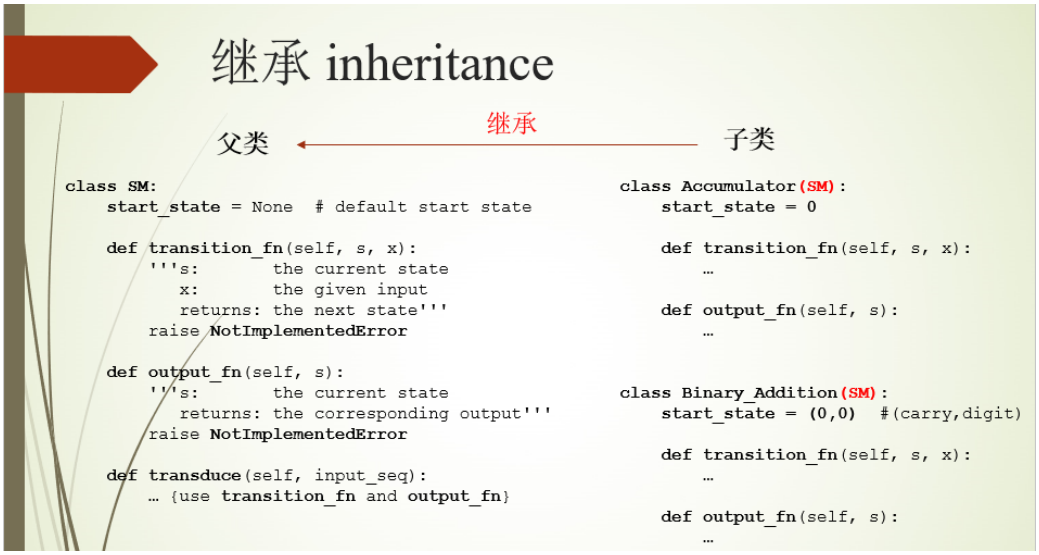
Python 程序设计 作业 2

注意事项：

（1） 作业提交**截止日期：2022.06.10，23:59pm**，迟交扣 20%，缺交 0 分。
（2） 提交方法：**Blackboard。**
（3） 提交要求：作业回答（word 文件）+源代码，打包上传，命名：学号_姓名_作业_2。
（4） 作业回答写解题思路、运行结果等（不需要使用实验报告模板）。
（5） 如同一题目需要多个 Python 文件运行，请写清楚程序运行方法。
（6） **禁止代码抄袭；一经发现，抄袭者和提供代码者统一 0 分处理！**

1. 完成课程讲义中状态机的实现。



(1) 累加器 Accumulator

例子：列表[-1, 2, 3, -2, 5, 6]中的数字相加，我们得到和是 13。

output = Accumulator().transduce([-1,2,3,-2,5,6]) #output=[-1, 1, 4, 2, 7, 13]

(2) 二进制相加 Binary_Addition

例子：011 和 001 相加我们得到 100。

output = Binary_Addition().transduce([(1,1),(1,0),(0,0)]) #output=[0, 0, 1]

(3) 反向器 Reverser

实现一个反向器，逆向输出一个序列。输入的形式如下：

sequence1 + [‘end’] + sequence2

其中 sequence1 是字符串列表，‘end’表示 sequence1 的终结，sequence2 为任意序列。反向器的作用如下：当 sequence1 里的每个元素输入时，反向器

输出 None。当[‘end’]和 sequence2 的每个元素输入时，反向器依次反向输出 sequence1 中的元素；当没有元素输出时，则输出 None。

例子：

```
output = Reverser().transduce(['foo',' ', 'bar'] + ['end'] + list(range(5)))
#output = [None, None, None, 'bar', ' ', 'foo', None, None, None]
```

解释：输出中前三个 None 为输入 sequence1 时输出。从输入‘end’开始，sequence1 中的元素开始反向输出。sequence1 输出完毕后，继续输出 None（即输入 list(range(5))中的 2，3，4 时）。

更多测试例子：

```
output=Reverser().transduce(['foo', ' ', 'bar'] + ['end'] + ['end']*3 +list(range(2)))
#output = [None, None, None, 'bar', ' ', 'foo', None, None, None]

output = Reverser().transduce(list('the') + ['end'] + list(range(3)))
#output = [None, None, None, 'e', 'h', 't', None]

output=Reverser().transduce([] + ['end'] + list(range(5)))
#output = [None, None, None, None, None, None]

output=output=Reverser().transduce(list('nehznehS evol I') + ['end'] +
list(range(15)))
#output = [None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, 'I', ' ', 'l', 'o', 'v', 'e', ' ', 'S', 'h', 'e', 'n', 'z', 'h', 'e',
'n', None]
```

实现要求：

（1） 不同的状态机子类继承父类 SM；

（2） 补充父类的 transduce 函数，子类不能重写 transduce 函数；

（3） 补充每个子类的 transition_fn 和 output_fn 函数，其中累加器的实现代码如下：

```
class Accumulator(SM):
    start_state = 0
    def transition_fn(self, s, x):
        return s + x
    def output_fn(self, s):
        return s
```

建议：利用累加器的代码实现父类的 transduce 函数，再实现二进制相加和反向器的 transition_fn 和 output_fn 函数。

2. 纸牌游戏：

本次作业我们将设计和实现一个扑克牌游戏。游戏规则如下：

(1) 游戏开始时，扑克牌在两名玩家中平分（扑克牌总共 52 张，每位玩家分 26 张）。
(2) 在每个回合中，两个玩家都会展示手中的第一张牌。扑克牌等级较高的玩家将同时获得这两张牌，并添加到自己的牌组中。然后玩家们重新调整手牌。
(3) 若（2）中两位玩家的扑克牌等级一样（即打平），玩家们将展示手中的第二张牌。如果不再有平局，扑克牌等级较高的玩家将同时获得这四张牌；如果继续平局，玩家们将展示手中的第三、四、……张牌，直到平局被打破。
(4) 游戏持续到一个玩家收集完所有 52 张牌为止。若（3）中平局持续到一方无牌，则还有手牌的玩家获胜；若两个玩家都没有牌了，则打平。

想一想，为了实现上面这个游戏，我们需要什么类和方法？我们知道每张扑克牌都有一个花色和一个等级，如红桃 K。因此，创建一个“扑克牌”（Card）的类，并包含这两个“属性”是有意义的。此外，在我们的游戏中，“大”的扑克牌是指等级较高的牌；“打平”的牌是指等级相同的牌（不管两张牌的花色）。所以我们需要自定义一些规则来衡量“大”、“小”、“相等”的牌。

每位玩家都有一“手”牌（Hand），是扑克牌的集合，我们也可以将它写成一个类，可以从一“手”牌中取出牌、或者洗牌。一“手”牌中如果包含所有的 52 张扑克牌，叫牌堆（Deck）。每位玩家（Player）有一“手”牌和名字。

最后我们需要一个类叫 Game，实现游戏的逻辑。

作业任务：

补充提供的文件 card_game.py 中 Card，Hand，Deck，Player，Game 类的实现（标注“补充代码”处）。测试代码以验证正确性。

3. SimpleGraph

本题将在 Python 实现一个类叫 SimpleGraph，表示有向图。我们需要定义一个 Vertex 类表示图的顶点，一个 Edge 类表示图的边，以及 SimpleGraph 类。规范如下：

- Vertex 类至少包含属性：
  - name, 代表顶点标签的字符串。
  - edges，代表从这个顶点向外到其邻居的边的集合。

  一个新的顶点初始化时 name 可选且默认为空字符串，edges 为一个空的集合（自行决定集合的数据类型）。

- Edge 类至少包含属性：
  - start, 代表边起点的顶点 Vertex。

- end, 代表边终点的顶点 Vertex。
- cost, 代表边的权重，类型为 float。

一条边需要一个 start 顶点和 一个 end 顶点，以便被实例化。cost 应该默认为 1。

- SimpleGraph 至少包含属性：
  - verts，图中顶点的集合（自行决定集合的数据类型）。
  - edges，图中边的集合（自行决定集合的数据类型）。

  以及以下方法：
  - add_vertex(v) #添加顶点
  - add_edge(v1, v2) #添加边，起点为 v1，终点为 v2
  - contains_vertex(v) #判断图中是否存在顶点 v
  - contains_edge(v1, v2) #判断图中是否存在边，起点为 v1，终点为 v2
  - get_neighbors(v) #返回顶点 v 的邻居，即从 v 出发一跳可到达的顶点
  - is_empty() #判断图是否为空（即没有顶点也没有边）
  - remove_vertex(v) #删除顶点 v
  - remove_edge(v1, v2) #删除起点为 v1，终点为 v2 的边
  - is_neighbor(v1, v2) #判断顶点 v2 是否为顶点 v1 的邻居
  - is_reachable(v1, v2) #判断是否存在路径可以由顶点 v1 到达顶点 v2
  - clear_all() #清空图，删除所有顶点和所有边