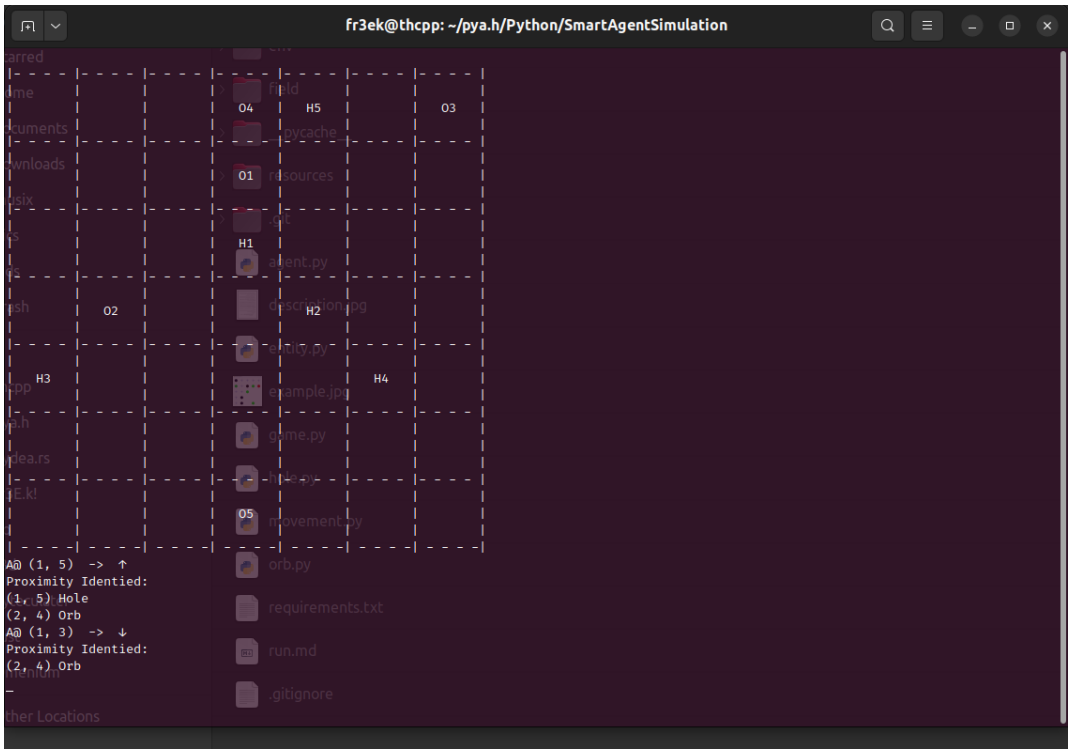
Smart Agent Simulation

ماهیت این پروژه را در واقع می توان این‌گونه در نظر گرفت که می‌خواهیم یک عامل هوشمند پیاده سازی کنیم که تعدادی عناصر از عناصر در مختصات های تصادفی را پیدا کند و آن ها را تا مقاصدی تصادفی – که حفره- نامیده می‌شوند- جا به جا کنیم.

ما این پروژه را به دو صورت پیاده کرده ایم:

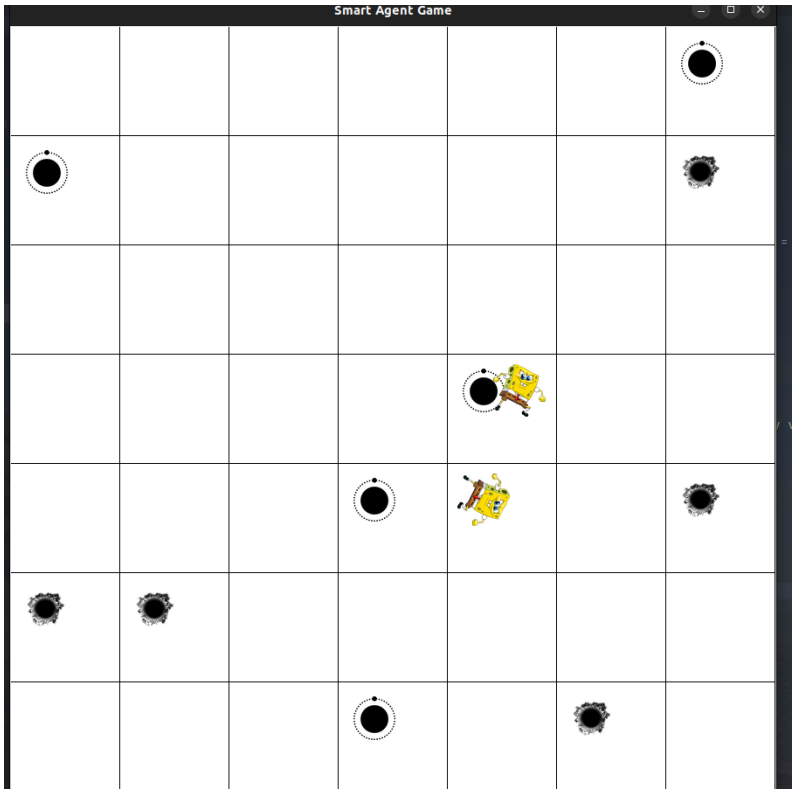
۱- رابط کاربری تحت کنسول (Console App)

در حالت کنسول اپ، ایجنت(های) ما A می باشند که گوی ها را - که با O مشخص می شوند- پیدا کرده و تا حفره ها – که با H مشخص شده اند– می برند.



۲- رابط کاربری گرافیکی (GUI App)

در این حالت، باب اسفنجی (ها) به دنبال گوی هایی که در ابتدا نمی‌داند کجاست، میگردد و هر کدام را که پیدا کرد به داخل حفره هایی که میشناسد منتقل میکند.



نکته: در این مرحله از پروژه فعلا از حالت کنسول ائ برای ارائه استفاده شد. علت این امر عدم اطمینان از پرفورمنس حالت گرافیکی پروژه بود. گرچه برای اجرای پروژه صرفا کافیست خط اول فایل game.py به صورت زیر تبدیل شود و برنامه بصورت خودکار حالت اجرا را به gui app تغییر می‌دهد:

```
from field.gui import Field
```

توضیح کد پروژه

این پروژه کاملا با رعایت اصول شی گرایی نوشته شده است. تمام اشیاء داخل بازی کلاس مخصوص خود را دارند و در هر جا که نیاز بوده است از ارث بری استفاده شده است. کد برنامه تقریبا فقط با استفاده از کتابخانه های

استاندارد پایتون نوشته شده است. فقط در رابط گرافیکی از کتابخانه های خارجی tkinter برای طراحی پنجره گرافیکی و ایتم های آن و Pillow برای بارگذاری عکس ها استفاده شده است.

ماژول movement

این ماژول حاوی دو کلاس بسیار پرکابرد است که در همه جای این پروژه استفاده شده‌اند. اولین کلاس آن کلاس Coordinates است که همان‌طور که از نامش برمی‌آید نماینده مختصات اشیا داخل پروژه است. تمام اشیاء اصلی پروژه –گوی، حفره و عامل هوشمند- داری فیلدهایی مانند position هستند که از جنس این کلاس Coordinates اند. این کلاس دارای دو متغیر(یا همان فیلد در مبحث کلاس) اصلی x و y هست که مشخص کننده مکان فعلی یک شی است. همچنین این کلاس دارای توابع(یا همان متد در حوزه کلاس) می‌باشد که جهت مقایسه مکان ها و نمایش آن ها استفاده می‌شوند.

این کلاس یک تابع Random دارد که که برای تولید مختصات تصادفی در محدوده خواسته شده برای x و y و که با پارامترهای x_max و y_max مشخص شده‌اند استفده می‌شود. تفاوت این تابع با تابع Randomize مهم است. تابع Random یک تابع استاتیک است که یک شی جدید از نوع Coordinates تولید می کند که حاوی مختصات تصادفی ست؛ اما تابع Randomize یک تابع عضو (member method) می باشد که برای تصادفی کردن مختصات شی در حال استفاده می‌باشد و شی جدیدی تولید نمی‌کند.

همچنین در این ماژول یک Enum داریم؛ که البته در زبان پایتون Enum هم از جنس کلاس می‌باشد و برخلاف زبان‌های دیگر یک ماهیت جداگانه نیست. این Enum که با نام Direction تعریف شده است در کلاس Agent استفاده می‌شود و در خرلحظه جهت عامل هوشمند را مشخص می‌کند که با چهار مقدار RIGHT – LEFT – UP –DOWN مشخص میگردد. هم چنین این Enum یک تابع Random دارد که برای تولید جهت تصادفی استفاده می‌شود، که در ابتدای بازی برای تولید جهت تصادفی عامل ها استفاده شده است.

کلاس Entity

این کلاس در فایل entity.py تعریف شده است و بصورت مستقیم استفاده نشده است؛ یعنی بعنوان کلاس پدر برای اشیا داخل بازی استفاده شده است. این کلاس شامل مشخصه ها و رفتارهای مشترک گوی ها، حفره‌ها و عامل هوشمند می‌باشد.

از جمله فیلدهای مشترک اشیا که در این کلاس تعریف شده اند می توان به id و name اشاره کرد که برای تمایز بین اشیا از یک دسته تعریف شده اند. فیلد position که می‌توانم گفت تقریبا مهم‌ترین مشخصه اشیا است نیز در این کلاس تعریف شده است و از طریق ارث بری در تمامی اشیا هم تعریف می شود و بدین تریت جایگاه همه اشیا در زمین بازی مشخص می گردد. همچنین avatars که لیست عکس های درنظر گرفته شده برای اشیا مختلف است. و از توابع این کلاس می توان به GetNextId اشاره کرد که با دریافت لیستی از اشیاء یک دسته، آیدی عنصر بعدی آن را پیدا می‌کند.

همچنین این کلاس یک فیلد(متغیر) بسیار مهم دارد به نام identified. این متغیر در واقع آیدی عامل هوشمندی که آن را شناسایی کرده است را در خود ذخیره می‌کند. درصورتی که مقدار آن صفر باشد به این معناست که فعلا توسط هیچ عاملی شناسایی نشده است.

کلاس Orb

این کلاس در فایل orb.py برای شبیه سازی گوی ها تعریف شده است که از کلاس Entity ارث بری می‌کند؛ بنابراین علاوه بر فیلدها و توابع مشترک موجود در کلاس پدر یک فیلد اضافی به نام hole دارد. این فیلد یا مقداری ندارد (مقدار None در پایتون) و یا مساوی با ابجکت حفره ای که این گوی را دربردارد می‌باشد. درواقع عامل هوشمند از این طریق متوجه می‌شود که آیا این گوی خارج از حفره‌ها هست یا نه. در صورتی که خارج بود آن را بعنوان یک کاندید احتمالی برای جابه‌جایی درنظر می-گیرد؛ البته این درصورتی است که از قبل توسط عامل شناسایی شده باشد.

این شی در حالت کنسول با حرف O و آیدی هر گوی (مانند O۱) و در حالت گرافیکی با عکس orb.png که در پوشه resources قرار دارد مشخص شده است. درواقع فیلد avatar این آبجکت از این عکس استفاده میکند.

توجه: فیلد avatars لیستی از اشیاء از نوع کلاس Avatar می باشد. کلاس Avatar در فایل resources/avatar.py تعریف شده است که حاوی آدرس عکس استفاده شده و سایز آن برای نمایش در حالت گرافیکی است. همچنین یک فیلد canvas_id دارد که وقتی یک شی در پنجره گرافیکی نمایان شد، آیدی المان آن (که مربوط به کتابخانه tkinter است) را ذخیره می کند.

کلاس Hole

این کلاس هم از کلاس Entity ارث می‌برد و همانند کلاس orb.py نوشته شده است با کمی تفاوت جزئی. اول اینکه این کلاس در ماژول hole.py تعریف شده است؛ همچنین به جای فیلد hole در کلاس Orb، این کلاس فیلد orbs را داراست که لیستی از گوی‌هایی که درون این حفره گنجانده شده‌اند را شامل می‌شود. این بخش بصورت پویا نوشته شده است و یک فیلد استاتیک به‌نام Hole.CAPACITY موجود است که نشان می‌دهد هر حفره حداکثر چند گوی را می‌تواند نگه دارد. چون در این برنامه این مقدار یک است پس این لیست حداکثر یک عضو دارد.

همچنین این کلاس یک تابع دارد به‌نام has_room. درواقع این تابع چک می‌کند آیا داخل این حفره جا برای گوی جدید هست یا نه. این امر با چک کردن تعداد اعضای فیلد orbs میسر می‌شود؛ بدین صورت که اگر این تعداد کمتر از CAPACITY – که در پروژه ما یک است- باشد پس جا برای گوی جدید دارد؛ درواقع عامل هوشمند از طریق این تابع می‌تواند تشخیص دهد که آیا این حفره قابل استفاده است یا نه.

ماژول agent

این ماژول شامل دو کلاس است. کلاس اول کلاس Candidate می باشد. این کلاس برای جابه‌جایی بهینه گوی ها به سمت حفره ها استفاده می‌شود. به این صورت که یک عامل هوشمند در هر مرحله بین ایا شناسایی شده اطرافش بررسی می‌کند و نزدیک‌ترین گوی به خودش را به نزدیک‌ترین حفره به آن گوی وصل می‌کند. این کار از طریق ساخت یک آبجکت جدید از نوع Candidate انجام می‌گیرد که این کلاس دو فیلد hole و orb را برای همین منظور داراست.

این کلاس همچنین تابع distance را دارد که فاصله گوی از حفره را مشخص می‌کند؛ و متد drop نیز بررسی می‌کند که اگه حفره مورد نظر خالی است گوی انتخابی را در حفره بیاندازد. این کار از طریق مقداردهی orb.hole با hole انتخابی و اضافه کردن orb انتخابی به لیست hole.orbs انجام می‌گیرد. این توابع فقط زمانی فراخوانی می‌شود که مکان این حفره و گوی دقیقا یکسان شود و یا به نوعی خروجی تابع distance شی Candidate صفر باشد.

کلاس Agent

ماژول مذکور همچنین شامل کلاس Agent است که همان عامل هوشمند می‌باشد. این کلاس هم دوباره از کلاس Entity ارث‌بری می‌کند و فیلدهای مشترک آن‌ها را داراست. یکی از تفاوت‌های این کلاس در این است که لیست avatars آن ۴ عضو دارد. درواقع در این کلاس این فیلد بصورت دیکشنری پایتون (dict) تعریف شده است که به ازای هر Direction یک آواتار دارد و هر شی آواتار یک عکس در جهت مرتبط دارد. سپس یک مشخصه (property) به نام avatar دارد که با توابع جهت کنونی عامل هوشمند آواتار مرتبط را مشخص می‌کند. پس این کلاس یک فیلد direction نیز دارد (همان طور که قبلا درباره Direction enum گفته شده بود.) که جهت حرکت عامل هوشمند درهر لحظه را مشخص می‌کند. فیلد بسیار مهم دیگر این کلاس moves است که به ازای هر تغییر مکان یک عامل هوشمند یکی به مقدار آن اضافه می‌شود.

محدودیت تعداد حرکتی برنامه هم بصورت پویا می‌باشد؛ بدین صورت که یک فیلد Game.MAX_MOVES داریم [کلاس Game بزودی بررسی می‌شود] و تا زمانی که فیلد moves یک عامل کمتر از این مقدار باشد بازی ادامه دارد. فیلد دیگر candidate می‌باشد که از جنس Candidate می‌باشد و کاربرد آن بصورت کامل در بخش قبلی بررسی شده است. چند تابع مهم در این کلاس وجود دارد که باید به تفصیل بررسی گردند:

۱. look_around

این تابع با هدف شناسایی المان‌های اطراف عامل هوشمند تعریف شده‌است. پس از هر حرکتی که ایجنت انجام می‌دهد، این تابع فراخوانی می‌شود، ۸ خانه اطراف عامل و خانه‌ای که درونش هست را بررسی می‌کند؛ درصورتی که حفره یا گوی‌ای در آن‌جا باشد، آن را شناسایی کرده و فیلد identified ان را مساوی به آیدی خودش می‌کند.

۲. find_next_best_displacement

پس از هر بار که گوی‌ای درون یک حفره می‌رود این تابع فراخوانی می‌شود و از میان اشیاء شناسایی شده توسط عامل(های) هوشمند بهتر گوی و حفره (از لحاظ فاصله) را انتخاب می‌کند و به عنوان Candidate برای عامل هوشمند بر می‌گرداند.

۳. direct_into

این تابع بعد از هرباری که یک Candidate به عامل(های) هوشمند معرفی می‌شود، در هر حرکت جهت حرکت را به سمت مکان حفره آن (candidate.hole.position) عامل تغییر می‌دهد(یا حفظ می‌کند) و درصورتی که هنوز به حفره نرسیده باشد مقدار False و اگر عامل دقیقا روی حفره قرار گرفت مقدار True را بر میگرداند. پس اگر این تابع مقدار True برگرداند عامل متوجه می‌شود که باید گوی را رها کند تا در حفره بیفتد.

۴. check_one_directional_moves

این تابع از جمله توابع بهینه سازی حرکت می‌باشد و بیشتر بدرد زمانی می‌خورد که عامل هیچ Candidate ای ندارد و درحال حرکت رندوم در زمین بازی است تا بالاخره حداقل یک حفره و گوی شناسایی کند؛ در این حالت اگر یک عامل به مدت طولانی صرفا در یک جهت حرکت کرده باشد (بخصوص در ضلع های زمین بازی)، این تابع جهت حرکت را به وسط زمین تغییر می‌دهد تا مانع گیرکردن عامل در حلقه حرکتی باطل شود.

۵. move_forward_to

این تابع بلافاصله بعد از اینکه یک Candidate برای عامل پیدا شد فرزاخوانی می‌شود و عامل را دقیقا به سمت محل گوی حرکت می‌دهد تا به مکان گوی برسد و آن را حمل کند.

همچنین این تابع مقدار فیلد moves عامل را بعد از حرکت افزایش می دهد تا برنامه آمار تعداد حرکت عامل را داشته باشد.

۶. move

این تابع وظیفه حرکت اصلی را دارد. چه در حالت حرکت رندوم برای پیدا کردن Candidate و چه زمانی که عامل به مکان حفره پیدا شده منتقل شد این تابع فراخوانی می‌شود و حرکت عامل را در جهت تعیین شده از قبل ادامه می‌دهد. همچنین با فراخوانی توابع check_one_directional_moves برای جلوگیری از حرکت رندوم طولانی در یک جهت و تابع check_agent_position جهت پیشگیری از خارج شدن عامل از زمین بازی (یعنی اینکه مولفه‌های مکان آن در محدوده ۱ الی ۷ باقی بماند.) حرکت عامل را کنترل می‌کند. درصورتی که حرکت بعدی عامل دقیقا مساوی با مکان عامل دیگر باشد این تابع از حرکت باز می‌ماند تا عامل دیگر از آن مکان رد شود. همچنین در این حالت احتمال

دارد که عوامل گیر کنند و در جایگاه خود ثابت بمانند (هر یک منتظر بماند که بعدی حرکت کند)، که اگر این تعداد حرکت نکردن به ۳ بار برسد تابع move با فراخوانی تابع force_move عوامل را از این حالت نجات می‌دهد.

همچنین این تابع مقدار فیلد moves عامل را بعد از حرکت افزایش می دهد تا برنامه آمار تعداد حرکت عامل را داشته باشد.

پوشه field

این پوشه شامل سه ماژول می باشد. ماژول های gui.py و console.py همانطور که از نامشان بر می‌آید مربوط به رابط کاربری برنامه می‌باشند. این توابع نکات خاص آنچنانی ندارند هر کدام از این ماژول‌ها نهایتا یک کلاس Field معرفی می‌کنند که از کلاس FieldLogic ارث‌بری می‌کنند. کلاس FieldLogic که در ادامه بررسی می‌شود حاوی منطق و جزییات و تمام اطلاعات مربوط به زمین بازی است که این دو ماژول با استفاده از آن بعنوان کلاس پدر، هرکدام به شیوه اجزا آن را بصورت تحت کنسول یا گرافیکی نمایش می‌دهند. از آن‌جا که منطق این دو ماژول کاملا واضح است به همین میزان توضیح کفایت می‌کنیم.

FieldLogic

این کلاس که در ماژول logic.py تعریف شده است حاوی اطلاعات مربوط به زمین و اجزا آن است. اطلاعاتی از قبیل طول و عرض زمین، آبجکت های مربوط به گوی و حفره هاو ...

این کلاس از طریق لیست orbs اطلاعات تمام گوی‌های داخل زمین را نگه می‌داد؛ و از طریق holes همین کار را برای حفره ها می‌کند. همچنین این کلاس یک لیست سه بعدی به نام cells دارد که با توجه به Position های هر کدام از اشیاء هر کدام از آن ها در اندیس متناظر به مکانشان در این لیست قرار می‌گیرند.

توابع این کلاس بسیار ساده اند و نیازی به توضیح مفصل ندارند. این کلاس شامل توابعی جهت افزودن گوی جدید به زمین، افزودن حفره جدید به زمین، افزودن تصادفی این اشیا به زمین، دریافت اشیا داخل یک سلول (عضوی از

لیست cells)، قرار دادن یا جابه‌جا کردن یک شی در یک سلول زمین بازی و ... می‌باشد. ولی در زیر به مهم ترین توابع این کلاس اشاره می‌شود:

۱. update_ui

این تابع در داخل خود FieldLogic تعریف نشده است و صرفا بصورت abstract تعریف شده است. درواقع این تابع توسط کلاس Field در هرکدام از ماژول‌های console.py و gui.py بصورت کامل تعریف می‌شود و وظیفه‌اش این است که اشیاء داخل فیلد cells را با ترتیب همان مکانشان در جایگاه مناسب داخل زمین، حال بصورت گرافیکی یا تحت کنسول رسم کند.

۲. get_remaining_orbs

اگرچه این تابع کد بسیار کوتاهی دارد اما بسیار وظیفه‌ی مهمی دارد. این تابع تعداد گوی‌های در دسترس را بدست می‌اورد؛ یعنی گوی‌هایی که هنوز وارد هیچ حفره‌ای نشده‌اند. اهمیت این تابع در این است که برنامه در هر مرحله با فراخوانی این تابع بررسی می‌کند که آیا عامل (ها) برنده شده‌اند یا نه.

۳. shake

این تابع هر باری که یک گوی درون یک حفره می‌افتد فراخوانی می‌شود. وظیفه این تابع این است که یک عدد تصادفی از ۰ تا ۱۰۰ تولید کند و اگر این مقدار کمتر مساوی ۱۰ بود، یک گوی را یک خانه جابه‌جا کند. در واقع این همان بخشی از سوال است که می‌خواست وقتی گوی داخل حفره رفت باقی گوی ها با احتمال ۱۰٪ جابه‌جا شوند.

فرایند این تابع به این صورت است که به ازای هر گوی خارج از حفره‌ها، با تولید عدد تصادفی بین ۰ تا ۱۰۰ همان احتمال ۱۰ درصدی را شبیه سازی می‌کند و طبق این احتمال یک گوی را (اگر گوی احتمال لازم را داشت) بصورت تصادفی در یکی از چهار جهت به یک خانه کناری جابه‌جا کند؛ البته بدیهی است که در این روند این تابع بررسی می‌کند که مکان جدید استاندارد باشد و خارج از زمین یا در خانه یک گوی دیگر نباشد.

Game

و اما کلاس گیم، در واقع Entry Point یا همان نقطه شروع برنامه‌است؛ این تابع تمام مفاهیم و امکاناتی که تا اینجا مطرح کردیم را با هم بصورت مناسب و همگون ترکیب می‌کند تا بازی به همان صورتی که انتظار می‌رود اجرا شود. این کلاس وظیفه ساختن اشیا Field و Agent و قرار دادن اشیا گوی و حفره –همگی با مکان تصادفی- در field را دارد. همچنین این کلاس به صورت پویا مقدار طول و عرض زمین بازی را دریافت می‌کند که درآینده اقدام برای افزایش یا کاهش ابعاد زمین بازی به راحتی قابل انجام باشد.

همچنین در ماژول game.py در کد بخش main -که در پایتون همان کدی است که در زیر شرط:

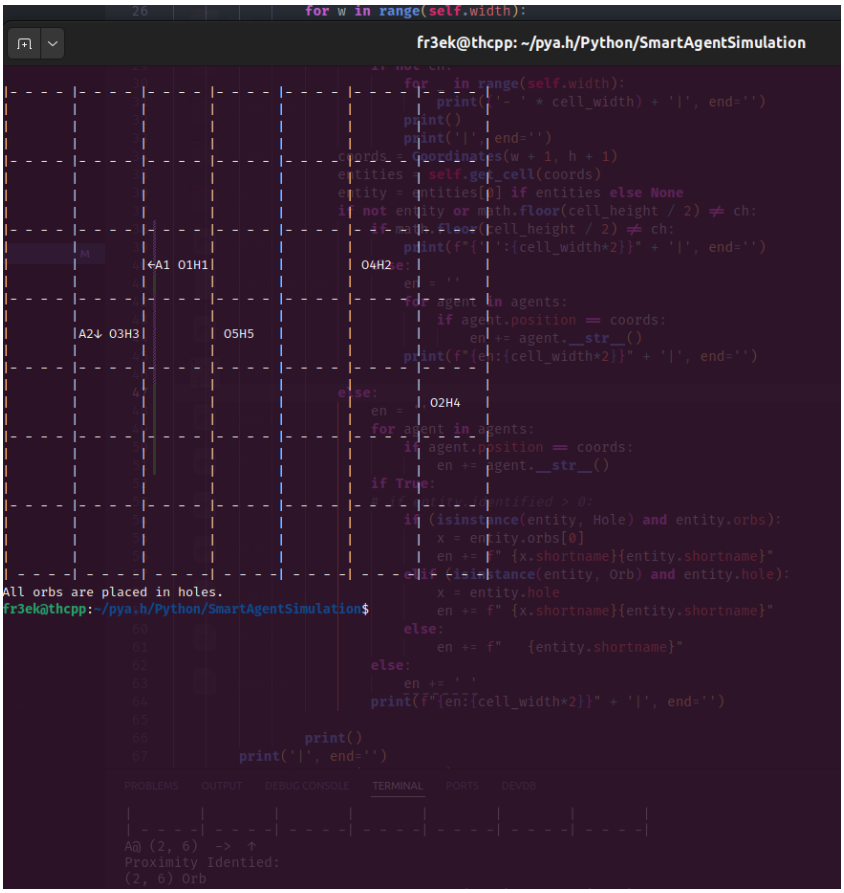
If __name__ == '__main__':

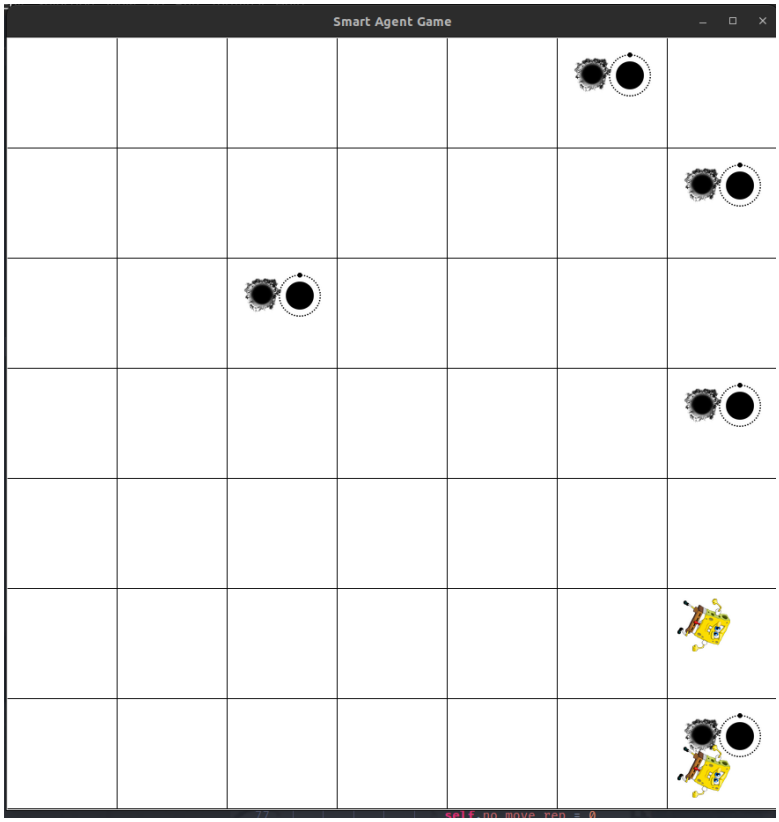
نوشته می‌شود- با ساختن شی‌ای از این کلاس Game و فراخوانی تابع simulate شبیه‌سازی و بازی را شروع می‌کند.

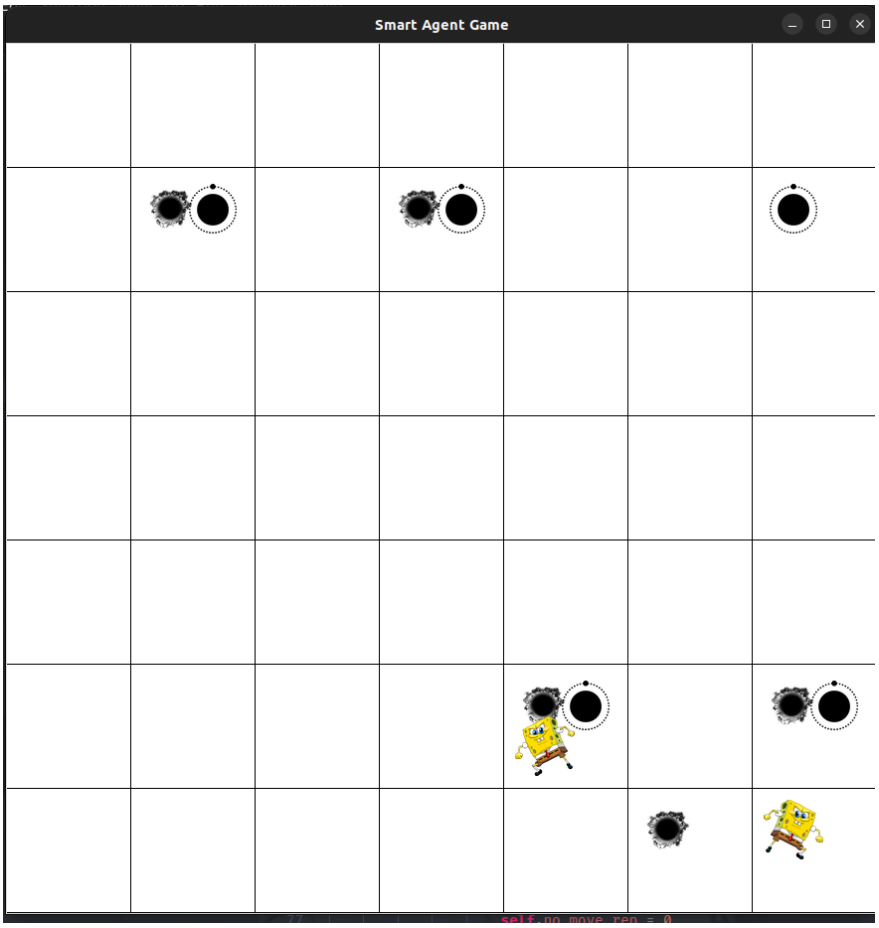
تابع simulate با توجه به نوع زمین بازی –گرافیکی یا تحت کنسول- اقدامات لازم را انجام می‌دهد که در میان این اقدامات فراخوانی تابع do_next_move مهم ترین بخش است. این تابع وظیفه بررسی حرکت عامل‌ها، بررسی به پایان رسیدن بازی یا برنده‌شدن عامل‌ها، و همچنین حرکت این عامل ها با توجه توابع موجود در کلاس Agent را دارد. بصورت خلاصه این تابع در هر مرحله برای هر عامل تابعه look_around را فراخوانی می‌کند تا اشیا اطراف شناسایی شوند؛ سپس اگر عامل (ها) Candidate ای نداشته باشند، با فراخوانی توابع مناسب آن را پیدا می‌کند و از طریق move_forward_to عامل را به گوی موردنظر می‌رساند.

درهرمرحله نیز اگر Candidate ای از قبل موجود بود این تابع با فراخوانی وابع direct_into و move عامل(ها) آن را به سمت هدف خود حرکت می‌دهد.

و اینگونه تمامی بخش ها و اجزا پروژه در جای خود اجرا می‌شوند. نمونه ای از خروجی نهای بازی:


fr3ek@thcpp: ~/pya.h/Python/SmartAgentSimulation
|←A1 O1H1|
O4H2
|A2↓ O3H3|
O5H5
O2H4
All orbs are placed in holes.
fr3ek@thcpp:~/pya.h/Python/SmartAgentSimulation$
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
DEVDB
A@ (2, 6) -> ↑
Proximity Identied:
(2, 6) Orb


Smart Agent Game


Smart Agent Game
self.no_move_rep = 0