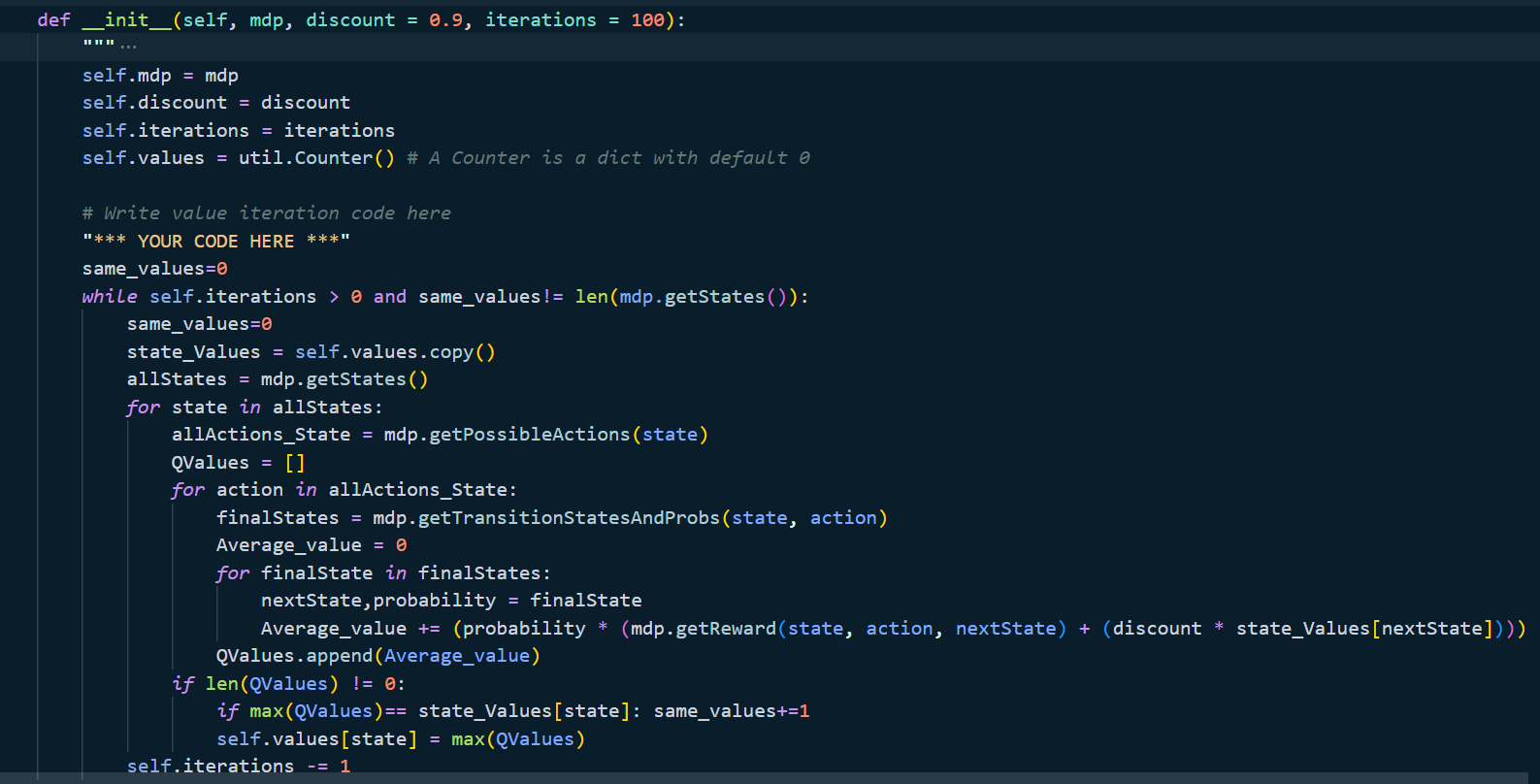
**باسمه تعالی**

**پروژه 3 درس هوش مصنوعی – دکترهوش مصنوعی**

**سید امیرمهدی میرشریفی – 9831105**

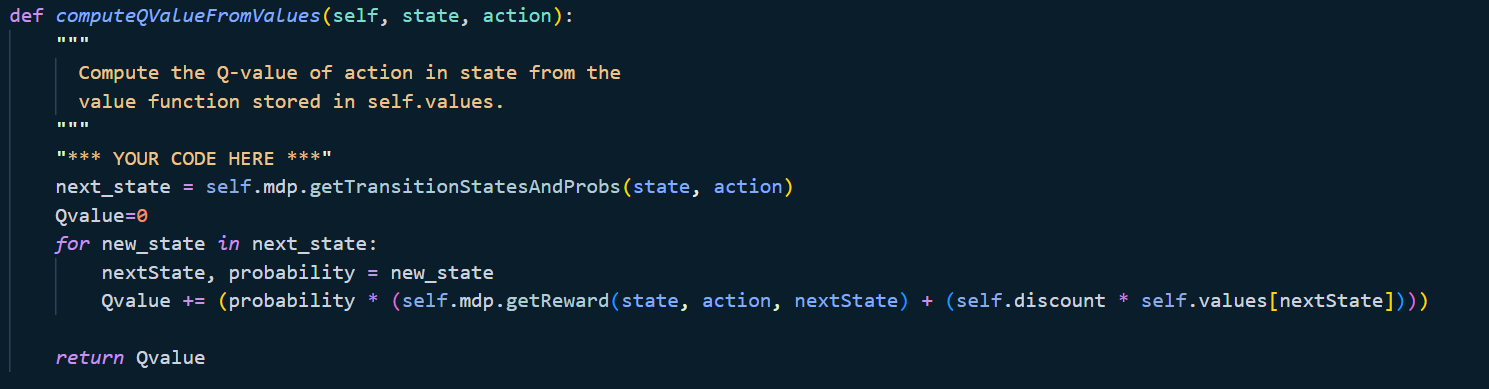
1. **تکرار ارزش**

 در کلاس valueIterationAgent چندین تابع تعریف شده است که هر کدام توضیح داده میشود:

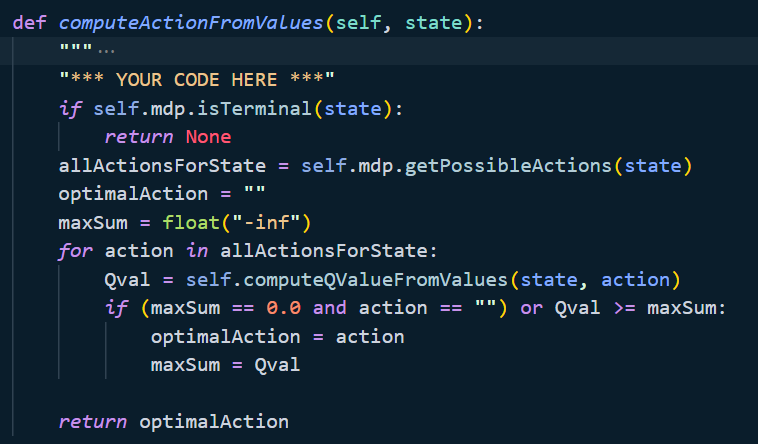
در این تابع که تابع سازنده شی مورد نظر از کلاس هست ، در ابتدا فیلد های شی را تعریف میکنیم. سپس با تعداد دوری که در ورودی جهت تکرار ارزش گرفته ایم ، به همان تعداد ابتدا برای تمام حالت ها اکشن و احتمال آنها را میگیریم ، سپس ماکسیمم Qstate رو برایشان حساب میکنیم و ماکسیمم Q را به عنوان ارزش آن حالت قرار میدهیم. در ابتدای هر دور چک میکنیم اگر ارزش ها همگرا شده بودند دست از چک کردن بر میداریم. مکانیزم چک کردن همگرایی از طریق فیلد same\_value شکل میگیرد که برای هر حالت اگر ارزش جدید با ارزش قبلی برابر باشد یکی به آن اضافه میکند . اگر در ابتدای دور بعد مقدار این متغیر برابر تعداد حالت ها باشد یعنی همگرا شده است . اگر نه که مجدد مقدار ان را صفر خواهیم کرد و یک دور جدید را آغاز میکنیم.

**\*این بخش در سوال 4 تغییری جزئی خواهد کرد.\***

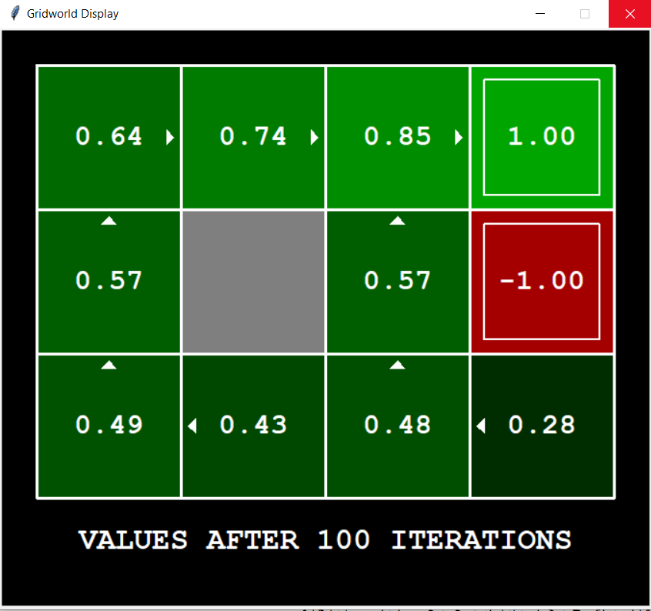
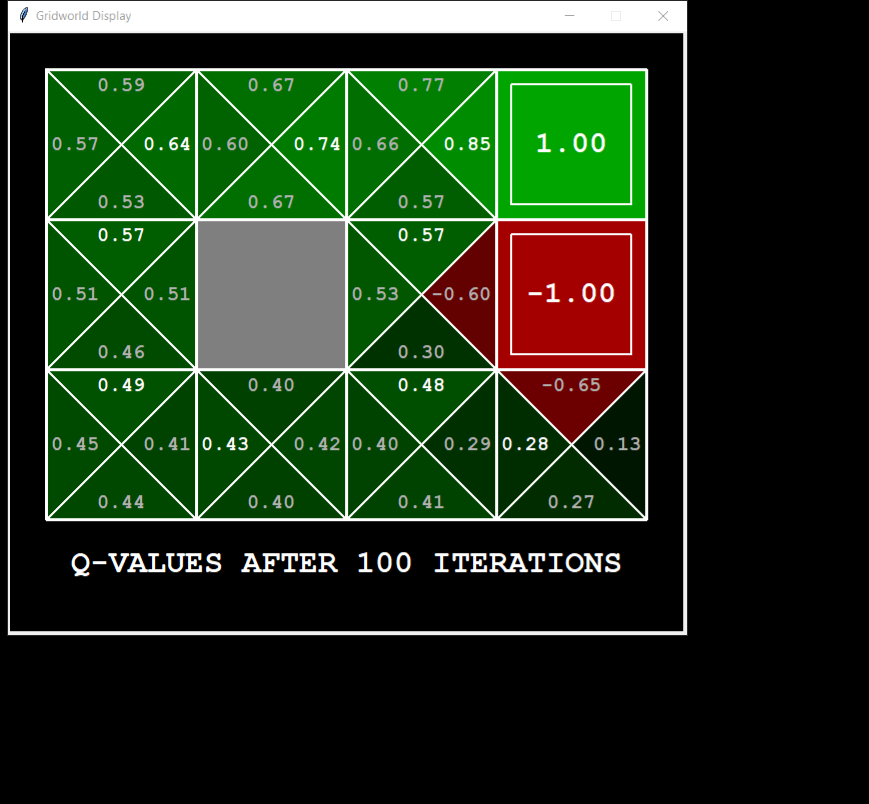
تابع بعدی تابعی است که با گرفتن حالت و اکشن مقدار Q متناسب با آن را حساب خواهد کرد. که این کار با فیلد هایی که در تابع آغازین مانند فیلد ارزش وضعیت ها ، پاداش ها و ... انجام میگیرد.



آخرین تابعی که در این بخش پیاده سازی شده است ، تابعی است که با توجه به وضعیتی که به تابع می دهیم حرکت و پالیسی مد نظر را برمیگرداند که این کار از طریق فراخوانی تابع بالا برای اکشن هایی که توان آن را دارد و حساب ماکسیمم آن و بهترین اکشن انجام میشود.



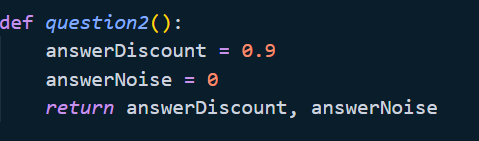
تصاویر نیز حاصل و نتیجه کد هستند:



**2)تجزیه و تحلیل عبور از پل**

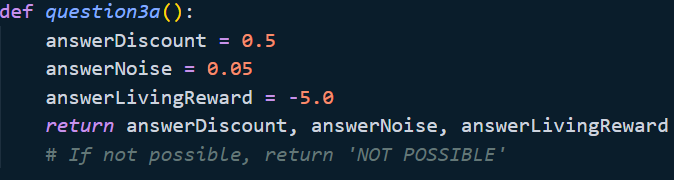
با توجه به این تصویر:

همانطورکه مشاهده میکنید چون حرکت ما نویز دارد بنابراین هر چه در وسط پل هستیم احتمال افتادن ما در آتیش ها خیلی بیشتر میشود و هرچه به لبه ها نزدیک تر هستیم این احتمال کاهش میابد. بنابراین با تنها تغییر یک عامل و آن هم نویز به صفر میتوان به نتیجه زیر دست پیدا کرد:

همچنین کد مربوط به این بخش:

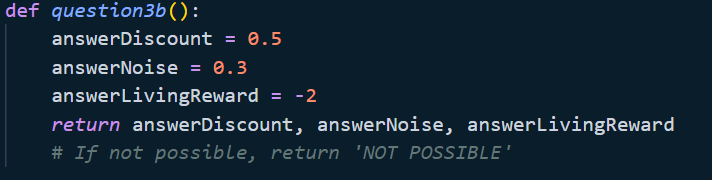
**3) سیاست ها**

- خروجی نزدیک را ترجیح دهد و ریسک صخره را بپذیرد:

در این بخش ما می بایست هزینه زندگی را بالا در نظر بگیریم تا سراغ حالت پر ارزش نرود ، همچنین نویز را بسیار کم در نظر بگیریم تا ریسک صخره را بپذیرد.

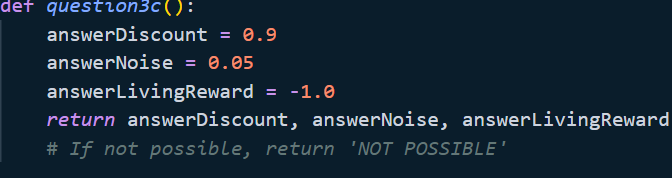
* خروجی نزدیک را ترجیح دهد و از صخره دوری کند:

در این مرحله همچنان باید هزینه زندگی بالا باشد اما نه به بالایی سری قبل تا باعث شود ریسک صخره را با وجود نویز بالا بپذیرد.

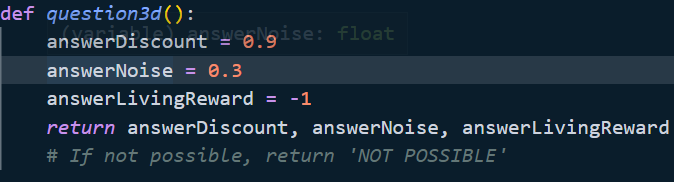


* خروجی دور را ترجیح دهد و ریسک صخره را بپذیرد:

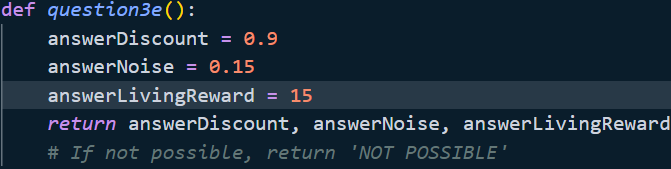
در این مرحله نویز و هزینه پایین را پایین و discount را بالا در نظر میگیریم



* خروجی دور را بپذیرد ولی از صخره دوری کند

در این مرحله نویز را بالا و هزینه زندگی را پایین در نظر میگیریم تا مسیر دور را انتخاب کند و بخاطر نویز از صخره دوری کند

* از هردو صخره اجتناب کند

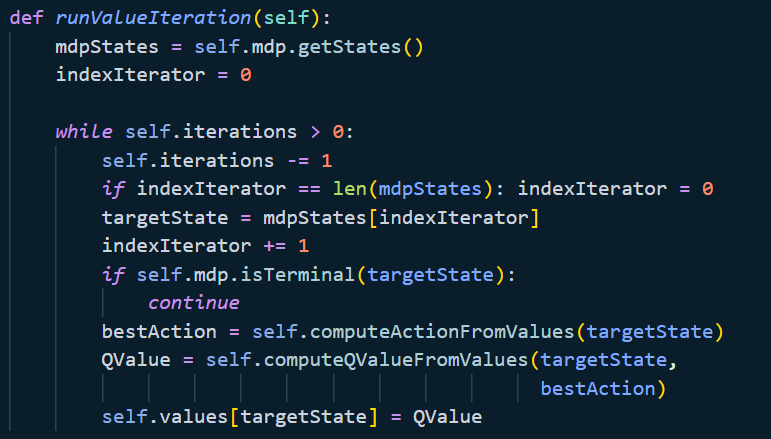
برای این مرحله تنها کاری که لازم است این است که هزینه زندگی را یک عدد مثبت و بیشتر از پر ارزش ترین پاداش در نظر بگیریم .

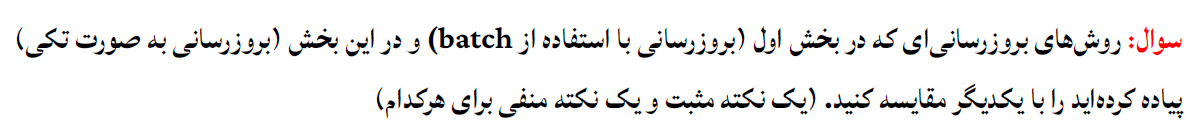
**گزارش: آیا استفاده از الگوریتم تکرار ارزش تحت هر شرایطی به همگرایی می انجامد؟**

بله . این الگوریتم تضمین میکند که همیشه به حالت بهینه همگرا شود.

**4) تکرار ارزش ناهمزمان**

در آخر بخش یک بیان شد که بخشی از کد تابع سازنده کلاس valueIterationAgent تغییر خواهد کرد.

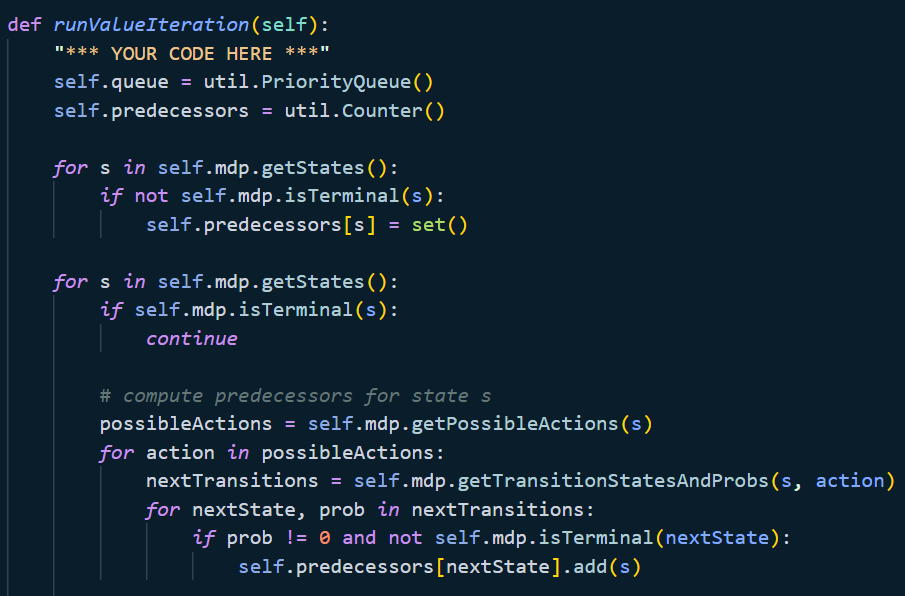
این تغییر در این بخش با انتقال کد valueIteraion به تابع runValueIteraion است که با توجه به این که در کلاس AsynchronousValueIterationAgent هم نیاز به کد valueIteraion اما به مدلی دیگر است ، و همچنین از کلاس valueIteraionAgent ارث بری می کند بنابراین کد هر کدام را در تابع runValueIteraion پیاده سازی کردیم که برای کلاس Asynchronous یک بار بازنویسی یا override صورت خواهد گرفت . تابع runValueIteraion برای کلاس AsynchtonousValueIteraionAgent است. برای کلاس valueIterationAgent نیز همان محتوایی است که در سوال یک آمده است.همچنین تابع runValueIteration در تابع سازنده صدا زده میشوند و متناسب با نوع کلاس عملیات تکرار ارزش انجام میشود.

در این تابع همانطور که خواسته شده اگر حالت ما ترمینال باشد آن را رد میکند . اگر تمام حالت ها چک بشوند به وسیله فیلد indexIterator مجددا از حالت اولیه شروع میشود.

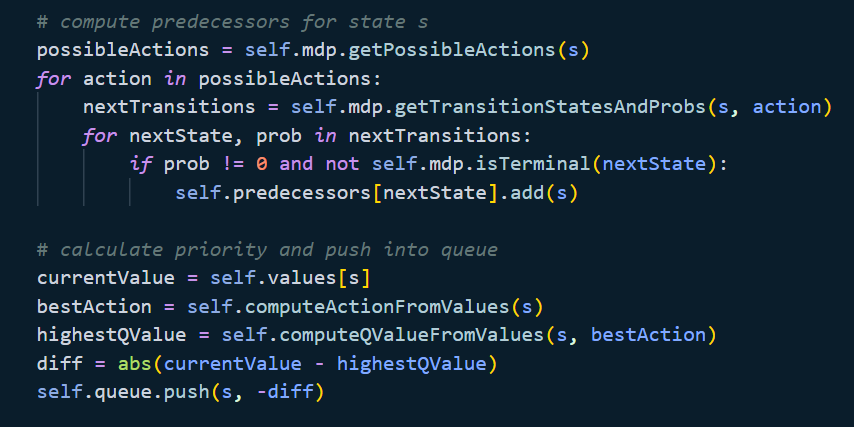
زمانی که ما در هر دور تمام حالات را بررسی میکنیم نیاز به یک آرایه داریم که حالات را درون آن کپی میکنیم اما زمانی که تکی این کار را انجام بدهیم نیاز به این کار نیست. این نکته مثبت. اما نکته منفی آن است که عملا تعداد باری که ما چک کردن را برای تمام حالات انجام میدهیم برابر است با تعداد داده شده تقسیم بر تعداد حالات که در این صورت اگر همگرایی در تکرار بالا به دست بیاید احتمالا در این حالت همگرا نشود.

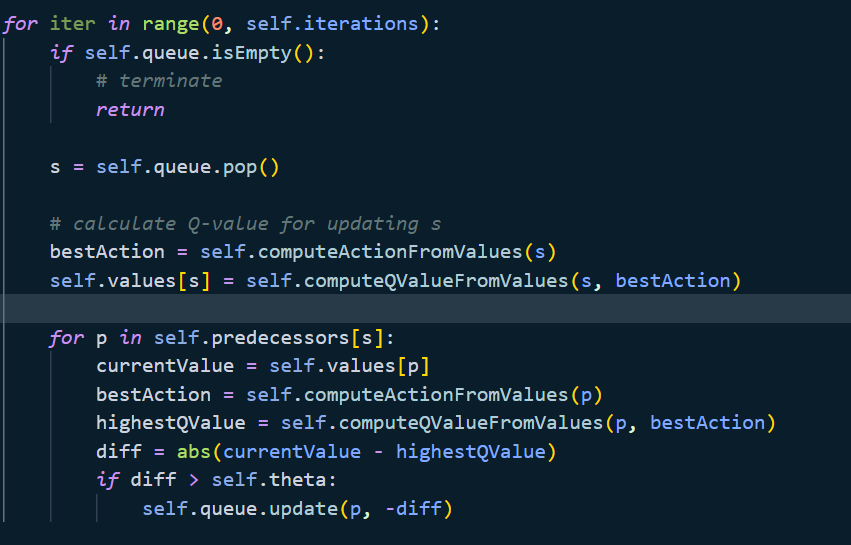
**5)تکرار ارزش اولویت بندی شده**

در این بخش همانند مراحل گفته شده در دستورکار ، ابتدا یک صف جهت تعریف اولویت حالت ها تعریف میکنیم که یک صف اولویت از نوع مین هیپ است. همچنین یک کانتر برای تمام حالت هایی که ترمینال نباشند تشکیل میشود تا حالت های پسینشان در آن نگه داری بشود که در ابتدا برای هر حالت یک مجموعه تهی است.



سپس برای هر حالت ابتدا حالت های پسینشان و در ادامه اولویت آن ها را محاسبه و در صف اولویت پوش میکنیم.

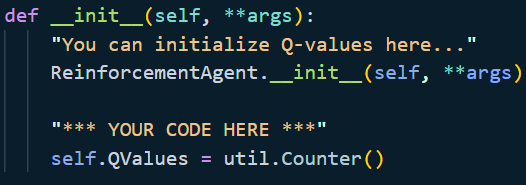


و ادامه کد هم که همان مراحل پیاده سازی گزارشکار:

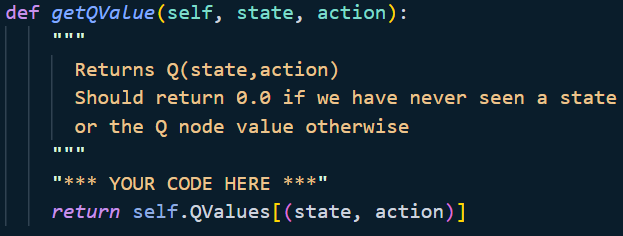
**6)یادگیری**

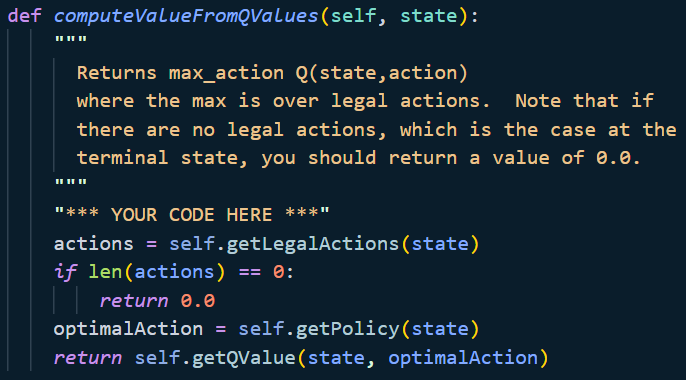
بیشتر توابعی که در ادامه بیان میشوند پیش تر تعریف و توضیح داده شده بودند اما مجددا توضیح آن بیان خواهد شد:

تابع سازنده: در این بخش یک کانتر برای Qvalue ها تعریف میکنیم تا هنگام ساخت شی ساخته بشود.

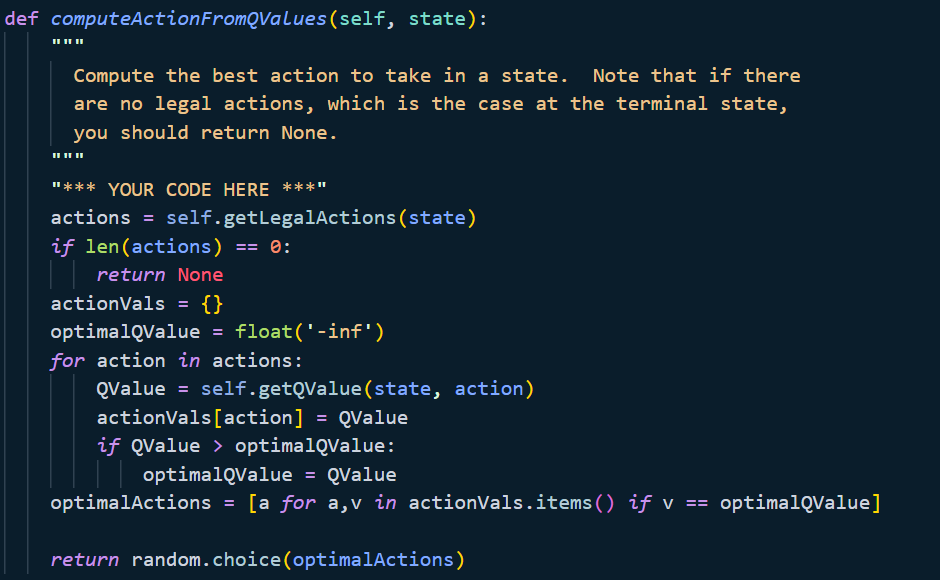


تابع getQvalue : در این تابع به ازای حالت و اکشن کیو ولیو مخصوص آن را بر میگردانیم

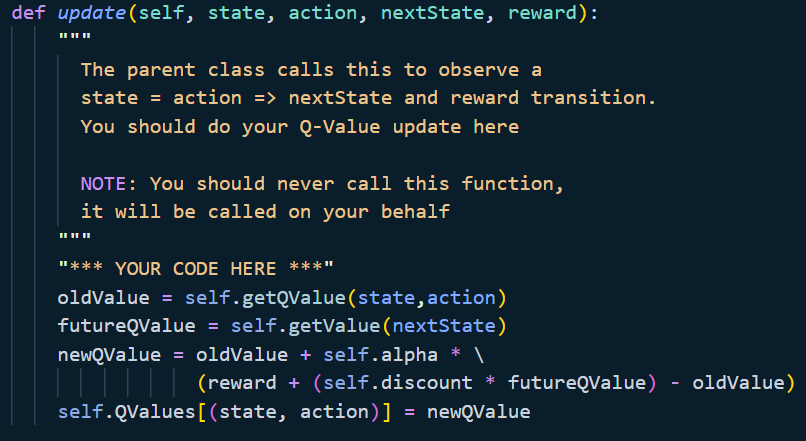


تابع computeValueFromQValues : در این تابع با حالت ورودی ، اکشن بهینه را دریافت و سپس به واسطه آن مقدار کیو ولیو آن را به عنوان ارزش حالت ورودی برمیگردانیم

تابع computeActionFromQValue : در این تابع اکشن بهینه برای حالت ورودی انتخاب میشود.

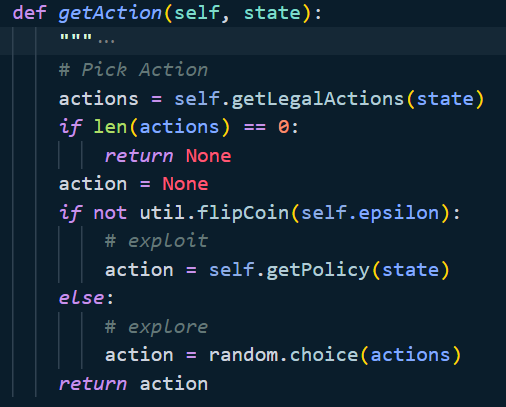


تابع update : همانطور که از نامش پیداست و بر اساس فرمولی که برای یادگیری داشتیم مقدار را متناسب با اکتشافی بودن آن و اهمیت حالات جدید و قدیم به روزرسانی می کند.



**7) اپسیلون حریصانه**

تابع getAction: با گرفتن حالت ، اگر صفر باشد همان اکشن بهینه را استفاده خواهد کرد که این حالت ، حالت بهره برداری و اگر اپسیلون صفر نباشد وارد فاز اکتشافی میشود و اکشن رندوم دریافت خواهد کرد.



**8)بررسی دوباره عبور از پل**

نتیجه یادگیری بدون نویز و 50 آموزش بدین شکل میشود:

و زمانی که اپسیلون را صفر میکنیم:

همانطور که ملاحظه میشود کم و زیاد کردن اپسلون روند کشف حالات جدید را تغییر میدهد بدین معنی که اپسیلون صفر یعنی هیچ گونه اکتشافی نباشد و اپسیلون یک یعنی دائم در حال اکتشاف.

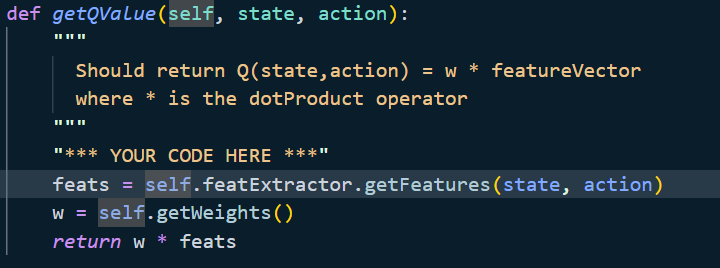
**9)پکمن**

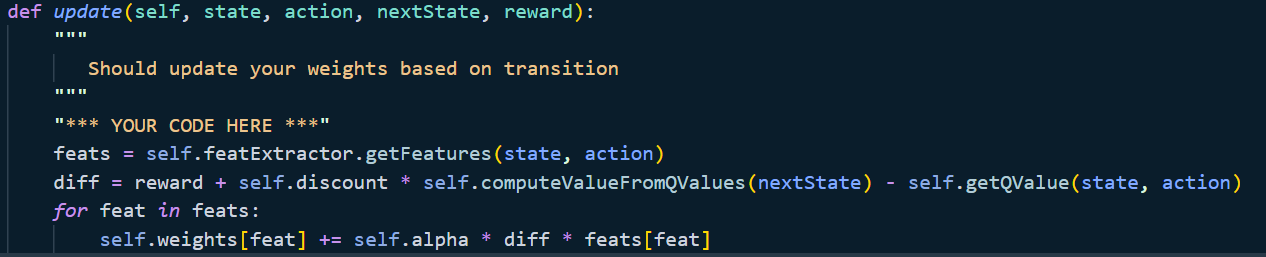
چون در تابع getAction و computeActionFromValue این که حالات دیده نشده هم در نظر گرفته بشوند برای اکشن اعمال شد در این بخش نیاز به کار خاصی نبود و نمره کامل دریافت میشود.

**10)یادگیری تقریبی**

کد این بخش به آسانی و در دو تابع پیاده سازی میشود که به آن می پردازیم:

تابع getQvalue : در این تابع بر اساس وزن ها و فیچر ها کیو ولیو محاسبه میشود:



تابع update : در این بخش هم طبق فرمول مربوطه وزن ها به روز رسانی خواهند شد

در آخر ممنونم از شما تدریسیار گرامی بابت وقتی که گذاشتید و پوزش بابت کم و کاستی ها احتمالی متاثر از کمبود وقت