Lab Guide 01

In the lab sessions, we will cover the basic concepts of the development of autonomous agents and multi-agent systems. In this guide, you will learn how to set up a python virtual environment, load a set of multi-agent domains, and run/visualize experiments comparing different agents.

1. Setting Up a Development Environment

Throughout the course you will be developing code in Python 3. We will cover the setup of virtual environments using the terminal, but feel free to use any Integrated Development Environment (IDE) you may be familiar with (as it will help a lot with debugging your code).

Our guide covers mainly installations on Linux. All code was tested on Python 3.7.5 and Python 3.10, but we are fairly confident that any version above 3.7 should suffice (let us know if you run into any issues). At the end of the section, we add some extra steps necessary to install within a Windows Prompt.

1.1. Verify Python Version 3

First and foremost, make sure you have Python 3 installed in your computer. You can do so by booting up a terminal and running python:

```
$ python --version
Python 3.7.5
```

Or alternatively,

```
$ python3 --version
Python 3.7.5
```

1.2 Cloning the AASMA's Code Repository

After verifying you have Python installed, go ahead and clone our main repository.

This can either be done on a terminal (if you have a git client installed) or by downloading this zip file directly and extracting it into your working directory.

```
$ git clone https://github.com/GAIPS/aasma-spring-22.git
Cloning into 'aasma-spring-22...
remote: Enumerating objects: 84, done.
remote: Counting objects: 100% (84/84), done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 84 (delta 44), reused 73 (delta 35), pack-reused 0
Unpacking objects: 100% (84/84), done.
$ cd aasma-spring-22/
aasma-spring-22$
```

1.3 Creating a Virtual Environment

You will now learn how to create and activate a Python virtual environment. Virtual environments are a great way for you to sandbox Python projects and freely install dependencies isolating your system's base Python installation. This command creates a virtual environment called *venv*:

```
aasma-spring-22$ python3 -m venv venv
aasma-spring-22$ source venv/bin/activate
(venv) aasma-spring-22$ which python
/aasma-spring-22/venv/bin/python
```

Beware that the virtual environment's name cannot conflict with any other of the project's folders.

1.4 Installing the Requirements

After creating the virtual environment, you may go ahead and install all dependencies. The known dependencies are:

- numpy
- ma-gym
- pygame
- matplotlib
- scipy

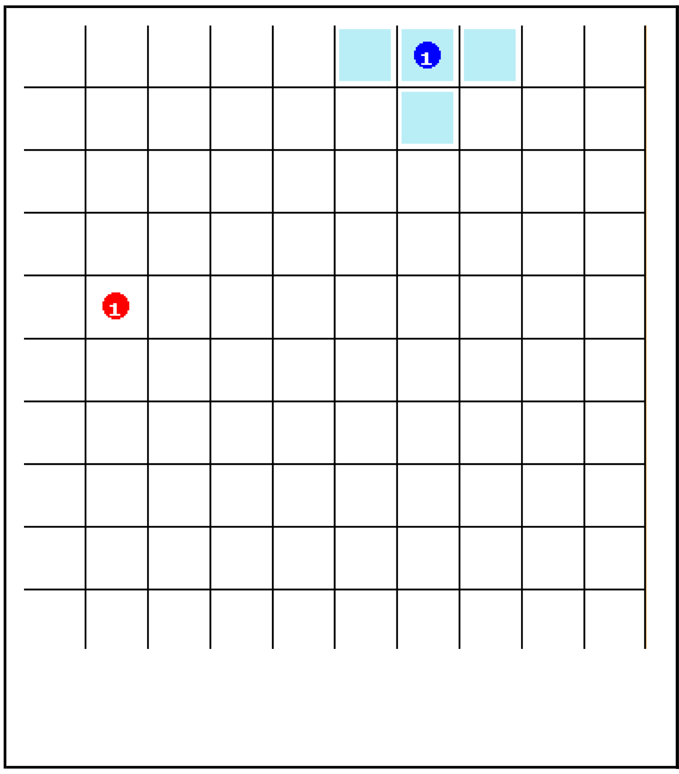
To install the dependencies, you should run the pip package manager and installer for python libraries.

```
(venv) aasma-spring-22$ pip install -r requirements.txt
Collecting numpy
  Downloading numpy-1.22.3-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (16.8
MB)
       |████████████████████████████████| 16.8 MB 12.7
...
many more packages later
...
  Downloading packaging-21.3-py3-none-any.whl (40 kB)
       |████████████████████████████████| 40 kB 6.8 MB/s
Using legacy 'setup.py install' for gym, since package 'wheel' is not installed.
Using legacy 'setup.py install' for future, since package 'wheel' is not installed.
Installing collected packages: six, pyparsing, numpy, future, cloudpickle, scipy,
python-dateutil, pyglet, pillow, packaging, kiwisolver, gym, fonttools, cycler, pygame,
matplotlib, ma-gym
       Running setup.py install for future ... done
       Running setup.py install for gym ... done
Successfully installed cloudpickle-1.6.0 cycler-0.11.0 fonttools-4.33.3 future-0.18.2
gym-0.19.0 kiwisolver-1.4.2 ma-gym-0.0.8 matplotlib-3.5.2 numpy-1.22.3 packaging-21.3
pillow-9.1.0 pygame-2.1.2 pyglet-1.5.0 pyparsing-3.0.9 python-dateutil-2.8.2 scipy-1.8.0
six-1.16.0
```

1.5 Testing the Installation

Finally, test the installation by running *python test_install.py*. These scripts should launch the Predator-Prey environment (explained in Section 2). A listing with the time steps and the print should appear along with an image.

NOTE: If you run into any issues with OpenGL, try degrading pyglet (a dependency installed by ma_gym) by running *pip install pyglet==1.5.27*



```
Timestep 1
        Observation: [6 7 2 7]
        Action: LEFT

Timestep 2
        Observation: [5 7 2 7]
        Action: LEFT

Timestep 3
        Observation: [4 7 1 7]
        Action: LEFT

Timestep 4
        Observation: [3 7 1 6]
        Action: LEFT

Timestep 5
        Observation: [2 7 1 5]
        Action: UP
```

1.7 The Windows Installation

The windows installation roughly follows the same steps above. However, in order to create a virtual environment (Section 1.3) from the Windows Powershell, the command reported to work is:

```
> python -m venv env
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

To activate the virtual environment.

```
><env>\Scripts\Activate.ps1
```

Or alternatively:

```
.\venv\Scripts\activate
```

2. Autonomous Agents and Multiagent Systems

At the core of autonomous agents systems is the interaction between two entities: the environment and the agent(s). The interaction happens sequentially through time in a closed loop fashion, where the agent perceives/sees an **observation** from the environment**,** makes a decision, and executes an **action** within the environment. The environment transitions to another state, enabling the agent to perceive another observation along with a **reward** (i.e., a numeric signal). For this lab, however, we will not be using the reward signals (this will be useful when we implement reinforcement learning techniques).

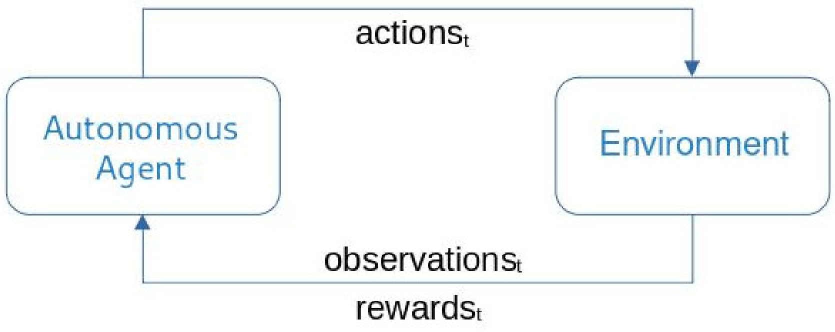


Fig. 1: The interaction between the environment and the agent happens sequentially. At each time step, the agent perceives an observation and a reward. The agent in turn makes a decision and executes an action within the environment. The environment reacts by transitioning to another state.

In the next subsections, we will specify these three important characteristics of autonomous agent and multi agents systems: the environment, the agents, and the information flow between them.

2.1. The Predator-Prey Environment

The Predator-Prey environment in **Fig. 2** showcases a scenario where 2 predators (blue agents, N=2) must cooperate in order to catch 1 moving prey (red agents, M=1). In this scenario, the objective or task is to reach a configuration where a specific number of predators are adjacent to the prey. When that happens, the prey is said to have been captured and the episode terminates. Otherwise, if the predators are not able to capture the prey before a given number of steps, the prey is able to flee and the episode terminates. Recall that a task that has a terminal state is considered to be episodic. The scenario also assumes that the first state is initialized randomly with the additional condition that the initial prey's location cannot be adjacent to any predator. In addition, for all agents (both predator and prey), we consider 5 possible actions --- *Down*, *Left*, *Up*, *Right*, *Noop.*

In this lab, we will be working with a simplified version of the original environment, with two modifications:

- We consider only a single moving prey as part of the environment and will henceforth not consider it as an agent that we can control.
- We assume that predators are able to fully observe each other and the prey.

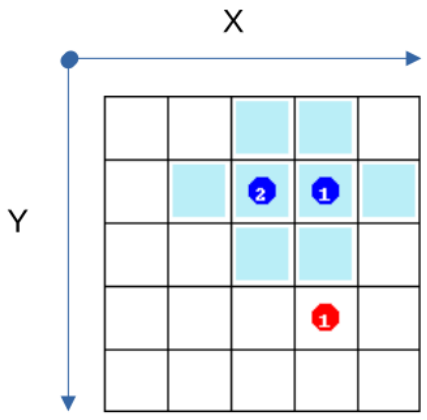


Fig. 2: The predator prey environment where two predators must catch the prey. In this lab session, we will implement predator agents. The light blue squares represent the region where the predator agent may catch the prey.

Finally, the environment offers the following methods to enable the interaction and information flow:

- **reset**: Resets the state of the environment (in the case of Predator-Prey, all positions are chosen randomly) and returns the corresponding observation.
- **step**: Simulates the interaction of the agents with the environment. Hence, it receives actions - a list of integers, where each integer represents an action of an agent - and executes them within the current state of the environment. The environment then transitions to the next state and the agents receive their **observation** and **reward**. Additionally, it returns a boolean **terminal**, that indicates that the episode has ended, and **info** with extra information.
- **close:** Destroys the graphic interface from memory.

The interaction between the agent and the environment endows the agent with the ability to, at least partially, affect the dynamics of the environment. This section presents the three pieces of structured information that allow this interaction to happen:

observation: At each time step, the agents observes x^{α}, y^{α} coordinates for the ɑ-predator agent and x^{p}, y^{p} coordinates for the P-prey agent. For a single predator game, one possible observation is (x^{α}_{1}, y^{α}_{1}, x^{p}_{1} y^{p}_{1}) = [1, 3, 2, 4] , where x^{α}_{1} is the number of columns from the left to right and y^{α}_{1} is the number of rows from top to bottom. And x^{p}_{1} y^{p}_{1} are the coordinates for the prey – they are always the last two coordinates of the observation array. The observation depicted in **Fig. 2** is:

(x^{α}_{1}, y^{α}_{1}, x^{α}_{2}, y^{α}_{2}, x^{p}_{1} y^{p}_{1}) = [3, 1, 2, 1, 3, 3].

Coordinates are zero based, and the observations always go from predator agent 1, 2, .., N and the prey.

rewards: Even though you do not have to use the rewards in this lab session, since they are usually used for reinforcement learning agents, the agent collects a reward of +4.99 when the prey is captured and a reward of -0.01 in all other time steps as a movement penalty.

actions: An autonomous agent has to select one of the following actions : *Down*, *Left*, *Up* , *Right*, *Noop*. These actions are each represented by an integer, where *Down* → *0*, *Left* → *1*, *Up* → *2* , *Right* → *3*, Noop → 4. For multiagent tasks, the environment receives a list of integers each representing the actions of each individual predator.

2.2. Agent

We have gone ahead and created an abstract class called Agent (aasma.agent)

```
class Agent(ABC):

        """
        Base agent class.
        Represents the concept of an autonomous agent.

        Attributes
        ----------
        name: str
            Name for identification purposes.
        observation: np.ndarray
            The most recent observation of the environment

        Methods
        -------
        see(observation)
            Collects an observation from the environment

        action(): int
            Abstract method.
            Returns an action, represented by an integer
            May take into account the observation (numpy.ndarray)

        References
        ----------
        ..[1] Michael Wooldridge "An Introduction to MultiAgent Systems - Second Edition",
          John Wiley & Sons, p 44.
        """

        def __init__(self, name: str):
            self.name = name
            self.observation = None

        def see(self, observation: np.ndarray):
            self.observation = observation

        @abstractmethod
        def action(self) -> int:
            raise NotImplementedError()
```

The primary role of the agent is to perform actions in the environment. The methods provided by the Agent class are:

- **see**: The method that receives an **observation** from the environment and returns nothing.
- **action:** This method should implement a strategy for the agent to select an action.

Although an autonomous agent might implement many more private methods, the ones presented herein are the public or external methods. Perhaps the most naive implementation consists of a random agent that is oblivious to the current observation and selects actions randomly. An alternative that makes better use of the information is the reactive agent that makes decisions based only on the current observation.

3. Exercises

In this section, we finally exercise the concepts introduced in the previous section. Namely, in Exercise 1, we implement an autonomous agent that selects its actions randomly and interacts with the environment. In exercise 2, we implement a reactive agent that makes better use of observations to make better decisions, which we call the greedy agent. In both exercises, we consider the task where one predator agent tries to capture the prey.

In Exercise 3, we extend the single predator agent system into a multiagent system with four predator agents. The task changes as it takes two agents to catch the prey. We compare the performance of three teams: the team formed from predator agents in Exercise 1; the team formed from the greedy agents from Exercise 2; and, the team that has the two types of agents (from Exercise 1 and 2).

3.1 Exercise 1: Single Agent Interaction & Random Agents

Open the file *exercise_1_single_random_agent.py*. The file contains the necessary skeleton code to implement the agent environment interactions loop. In this code, you should find the *main* section, a *run_single_agent* function, and a *RandomAgent* class.

```
if __name__ == '__main__':

         parser = argparse.ArgumentParser()
         parser.add_argument("--episodes", type=int, default=30)
         opt = parser.parse_args()

         # 1 - Setup environment
         environment = SimplifiedPredatorPrey(
                  grid_shape=(7, 7),
                  n_agents=1, n_preys=1,
                  max_steps=100, required_captors=1
         )
         environment = SingleAgentWrapper(environment, agent_id=0)

         # 2 - Setup agent
         agent = RandomAgent(environment.action_space.n)

         # 3 - Evaluate agent
         results = {
                  agent.name: run_single_agent(environment, agent, opt.episodes)
         }

         # 4 - Compare results
         compare_results(results, title="Random Agent on 'Predator Prey' Environment", colors=["orange"])
```

The *main* section of the code consists of 4 steps: setup of the environment, set up of the agent, code to run an evaluation loop, and code to run a comparative analysis.

3.1.1 Implement RandomAgent Class

Your first task is to implement the constructor (*__init__)* and *action* method, as shown below:

```
class RandomAgent(Agent):

        def __init__(self, n_actions: int):
            super(RandomAgent, self).__init__("Random Agent")
            # TODO
            raise NotImplementedError()

        def action(self) -> int:
            # TODO
            raise NotImplementedError()
```

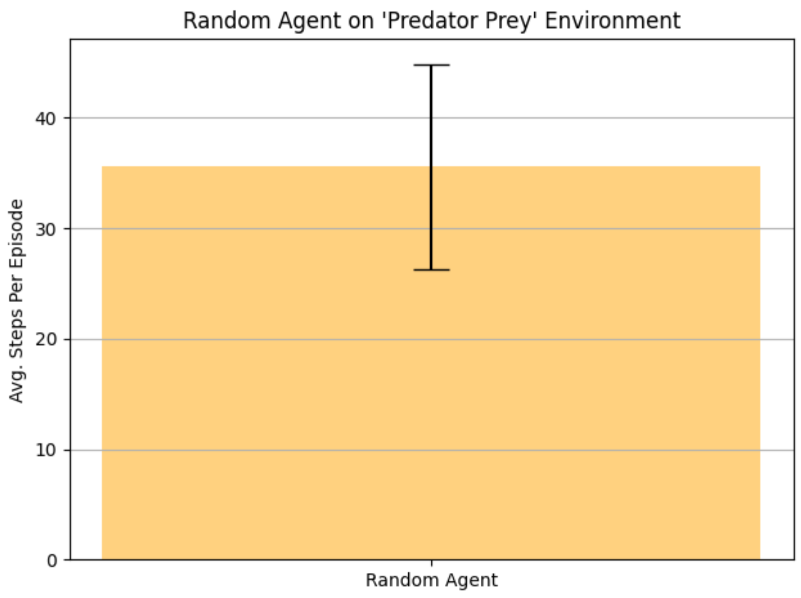
Your agent must select an action from the set **{**0: *Down*, 1: *Left*, 2: *Up* , 3: *Right*, 4: *Noop*} using a uniform distribution. TIPS: use numpy.random library and check *test_install.py*

3.1.2 Complete the Agent Environment Loop

Also, add code in the *while* loop so that the *run_single_agent* function can implement the interaction of the agent with the environment. TIP: See **Section 2.1**.

```
def run_single_agent(environment: Env, agent: Agent, n_episodes: int) -> np.ndarray:

        results = np.zeros(n_episodes)

        for episode in range(n_episodes):

                steps = 0
                terminal = False
                observation = environment.reset()

                while not terminal:
                        steps += 1
                        # TODO - Main Loop (4 lines of code)
                        raise NotImplementedError()

                environment.close()

                results[episode] = steps

        return results
```

After running the code, you should be able to see the following plot, which depicts the random agent's average number of time steps to capture the prey (with a 95% confidence interval):



Now try to answer the following questions:

1. On average, how many time steps does it take for the episode to terminate?

   30 something steps

2. Can the agent catch the prey in 20 time steps or less?

   No. Even with the confidence level of 95%, considering the lowest average of time steps per episode, the agent doesn't catch the prey in less than 25 time steps.

3. A colleague watched an episode terminate at 100 time steps. Did the prey flee or get captured?

   The prey fled (100 is the default number of time steps)

4. How many episodes terminate with the prey alive? Hint: Check the **info** dictionary returned by the step method.

   Around 3/4

Bonus question:

1. The scenario above assumes that the prey moves randomly within the environment. Consider a different scenario where the prey always: (a) moves first and (b) tries to move away from the predator. Can the predator still capture the prey?

   Yes: imagine if the prey spawns in a corner and the predator in the closest position possible according to the rules - even if the prey moves first, the predator will catch it.

   If the border is infinite or if he border is sufficiently bigger than the limit of time steps, the predator will always catch the prey (it will always corner it)

3.2 Exercise 2: Reactive Agents vs Random Agents

Open the file *exercise_2_single_random_vs_greedy.py*. The file contains a *main* section with the same structure as *exercise_1_single_random.py*, but with a GreedyAgent class, as shown below.

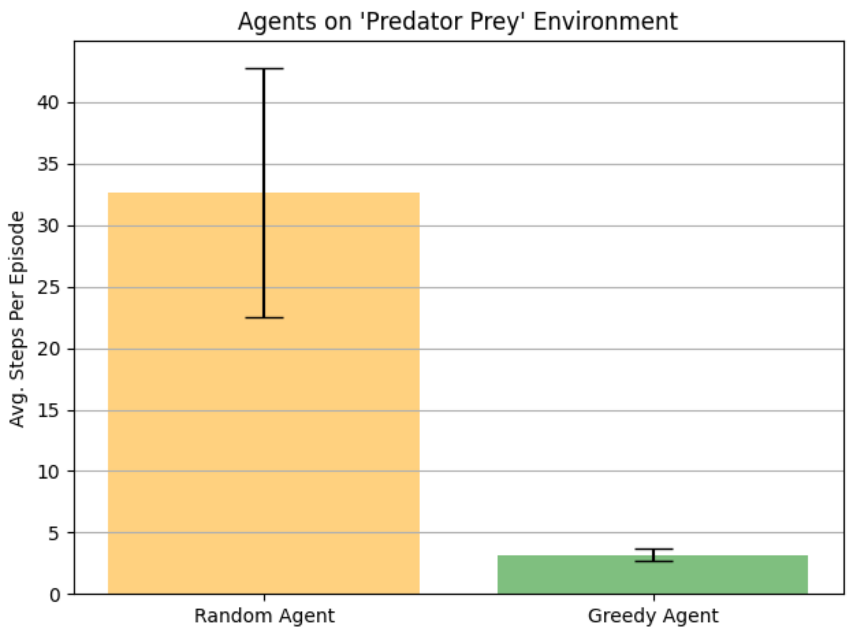
Keep in mind that the greedy predator looks for the prey and chooses to make a step that goes towards it.

Differently from the random agent, the greedy agent maintains an internal state.

Your task is to complete the code for the *action* method in the Greedy agent:

```python
class GreedyAgent(Agent):

    """
    A baseline agent for the SimplifiedPredatorPrey environment.
    The greedy agent finds the nearest prey and moves towards it.
    """

    def __init__(self, agent_id, n_agents):
        super(GreedyAgent, self).__init__(f"Greedy Agent")
        self.agent_id = agent_id
        self.n_agents = n_agents
        self.n_actions = N_ACTIONS

    def action(self) -> int:
        agents_positions = self.observation[:self.n_agents * 2]
        # TODO - Expect 5-8 lines of code
        #  You may use the two auxiliary methods down below
        #  Warning: Your code should work for an arbitrary number of preys
        #  (even though the examples considers a single one)
        raise NotImplementedError()
```

After running the code, you should be able to see the following plot, which depicts the random agent's and greedy agent's average number of time steps to capture the prey (with a 95% confidence interval):



Now try to answer the following questions:

1. Which agent, on average, captures the prey faster?

   Interpreting the resulting graph, since the average number of steps taken by the greedy agent is lower than the value associated with the random agent, we can state the former one catches the prey faster.

2. Can you say which is the best strategy? Justify your answer.

   The best strategy is to follow the closest prey (greedy approach) since the results are clearly better.

3. In this exercise, we ran the *run_single_agent* function to test two different agents (i.e., the random agent and the greedy agent). Note that we assume the prey is part of the environment since it is not controllable. Nevertheless, we still refer to the system as a single agent system and not a multiagent system. Why?

   Because the 2 different agents aren't active in the same environment - they don't interact with each other, thus not being a multiagent system.

Bonus question:

1. Suppose you want to change the task and develop a smarter prey to study the effects of implementing different strategies on the prey's side. Would this new task be considered an autonomous agent system or a multiagent system?

   It would become a multiagent system since there would be 2 different agents interacting (in this case competing) in the same environment.

3.3 Exercise 3: Multi-Agent Environments

Open the file *exercise_3_multi_agent.py*. We now extend the environment to include multiple predators and compare the performance of this task with different predator teams. In particular, the teams are formed with the RandomAgent and GreedyAgent from the previous exercises. Also the *main* section has been updated to evaluate teams instead of individual agents, as shown below.

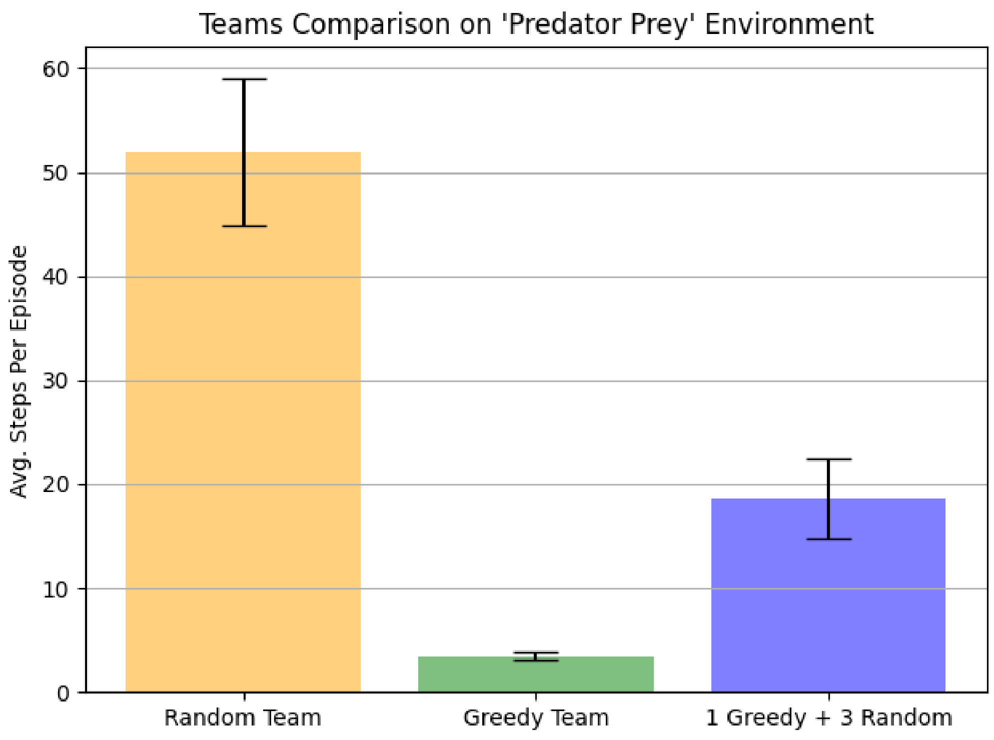
```
if __name__ == '__main__':

        parser = argparse.ArgumentParser()

        parser.add_argument("--episodes", type=int, default=100)
        opt = parser.parse_args()

        # 1 - Setup the environment
        environment = SimplifiedPredatorPrey(grid_shape=(7, 7), n_agents=4, n_preys=1,
max_steps=100)

        # 2 - Setup the teams
        teams = {

        "Random Team": [
                RandomAgent(environment.action_space[0].n),
                RandomAgent(environment.action_space[1].n),
                RandomAgent(environment.action_space[2].n),
                RandomAgent(environment.action_space[3].n),
        ],
        "Greedy Team": [
                GreedyAgent(agent_id=0, n_agents=4),
                GreedyAgent(agent_id=1, n_agents=4),
                GreedyAgent(agent_id=2, n_agents=4),
                GreedyAgent(agent_id=3, n_agents=4)
        ],
        "1 Greedy + 3 Random": [
                GreedyAgent(agent_id=0, n_agents=4),
                RandomAgent(environment.action_space[1].n),
                RandomAgent(environment.action_space[2].n),
                RandomAgent(environment.action_space[3].n)
        ]}

        # 3 - Evaluate teams
        results = {}
        for team, agents in teams.items():
              result = run_multi_agent(environment, agents, opt.episodes)
              results[team] = result

        # 4 - Compare results
        compare_results(
                results, title="Teams Comparison on 'Predator Prey' Environment”,
               colors=["orange", "green", "blue"]
        )
```

Given that we are now dealing with multiple agents, your task is to add code to the *run_multi_agent* function, as shown below:

```
def run_multi_agent(environment: Env, agents: Sequence[Agent], n_episodes: int) -> np.ndarray:

        results = np.zeros(n_episodes)

        for episode in range(n_episodes):

                steps = 0
                terminals = [False for _ in range(len(agents))]
                observations = environment.reset()

                while not all(terminals):
                        steps += 1
                       # TODO - Main Loop (4-6 lines of code)
                       raise NotImplementedError()

        results[episode] = steps

        environment.close()

        return results
```

TIP: See SingleAgentWrapper (aasma/wrappers.py) to provide some inspiration

After running the code, you should be able to see the following plot, which depicts each team's average number of time steps to capture the prey (with a 95% confidence interval):



Now try to answer the following questions:

1. Compare the random team's performance with the single random agent's performance (Exercise 1). Which one has the best performance? Justify your answer.

   The random team's performance is overall worse than the single random agent's performance, having a higher average of time steps taken (due to there being more agents - there is a need for 2 agents to be adjacent to the prey in order to capture it and since the actions are random, it is difficult to have coordenate).

2. Compare the greedy team's performance with the single greedy agent's performance (Exercise 2). Which one has the best performance? Justify your answer.

   It is more or less the same performance. They can coordenate better than the random team but still struggle with the added criteria for capturing the prey.

3. Can you think of any limitations in the greedy strategy? Hint: Try adding required_captors=4 in line 44 and render the interaction to see what happens.

   The prey would (and does) survive almost every time. Even the greedy team only manages an average of 90 time steps.

Bonus question:

1. Both random and greedy agents act independently. If the agents could coordinate their actions with each other, (that is, agree on a plan before the next move) could they implement an even better strategy. Think of an example.

   Yes, the strategy would improve. Perhaps by trying to surround the prey they would have better results (1 predator would approach from the left (from tiles with lower x value), another would approach from above (from tiles with higher y value), and so on).

The End