

SHANGHAI JIAOTONG UNIVERSITY



强化学习课程设计 - Project1

Dynamic Programming

姓名：薛春宇

学号：518021910698

完成时间：2021/3/24

1 实验目的

基于动态规划的思想，分别使用Policy Iteration和Value Iteration优化随机策略，以解决Gridworld下的最短路径问题。

2 实验准备

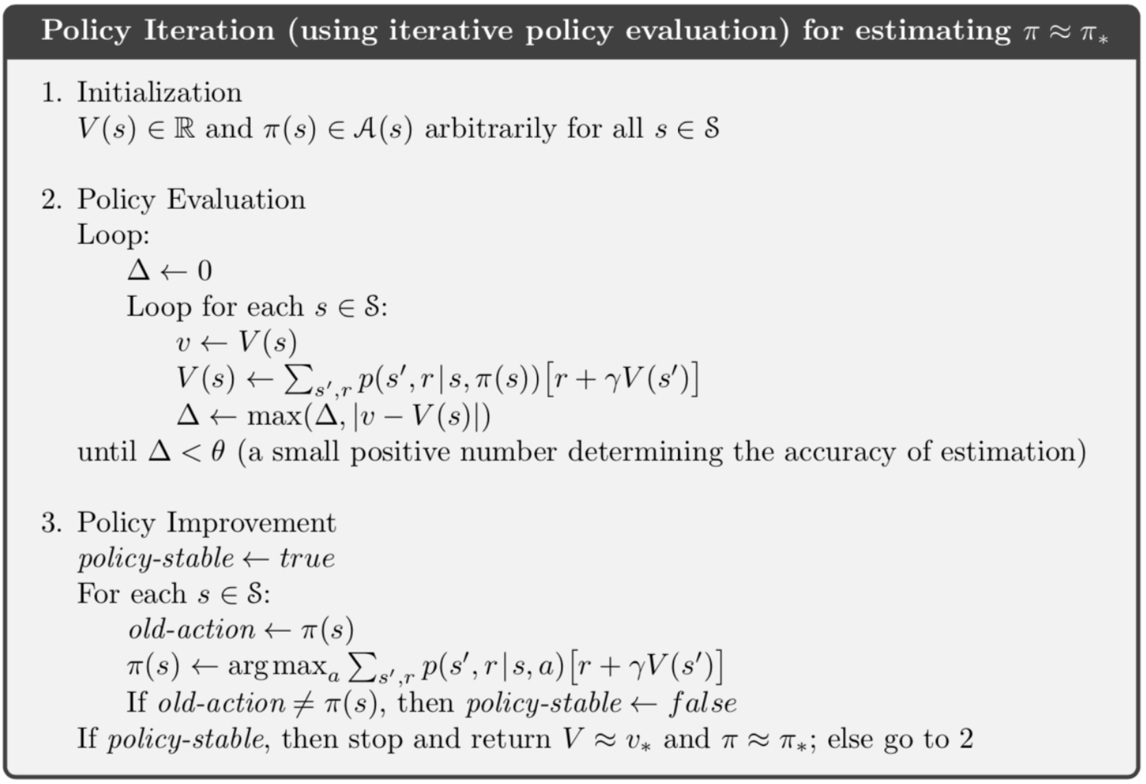
在本节中，我们将分别介绍Policy Iteration和Value Iteration的相关内容，以为接下来的代码实现做准备。

2.1 策略迭代Policy Iteration

Policy Iteration是本次实验中实现起来最具挑战性的模块，其可以分为两个部分，其名称及作用分别是：  
 (1) 策略评估Policy Evaluation：基于已有的策略，遍历MDP中的每一个状态s，使用贝尔曼方程对值函数V进行迭代更新，直到满足结束条件趋于收敛。

(2) 策略提升Policy Improvement：基于已有的值函数V，使用贝尔曼方程对策略遍历MDP中的每一个状态s进行一次更新，并判断策略是否发生了改变，若没有改变，则将值函数V和策略返回；否则重复policy evaluation的步骤。

整个policy iteration的伪代码如下所示：

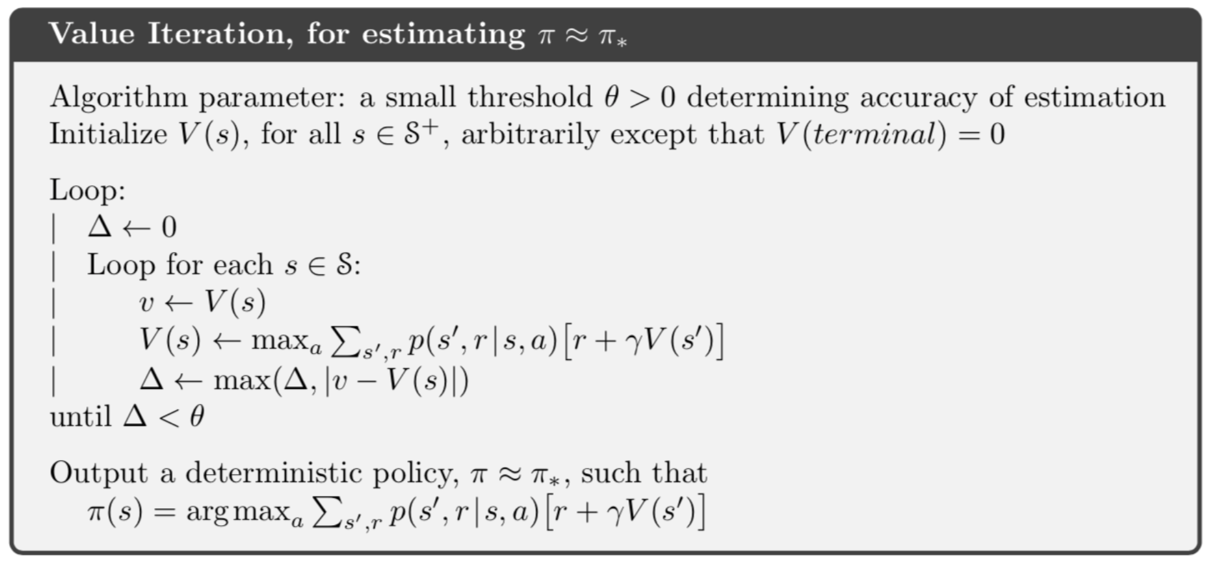
**

我们的python代码也是基本按照上述伪代码的思想进行实现的。

2.2 值迭代Value Iteration

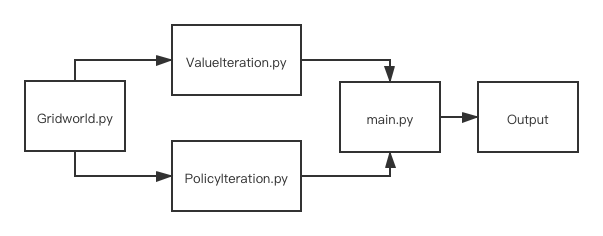
相较于策略迭代，value iteration在整个运行过程中只会在最终输出确定性策略的时候，使用贝尔曼方程，基于已有的值函数V求出策略。Value iteration的主体部分是一个针对值函数V的迭代更新，同样是基于贝尔曼方程，并设置一个阈值来控制迭代的结束。

Value iteration的伪代码如下所示：



3 实验内容

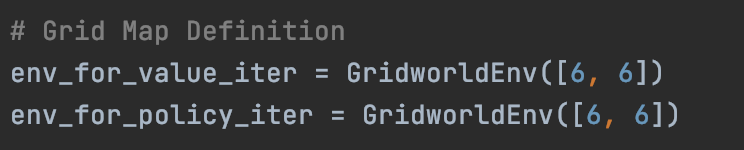
本次实验的代码及逻辑结构如下所示：



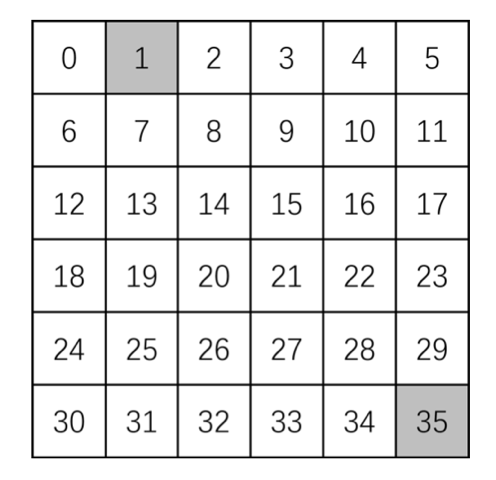
接下来，我们将对四个主要文件进行逐一分析，并分析两类迭代算法的性能差异。相关代码位于./code目录下。

3.1 Gridworld.py

该模块的实现参考了Sutton版RL提供的Gridworld类，实现了一个可实例化的格子世界，可以通过如下指令进行实例化：

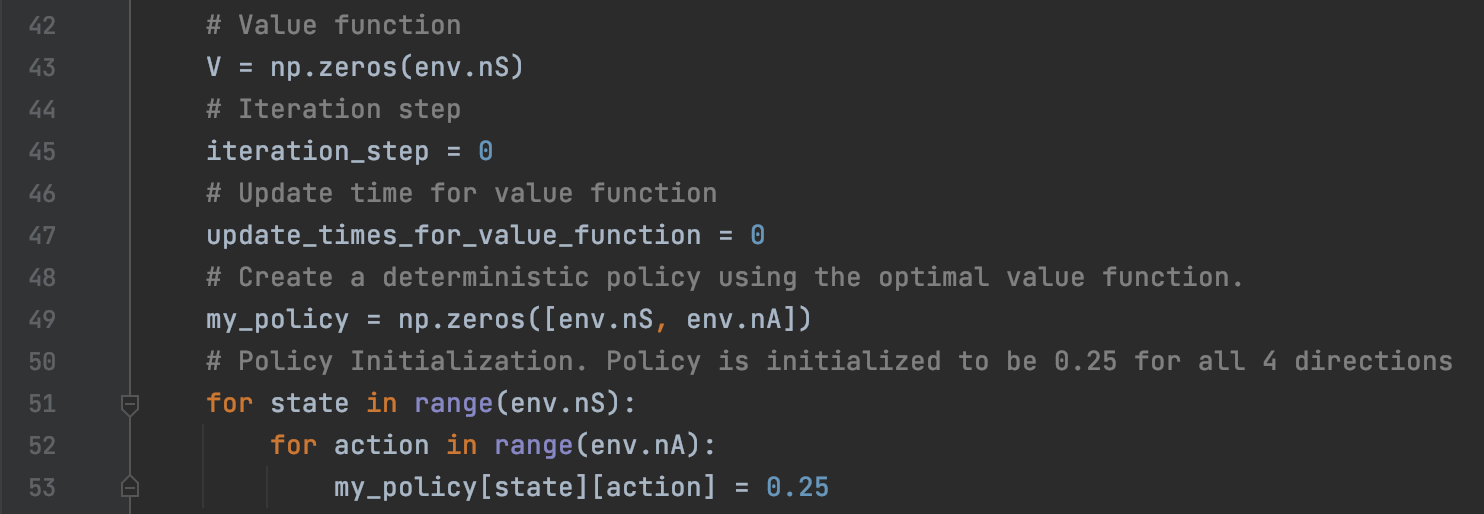


需要注意的是，本项目中的Terminal State、Action Tpye、Reward和概率转移矩阵P均需要在Gridworld.py中声明。本项目中的Gridworld需要被实例化为如下结构（其中1和35号grid被设置为terminal state）：

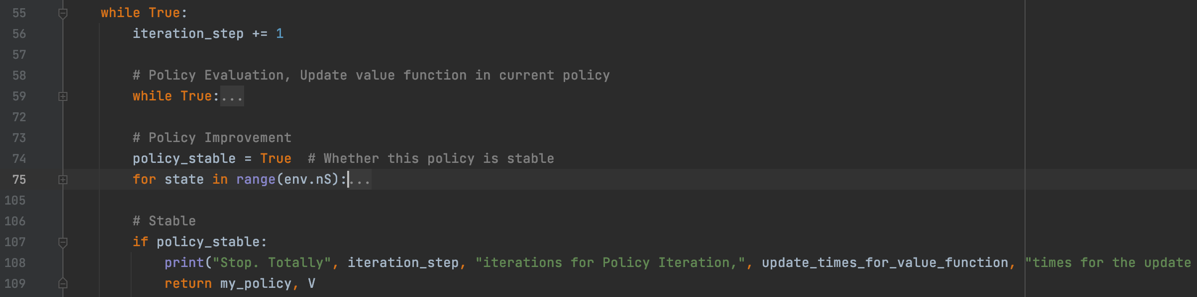


3.2 PolicyIteration.py

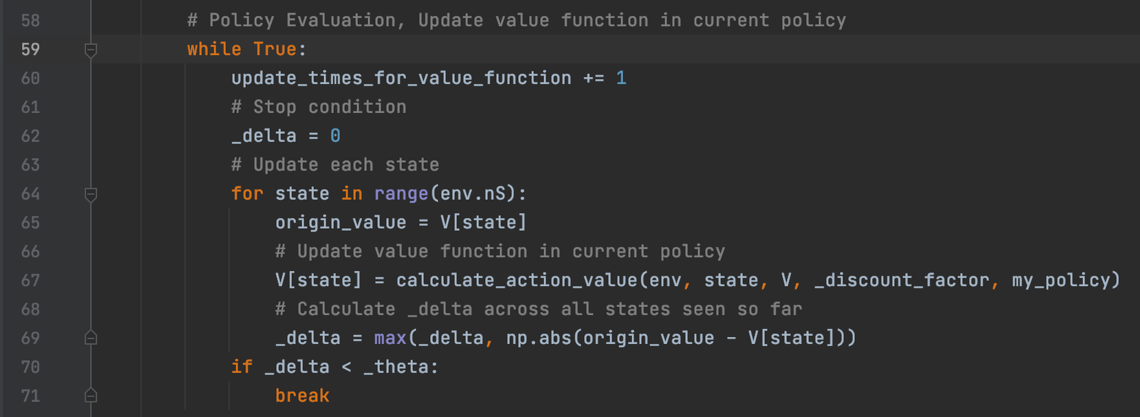
该模块的实现基本遵循了2.1节中给出的伪代码的思想，并将整体内容设置成一个可调用的函数policy\_iteration()，供用户在main.py中调用。



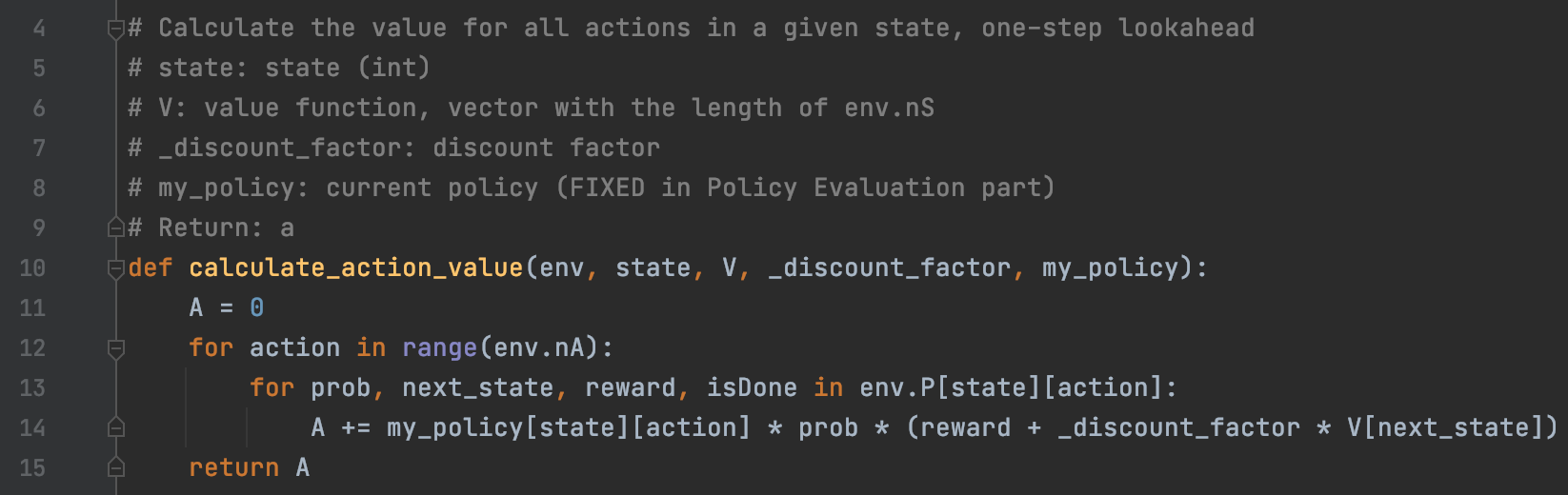
在函数的开始部分，我们根据已知的参数大小（env.nS为Gridworld的全部状态数，env.nA为Gridworld的全部行为数）来创建值函数及策略的容器，并进行策略的初始化（四个方向各位0.25）。



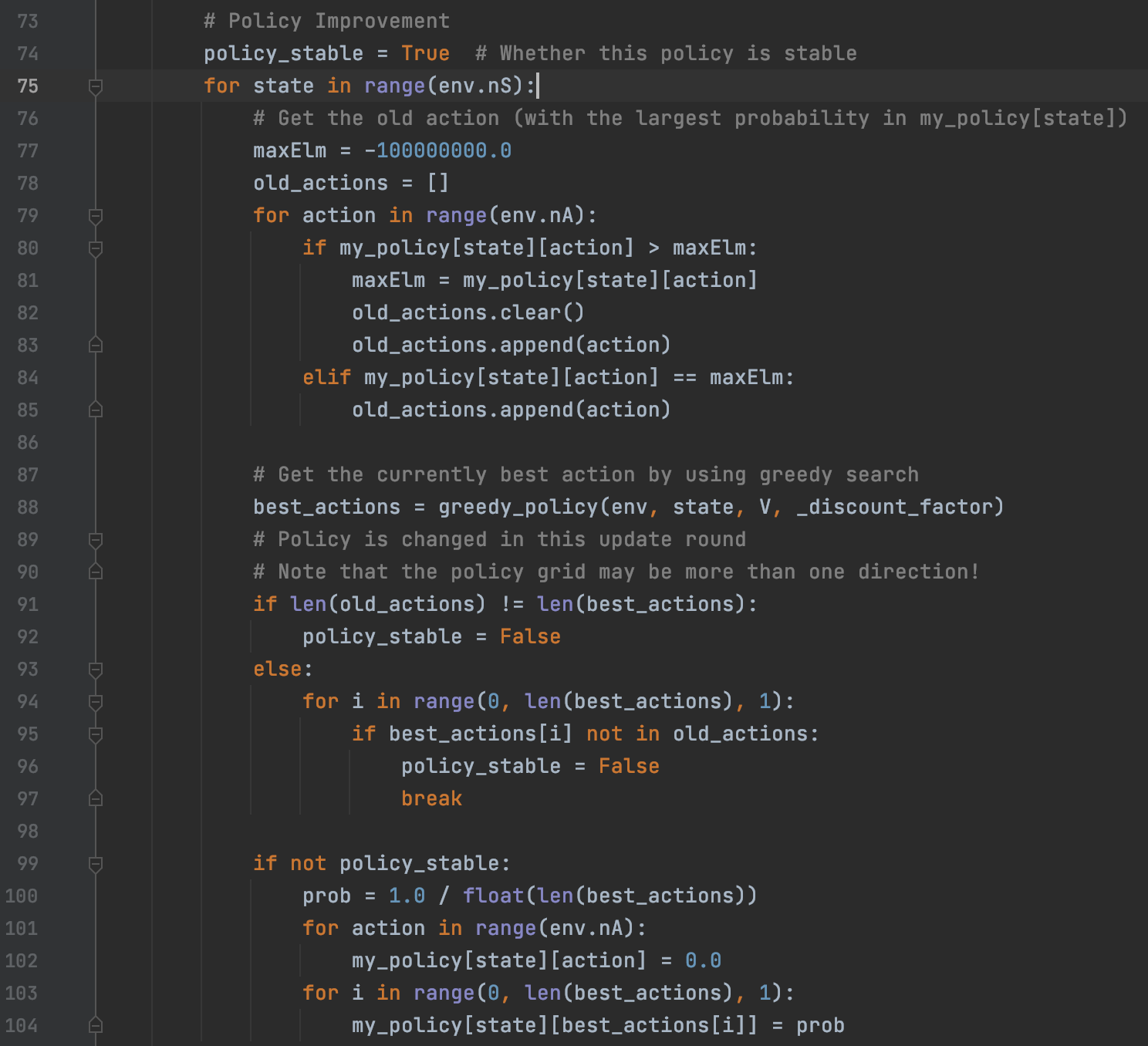
函数主体为一个while循环，分为policy evaluation和policy improvement两个子模块，最后若策略稳定，则结束循环将策略和值函数返回。



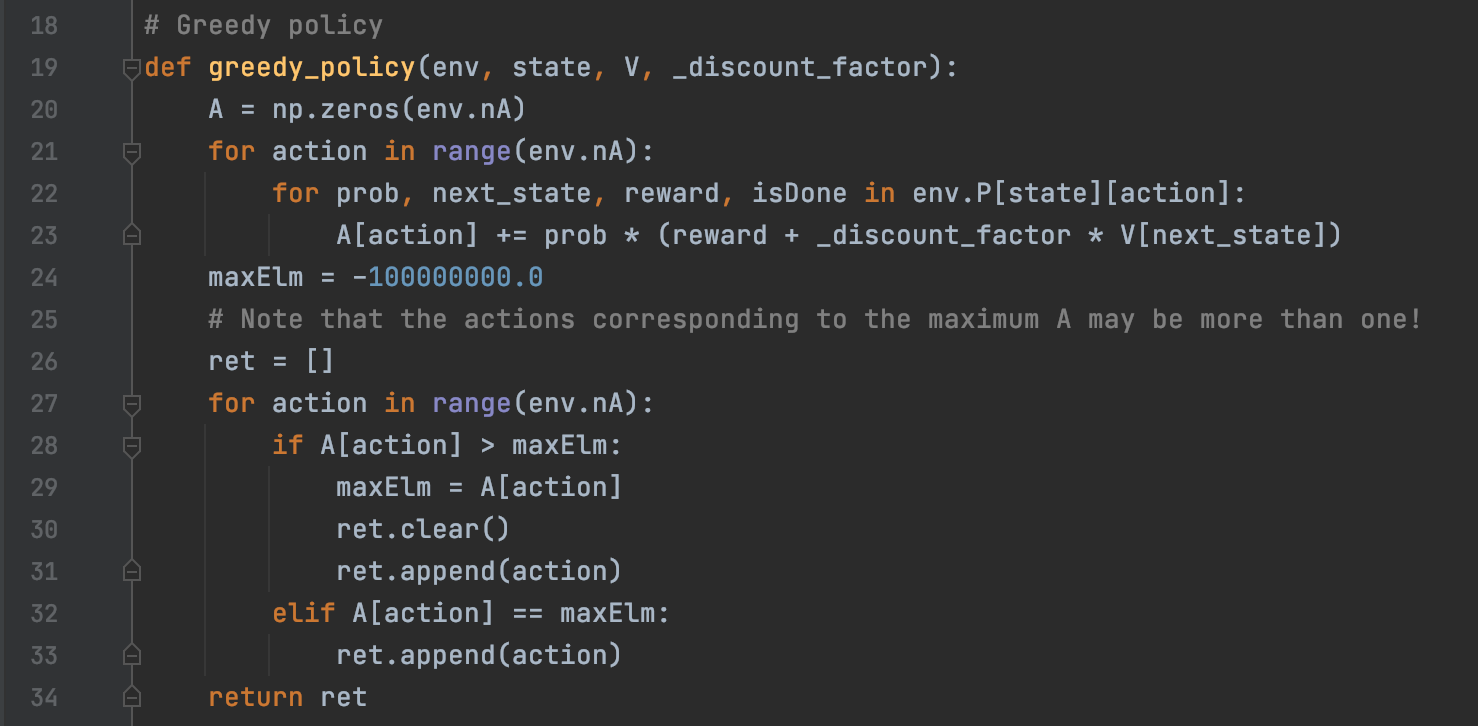
第一个模块为策略评估policy evaluation，设置\_delta为停止条件，在一个while循环中，调用calculate\_action\_value()函数实现贝尔曼方程对值函数V进行更新迭代，并更新\_delta，当小于预设的阈值\_theta时退出该模块。子函数的实现如下：



Calculate\_action\_value()函数严格按照policy evluation的伪代码进行实现。



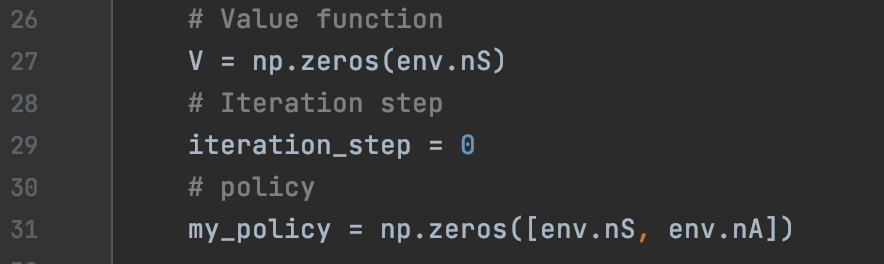
第二个模块为策略提升policy improvement，设置一个布尔变量policy\_stable指示在本次执行中策略是否发生了变化。执行主体为一个遍历所有状态s的for循环，需要特别的注意的是，在判断old\_policy和best\_policy是否相同的时候，需要考虑到二者均可为一个包含一个以上action的列表，原因是在调用贪心算法greedy\_policy()寻找当前值函数的最优策略时，每个state对应的最优action可能并不止一个：



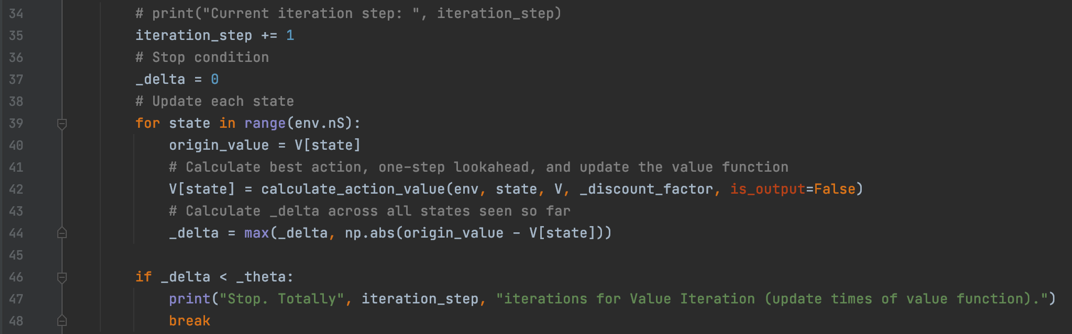
因此在判断最大值的时候需要特别注意，需要使用一个list来容纳可能不止一个的best action并返回。此外，在更新policy的时候，需要将位于list中的action对应的my\_policy[state][action]置为1/len(list)，其他则置为0。

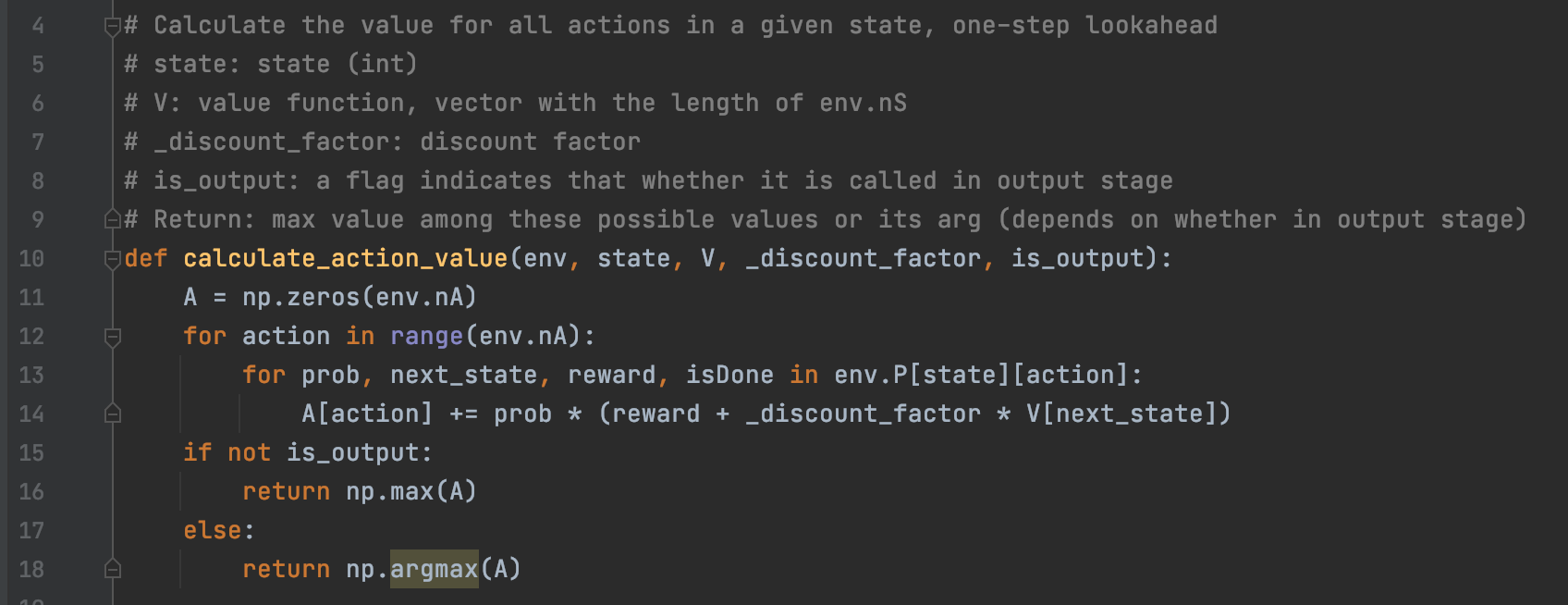
3.3 ValueIteration.py

该模块的实现基本遵循了2.2节中给出的伪代码的思想，并将整体内容设置成一个可调用的函数value\_iteration()，供用户在main.py中调用

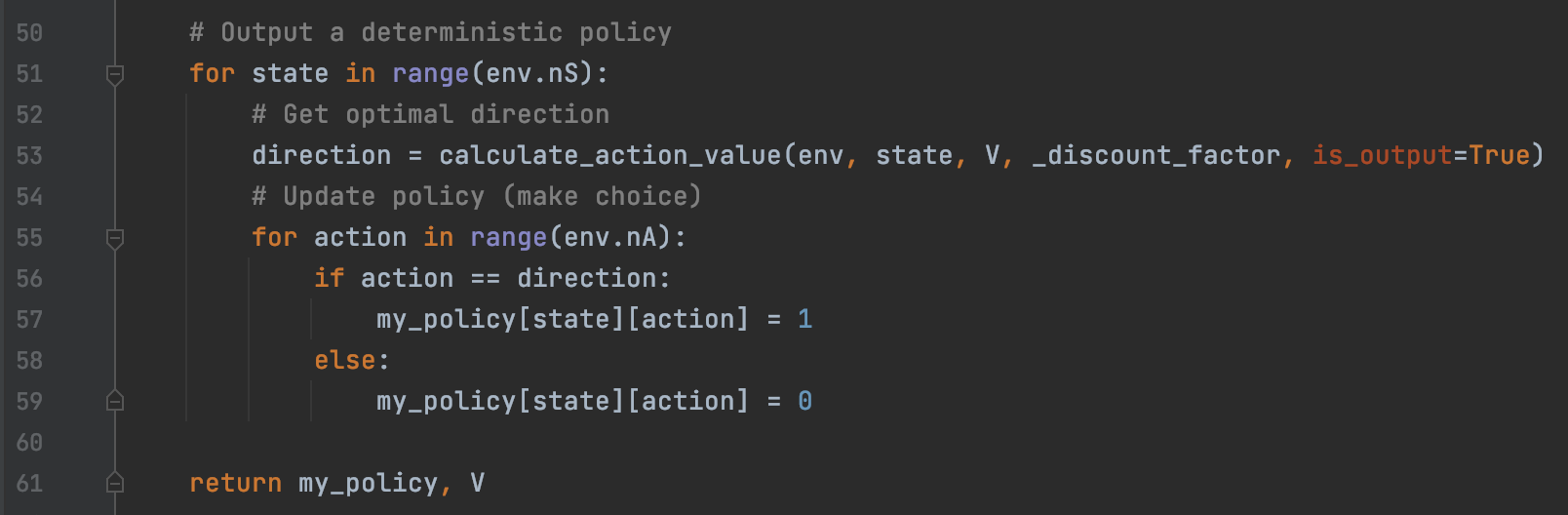


在函数的开始部分，我们根据已知的参数大小（env.nS为Gridworld的全部状态数，env.nA为Gridworld的全部行为数）来创建值函数及策略的容器。由于本模块是对值函数而非策略进行迭代，我们不需要单独对my\_policy容器进行初始化。



模块主体由一个while循环组成，第一个子模块为对值函数V的迭代。这里调用了calculate\_action\_value()函数来实现贝尔曼方程：  


值得注意的是，在最后输出确定性策略的时候，我们也会调用该函数，不过会将is\_output设置为True，以获得每个state的最优action。



输出确定性策略的子模块如上所示。

3.4 main.py

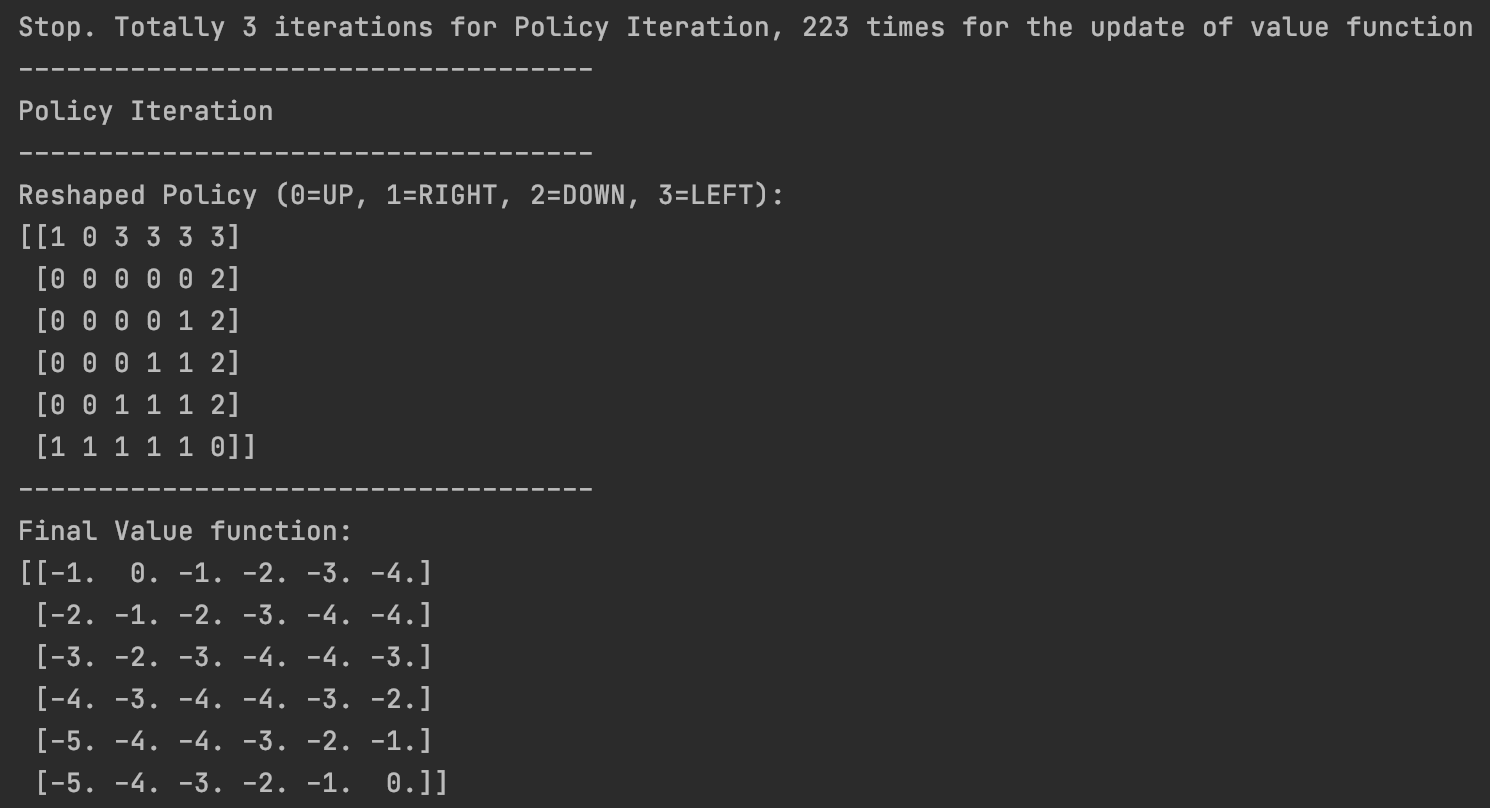
最后的主函数主要负责Gridworld类的实例化，两类迭代函数的调用以及输出的格式化。



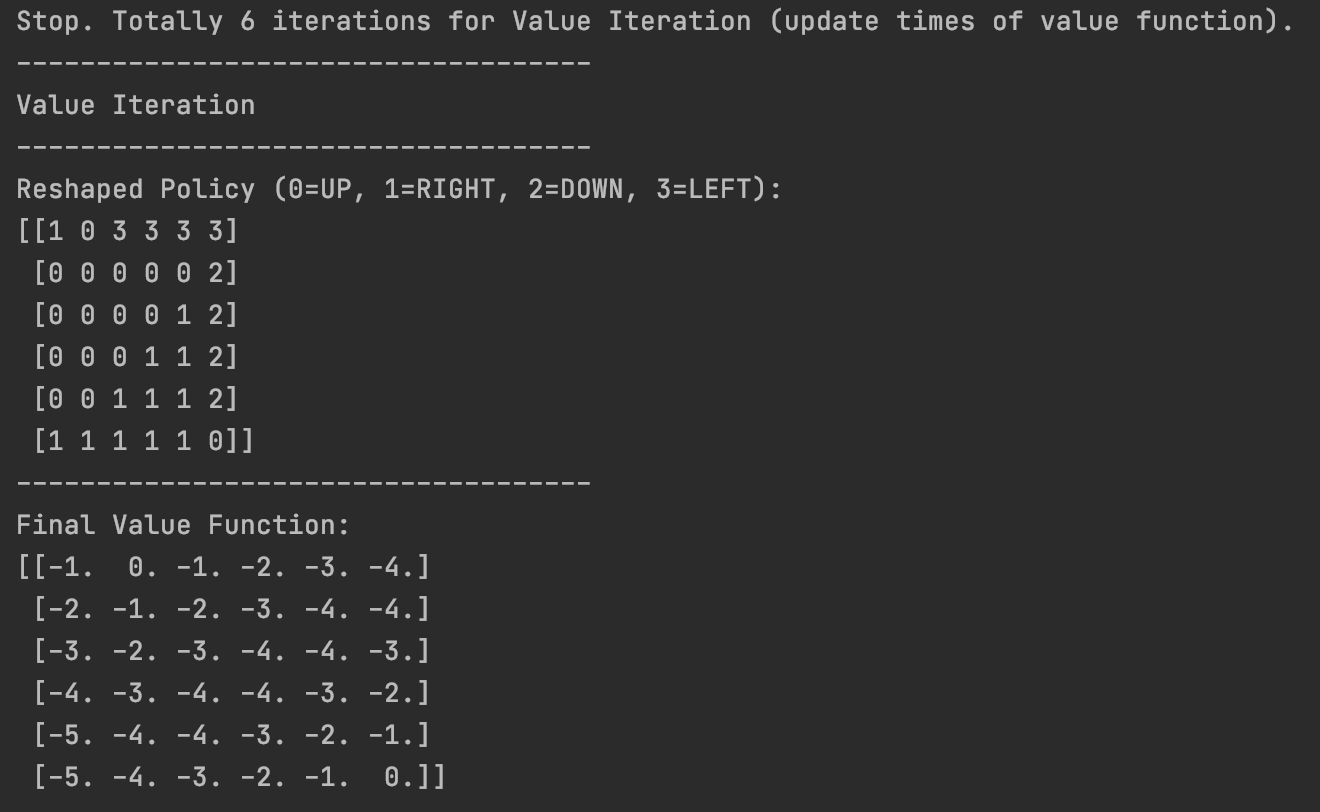
其中，value\_iteration()和policy\_iteration()分别在value\_iteration\_policy()和policy\_iteration\_policy()两个封装函数中调用，输出的格式化也在里面实现。

3.5 Policy Iteration和Value Iteration的性能对比

策略迭代的运行结果如下：



值迭代的运行结果如下：



根据上述结果可以看到，虽然policy iteration的值函数更新次数（223次）要远多于value iteration的值函数更新次数（6次），但其总的迭代次数（3次）相比之下要比value iteration的迭代次数（6次）少一半，收敛速度也比较快。但由于其每次迭代都需要进行策略的生成，总体的运行速度还是要比value iteration要慢。实际上，我们可以采取一定的方法让policy iteration提前终止，以提高其运行速度。

4 实验结果

对比两种迭代方案的运行结果可以看到，两类迭代方案的结果完全相同且正确。本次实验成功。

5 实验心得

在本次实验的过程中，我首先在课堂授课内容的基础上，系统性地学习了policy iteration和value iteration的相关知识，并掌握了其在代码层面上的实现方法。在实现的过程中，由于从伪代码到python代码的差别还是比较大的，我遇到了一些困难，其中就包括了在判别策略是否稳定的过程中，没有考虑到一个state可能会对应多种最优action的情况，因此走了不少弯路。

整个过程约花费半天时间，在本次实验中，我不仅了解了两种迭代方案的基本知识，掌握了从伪代码到可运行代码的复现方法，更是提高了自身发现bug，解决bug的能力。希望在接下来的实验中也能收获满满！