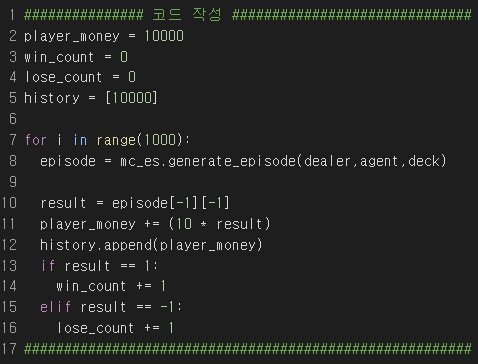
* **목표**

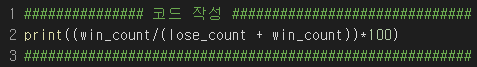
주어진 코드로부터 최적 Policy를 학습시킨 Agent를 사용하여, 블랙잭 게임을 시뮬레이션한 뒤 승률을 계산.

* **초기 설정**
* 플레이어는 1,000,000번의 에피소드로부터 최적 Policy를 학습
* 플레이어의 초기 자금: 10,000 달러
* 플레이어는 게임 참가 시 10 달러를 지불, 결과에 따라 금액을 획득
  + 승리 시: 20 달러 획득
  + 무승부 시: 10 달러 획득
  + 패배 시: 0 달러 획득
* **요구사항**
* 플레이어와 딜러가 1,000번의 게임을 진행
* 1,000번의 게임 후 플레이어의 승률을 계산
* 매 게임 별 플레이어의 소지금 변화를 그래프로 시각화
* **(과제 1) 1,000번의 게임 진행**

작성된**** 코드에서 이미 학습을 완료한 MonteCarlo 객체 mc\_es를 그대로 이용했습니다.

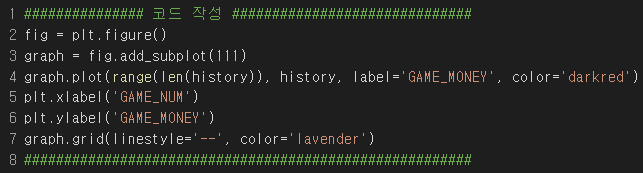
mc\_es의 generate\_episode() 함수를 이용해서 1000번의 게임, 즉 1000개의 에피소드를 생성하고 각 에피소드의 마지막 원소의 reward를 통해 게임의 승자를 판별합니다.

* **(과제 2) 플레이어의 승률을 계산**

****

플레이어의 승률을 계산하기 위해서 1000개의 에피소드를 진행하는 과정에서 에피소드의 reward를 이용해 플레이어가 승리한 게임의 수와 패배한 게임의 수를 저장해 두고 승률을 계산할 때 사용합니다.

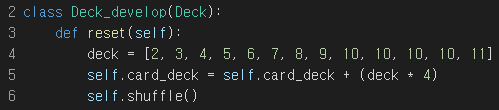
* **(과제 3) 플레이어의 소지금 변화를 그래프로 시각화**

****

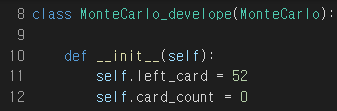
플레이어의 소지금 변화를 시각화하기 위해서 1000개의 에피소드를 진행하는 과정에서 플레이어의 현재 소지금을 list 변수 history에 저장합니다. x축을 진행된 게임의 수, y축을 플레이어의 현재 소지금으로 하는 그래프를 Matplotlib를 이용해 그립니다.

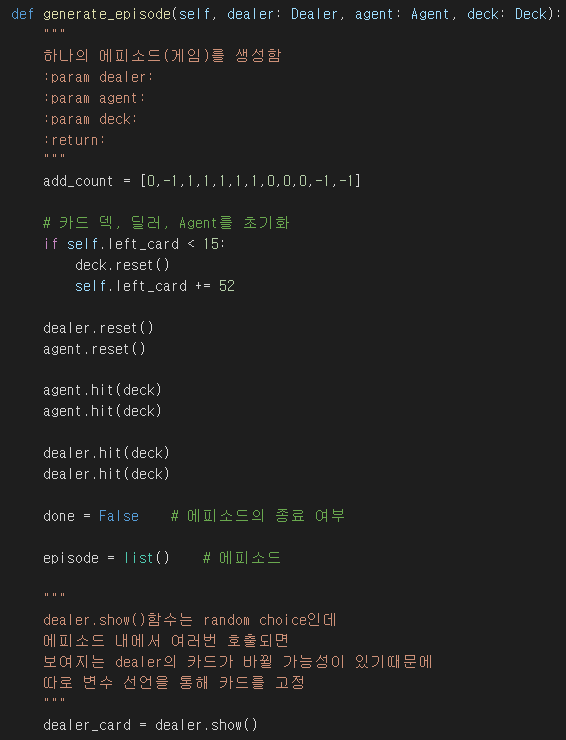
* **(추가 과제) 플레이어의 승률 높이기**
* **State를 수정하여 승률이 더 높은 policy를 찾기**
* **기존 코드의 수정 필요**
  + **딜러는 게임이 끝났을 때, 남은 카드의 수를 확인**
  + **15장 이상이라면 해당 덱을 다음 게임에서 그대로 사용**
  + **15장 미만이라면 52장의 셔플된 새로운 카드를 기존 덱에 추가**

기존의 코드를 수정을 위해 Deck과 MonteCarlo 클래스를 상속받는 새로운 클래스 Deck\_develop과 MonteCarlo\_develop을 정의해 수정이 필요한 함수를 Override하는 방식으로 진행했습니다.



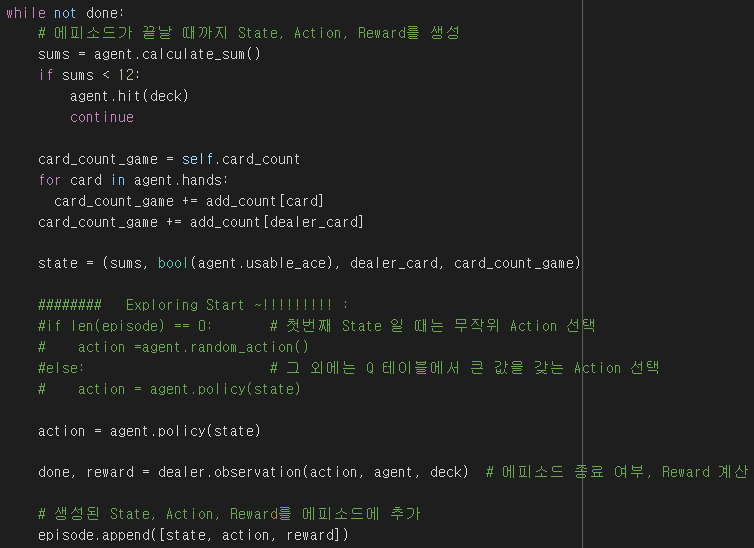
과제의 조건에 따라 딜러가 게임이 끝났을 때, 남은 카드의 수를 확인하고 15장 미만이라면 52장의 셔플된 새로운 카드를 기존 덱에 추가해야 합니다. 따라서 기존의 reset함수와 다르게 card\_deck의 초기화 없이 추가만을 진행했습니다.

그리고 MonteCarlo\_develope 클래스에서 현제 덱에 남아있는 카드의 수를 저장하는 변수 left\_card와 카드 카운팅을 위한 변수 card\_count 변수를 생성합니다.

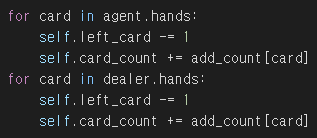
MonteCarlo\_develop의 generate\_epeiosde도 문제의 조건에 맞게 수정했습니다.

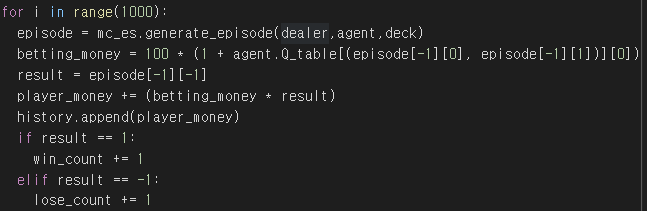
카드 카운팅은 A, J, Q, K의 카드에는 -1의 가중치를 2, 3, 4, 5, 6의 카드에는 1의 가중치를, 7, 8, 9의 카드에는 0의 가중치를 부여했습니다. 이는 덱에 남아있는 카드들을 현재 손에 있는 카드들의 가중치의 합을 이용해 유추할 수 있도록 하기 위한 전략입니다. 가중치 1의 카드들은 딜러에게 불리한 카드들, -1은 딜러에게 유리한 카드들, 0은 공평한 카드를 의미합니다. 덱의 초기화 시, 새로운 52장의 카드들이 추가되더라도 card\_count 변수의 값이 변하지 않게 합이 0이 되게 하기 위해서 1과 -1인 값들의 수를 맞췄습니다.

그리고 dealer\_card란 변수를 통해서 딜러가 공개하는 카드의 수를 저장하는데 기존의 코드에선 dealer.show() 함수가 이후의 while문 안에서 사용하는 방식이었습니다. 이렇게 수정한 이유는 dealer.show() 함수가 random choice를 통해 딜러의 손에 있는 카드 중 한 장을 반환하는데 한 에피소드 내에서 show() 함수가 여러 번 호출되면 반환되는 dealer의 카드가 바뀔 가능성이 있기 때문에 그 카드를 고정했습니다.

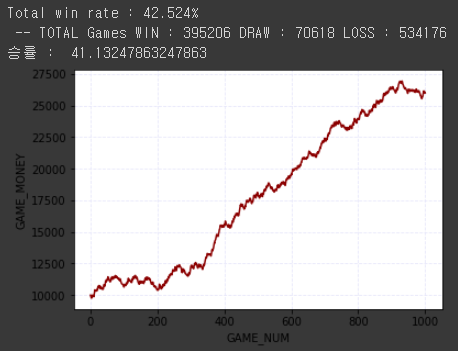


이후의 반복문에서 agent가 policy에 따라 결정을 내리기 전에 agent의 hand와 dealer가 공개한 카드를 이용해 카드 카운팅을 수행해 주고 그 결과를 state에 추가해준 다음 policy를 수행해 줍니다. 이때 기존의 코드는 매 에피소드의 첫번째 decision 상황 일 때는 반드시 무작위 action을 선택하는 방식이었습니다. 하지만 만약 Q 테이블에 존재하지 않았던 state에 대한 무작위 action의 수행은 agent.policy()함수 내에서 이미 구현이 되어있으며 강화학습에서 Q 테이블에서의 비교 없이 무조건적으로 random한 action을 취하는 것은 승률에 도움이 되지 않는다고 판단해 항상 agent의 policy() 함수를 통해 결정을 내렸습니다.

에피소드가 끝이 나면 해당 에피소드에서 사용한 카드들의 정보를 카드 카운팅에 사용한 후 결과를 반환합니다.



다음과 같이 Override된 함수들을 이용해 강화학습을 진행하고 1000번의 게임을 진행합니다. 이때 이전에는 배팅하는 금액이 항상 10으로 고정되어 있어 소지금이 감소할 수밖에 없었지만, 베팅 금액에 변화를 주어 플레이어의 소지금을 늘릴 수 있는 방안을 만들어 보았습니다. Agent의 Q 테이블은 해당 state에 대한 reward의 기대값을 저장하고 있는데 이 기대값을 이용해 플레이어가 승리할 수 있는 가능성을 예상할 수 있습니다. 기대값이 -1~1까지의 범위로 분포해 있기 때문에 (기대값+1)의 값을 고정된 배팅금액에 곱해주어서 승리할 가능성이 높을 땐 높은 금액을, 패배할 가능성이 높을 땐 낮은 금액을 배팅하도록 구현했습니다.

그 결과 학습하는 동안의 최종 승률은 약 43%였으며 승률이 50%를 넘지 않음에도 배팅전략을 통해 플레이어의 소지금을 상승시킬 수 있었습니다.

* **실험 결과 및 고찰**

실험 결과 Monte Carlo Control을 통해 Black Jack 게임 시뮬레이션의 승률을 높일 수 있었습니다. 강화학습이라는 개념 자체도 흥미로웠지만 다른 것 보다 ace의 보유 여부가 결과에 영향을 미친다는 점이 아주 흥미로웠습니다. 그리고 강화학습이 아주 창의적인 아이디어이긴 하지만 아주 많은 경우에 대한 학습을 필요로 하며 state에 대한 정의가 아주 중요한 전략이라 적용할 수 있는 분야가 조금 제한적일 것 같다는 생각을 했습니다. 하지만 state에 대한 정의가 분명하게 내릴 수 있고 충분한 학습을 진행할 수만 있다면 아주 강력한 전략임은 분명하게 확인할 수 있었습니다.