +
    '</body></html>', 'html', 'utf-8'))

#py文件
textfile=MIMEText(open('smtp.py','rb').read(),'base64','utf-8')
textfile['Content-Type']='application/octet-stream'
textfile['Content-Disposition']='attachment; filename="smtp.py"'
message.attach(textfile)

#发送图片
picfile=MIMEImage(open('y.jpeg', 'rb').read())
picfile.add_header('Content-Disposition', 'attachment', filename="y.jpeg")# 加上必要的头信息
message.attach(picfile)

#发送word文档
docfile=MIMEApplication(open('test.docx','rb').read())
docfile.add_header('Content-Disposition', 'attachment', filename="test.docx")
message.attach(docfile)# 添加到MIMEMultipart:

server=SMTP_SSL(smtp_server,port)
server.login(from_addr, passwd)
print('登陆成功')
print("准备发送邮件")
server.sendmail(from_addr, [to_addr], message.as_string())#发邮件，可以一次发给多人
server.quit()
print("发送完成！")
```

（2）POP3（暂时略去）

> 详见下面链接：

POP3收取邮件
https://www.liaoxuefeng.com/wiki/1016959663602400/1017800447489504

（3）IMAP（略去）

八、数据库编程

💡 本节内容在下学期数据库应用实验上应该会详细介绍，这里仅概况基本内容

1.数据库连接与操作

（1）关系数据库

1. psycopg2(以下是未执行过的伪码)
   - 一般的使用逻辑

```
import psycopg2 #导入库
#创建数据库会话
conn = psycopg2.connect("dbname=test user=postgres password=secret")
#创建游标并通过游标执行SQL语句
cur = conn.cursor()
#执行SQL语句（创建、插入、查询等）
cur.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data varchar);")
cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)",(100, "abc'def"))
cur.execute("SELECT * FROM test;")

#通过会话完成事务的提交或回滚
conn.commit()#提交
#conn.rollback()#回滚
#关闭数据库会话
cur.close()
conn.close()
```

   - 利用with语句进行连接和游标的管理

```
with psycopg2.connect(DSN) as conn:
  with conn.cursor() as curs:
    '''
    SQL语句
    '''
#多次使用
conn = psucopg2.connect(DSN)
with conn:
  with conn.cursor() as curs:
    curs.execute(SQL1)
with conn:
  with conn.cursor() as curs:
    curs.execute(SQL2)
conn.close() #直以推出with上下文时并不关闭连接
```

   - 给SQL语句传参

```
#execute最后一个参数是元组或字典
cur.execute('INSERT INTO foo VALUES ( %s )', ('bar'))
cur.execute('INSERT INTO numbers VALUES ( %s )', (10,))
```

   - 结果的获取

```
#cursor实例本身可迭代
cur.execute("SELECT * FROM test;")
for record in cur:
  print(record)
fetechone()#返回tuple
fetchmany([size])#返回tuple的list，如果无数据即返回[]
fetchall()#返回tuple的list

#以字典的形式返回执行结果
dict_cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
dict_cur.execute("SELECT * FROM test")
rec = dict_cur.fetchone()
```

- 批量操作
- 协程支持
- 异步IO

2. pymysql（建议自学并了解）

（2）非关系数据库

1. pymongo(以下是未执行过的伪码）

- 一般的使用逻辑

```
#创建连接
from pymongo import MongoClient
client = MongoClient()
client = MongoClient('localhost', 27017)
client = MongoClient('mongodb://localhost:27017/’)
#指定数据库
db = client.test_database
db = client['test-database']
#指定集合collection
collection = db.test_collection
collection = db['test-collection']
#插入文档
insert_one()
#批量插入
insert_many()
```

- 返回结果

```
#returns a single document matching a query (or None if there are no matches)
find_one([query])
#To get more than a single document as the result of a query
find([query])
#returns a Cursor instance that can be iterated
find().limit(size)
```

- 计数、排序

```
#返回满足要求的文档数
count_documents({})
#对查询结果进行排序
sort(“name”,1) #ascending
sort(”name”,-1) #descending
```

- 删除文档和更新

```
#删除文档
delete_one(myquery)
delete_many(myquery)
delete_many({})#删除collection中的所有文档
drop()#删除整个collection
#更新
myquery = { "address": "Valley 345" }
newvalues = { "$set": { "address": "Canyon 123" } }
update_one(myquery, newvalues)
myquery = { "address": { "$regex": "^S" } }
newvalues = { "$set": { "name": "Minnie" } }
x = mycol.update_many(myquery, newvalues)
```

2.ORM简介

- 对象关系映射(Object Relational Mapping)

（1）面向关系数据库的ORM

- 包含

  sqlalchemy

  peewee

  PonyORM

  Django ORM

- 一般的逻辑
  - 创建Mapping：业务逻辑中的实体类与数据库的表建立对应关系
  - 构建数据加会话并进行存储或查询

（2）面向非关系数据库的ORM

- 包含

  Django ORM

  MongoEngine

  MongoKit

  Ming

3.Web开发框架简介

框架简介

框架	ORM	Database connector	Relational Database
Bottle	PeeWee	psycopg	PostgreSQL
Flask	Pony ORM	psycopg	PostgreSQL
Flask	SQLAlchemy	psycopg	PostgreSQL
Django	Django ORM	psycopg	PostgreSQL

- 上述框架了解一个就行，比如Django

九、常见函数/模块介绍

这里列举了一些内置的比较重要的，或者常用的模块，用法以后补齐

1.内置的

enumerate

json

os

zipfile

collections

time

math

sys

random

2.外部的

numpy

matplotlib

@Chunhui Li

> 最后整理时间 2019-12-26 21：43