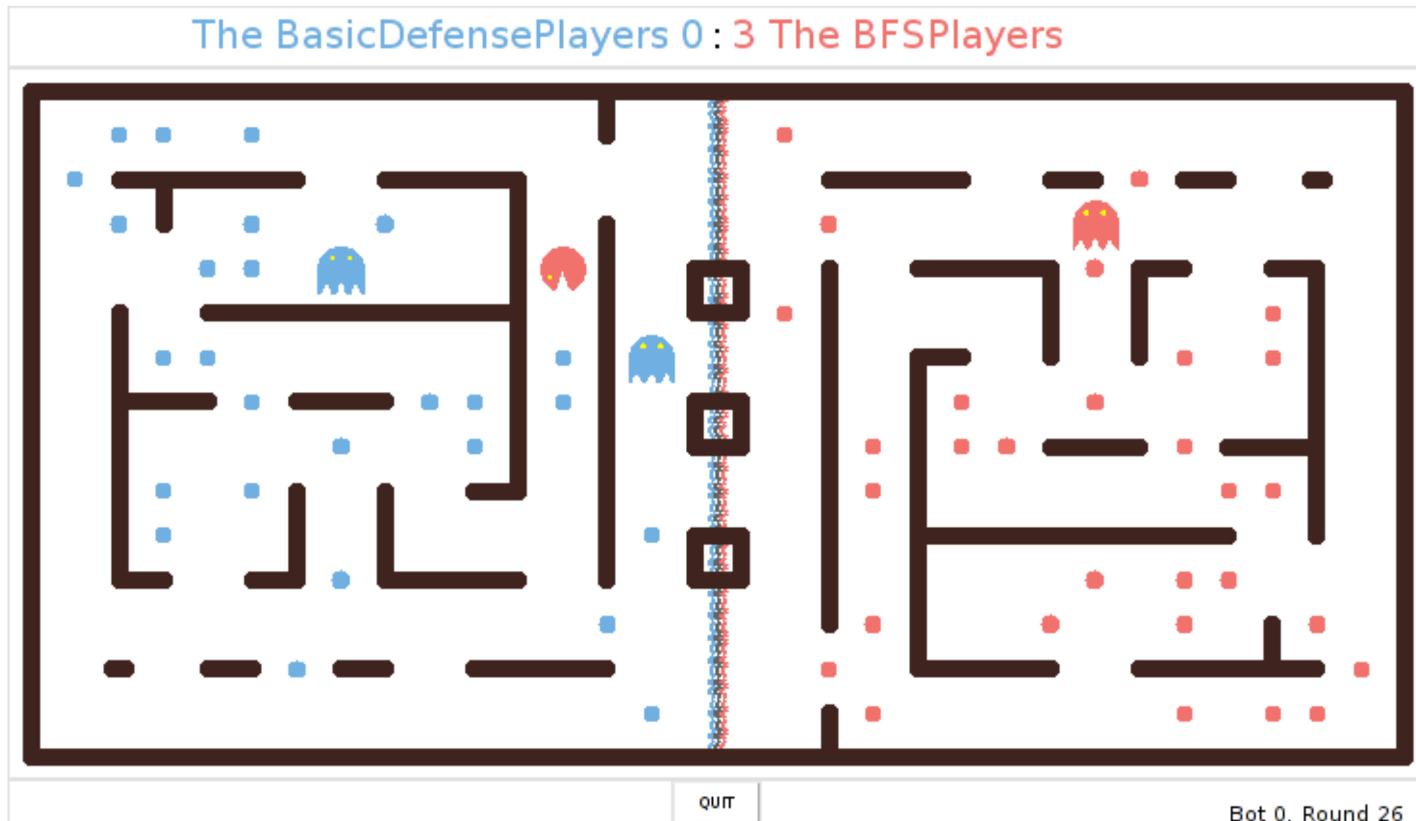
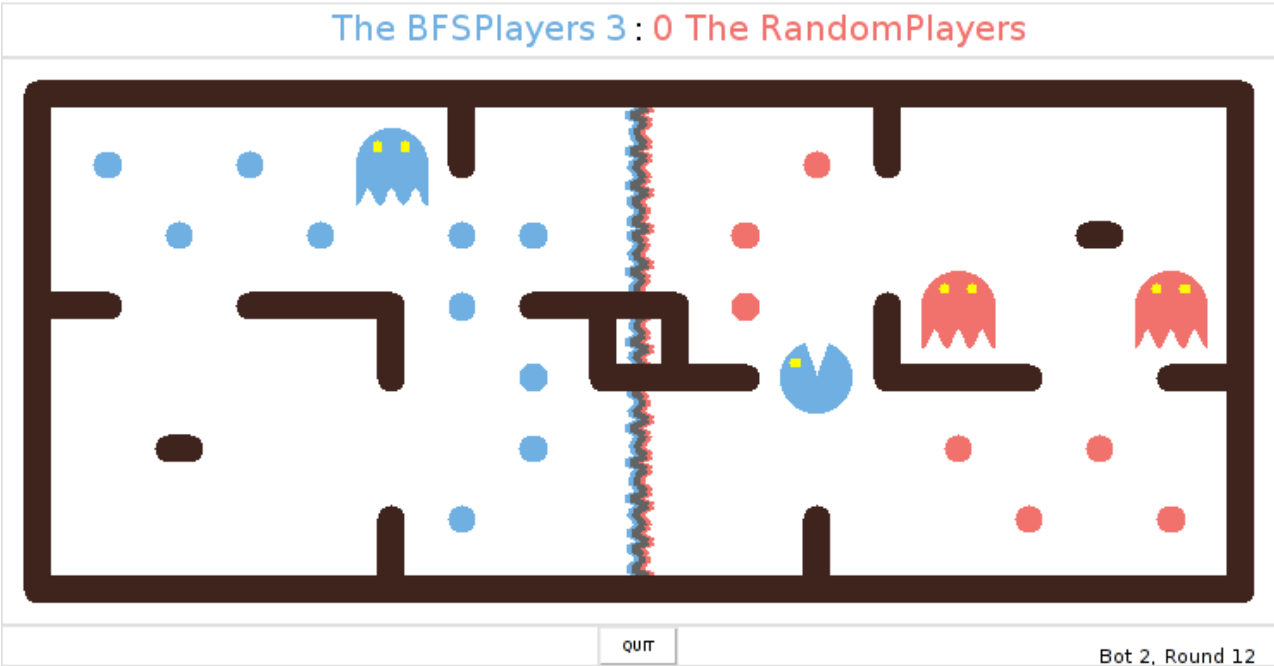
The **Pelita** *contest*

(a brief introduction)

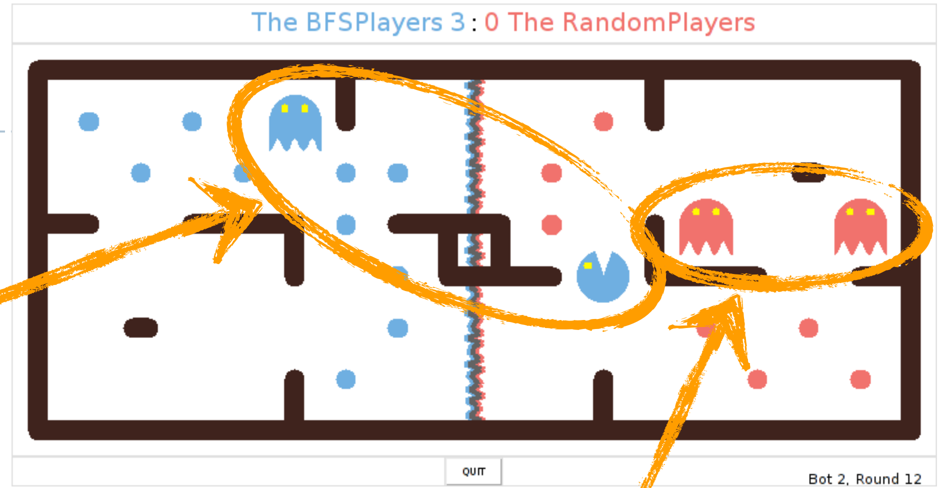


Overview



Overview

- Each Team owns two Bots

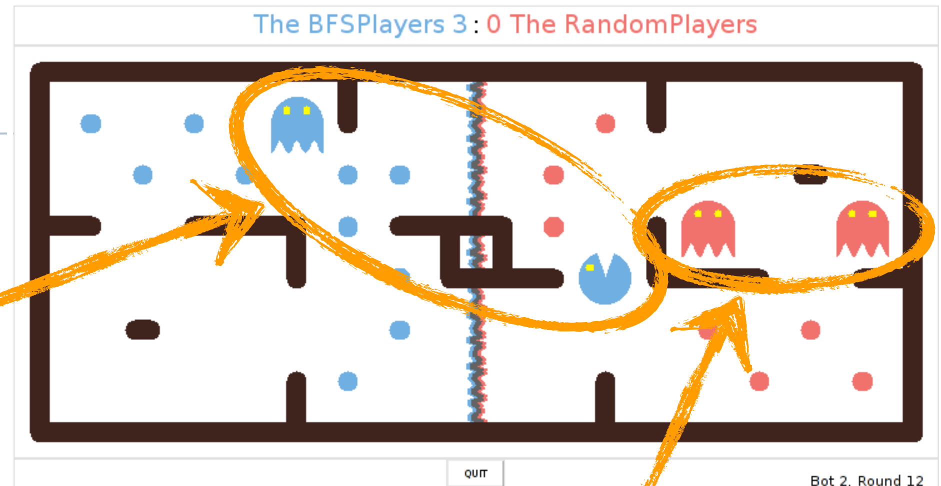


Bots for team 0

Bots for team 1

Overview

- Each Team owns two Bots
- Each Bot is controlled by a Player



Bots for team 0

Player for team 0

Bots for team 1

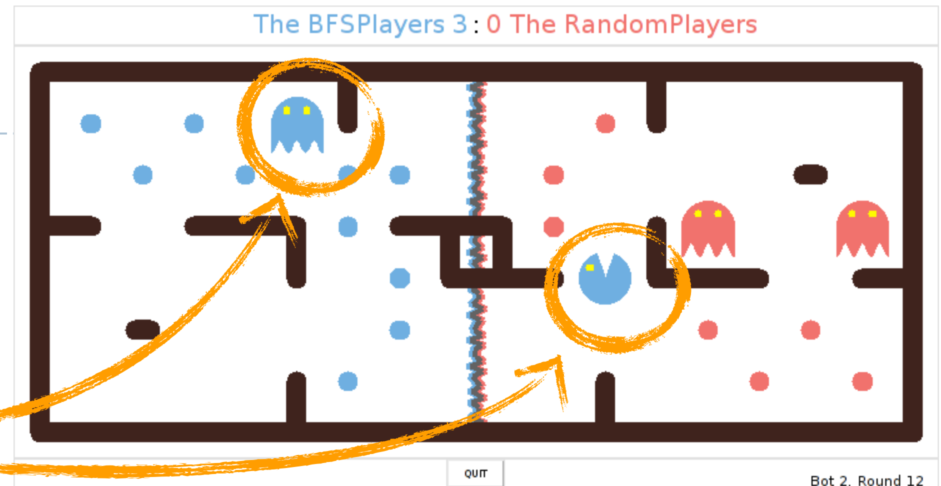
Player for team 1

```
1 import random
2
3 from pelita.datamodel import north, east, south, west, stop
4 from pelita.player import AbstractPlayer
5
6 class UnidirectionalPlayer(AbstractPlayer):
7     def get_move(self):
8         return west
9
10 class UnidirectionalPlayer(AbstractPlayer):
11     def get_move(self):
12         all_directions = self.legal_moves
13         random_direction = random.choice(all_directions)
14         return random_direction
15
```

```
1 import random
2
3 from pelita.datamodel import north, east, south, west, stop
4 from pelita.player import AbstractPlayer
5
6 class UnidirectionalPlayer(AbstractPlayer):
7     def get_move(self):
8         return west
9
10 class UnidirectionalPlayer(AbstractPlayer):
11     def get_move(self):
12         all_directions = self.legal_moves
13         random_direction = random.choice(all_directions)
14         return random_direction
15
```

Overview

- Each Team owns two Bots
- Each Bot is controlled by a Player
- Harvester or Destroyer Bots



Overview

- Each Team owns two Bots
- Each Bot is controlled by a Player
- Harvester or Destroyer Bots
- Bots are Destroyers in homezone
- Harvesters in enemy's homezone
- Game ends when all food pellets are eaten

The rules

- **Scoring**: When a Bot eats a food pellet, the food is permanently removed and **one point** is scored for that Bot's team.
- **Timeout**: Each Player only has 3 seconds to return a valid move. If it doesn't, a random move is executed. (All later return values are discarded.) **Five timeouts and you're out!**
- **Eating a Bot**: When a Bot is eaten by an opposing destroyer, it returns to its starting position (as a harvester). **Five points** are awarded for eating an opponent.
- **Winning**: A game ends when either one team eats all of the opponents' food pellets, or the team with more points after **300 rounds**.
- **Observations**: Bots can only observe an opponent's exact position, if they or their teammate are within 5 squares (maze distance). If they are further away, the opponent's positions are noised.

Getting ready

- Clone the central repository with the game files:
  - `git clone git://github.com/Debilski/pelita.git`
- Run a simple demo game:
  - `~/pelita/pelitagame`
- For help:
  - `~/pelita/pelitagame --help`
- See the Pelita documentation:
  - http://debilski.github.com/pelita
- Write your own player

Implementing the first players

Standard imports

Pelita imports

Implement a simple player

Use the player API

```
1  import random
2
3  from pelita.datamodel import north, east, south, west, stop
4  from pelita.player import AbstractPlayer
5
6  class UnidirectionalPlayer(AbstractPlayer):
7      def get_move(self):
8          return west
9
10 class DrunkPlayer(AbstractPlayer):
11     def get_move(self):
12         all_directions = self.legal_moves
13         random_direction = random.choice(all_directions)
14         return random_direction
15
```

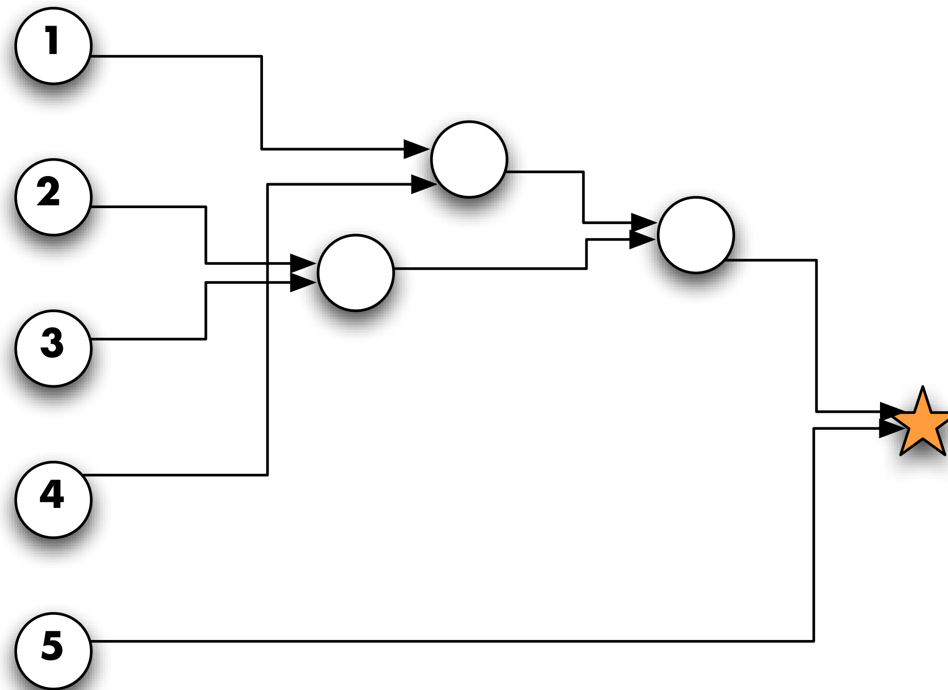
Invalid return values of get_move result in an automatic random move.

The tournament — preliminary rounds

- On the last day, we'll have a tournament in two parts
- Preliminary rounds: all-against-all

The tournament — finals

- Final rounds for the four best teams
- Last-chance final against the fifth best team



Writing Players 101 — the factory

- For the tournament, you'll need a specific project structure
- Clone your group's repository:
  - `git clone <name>@python.g-node,de:/git/groupX`
- Make it a module by adding an init file with a special method factory
  - `groupX/__init__.py`
  -
```
from pelita.players import SimpleTeam, AbstractPlayer

class MyPlayer(AbstractPlayer):
    def get_move(self):
        return (-1, 0)

def factory:
    return SimpleTeam("The Winners", MyPlayer(), MyPlayer())
```
- More information and an example package in the wiki

Writing Players 101 — Player

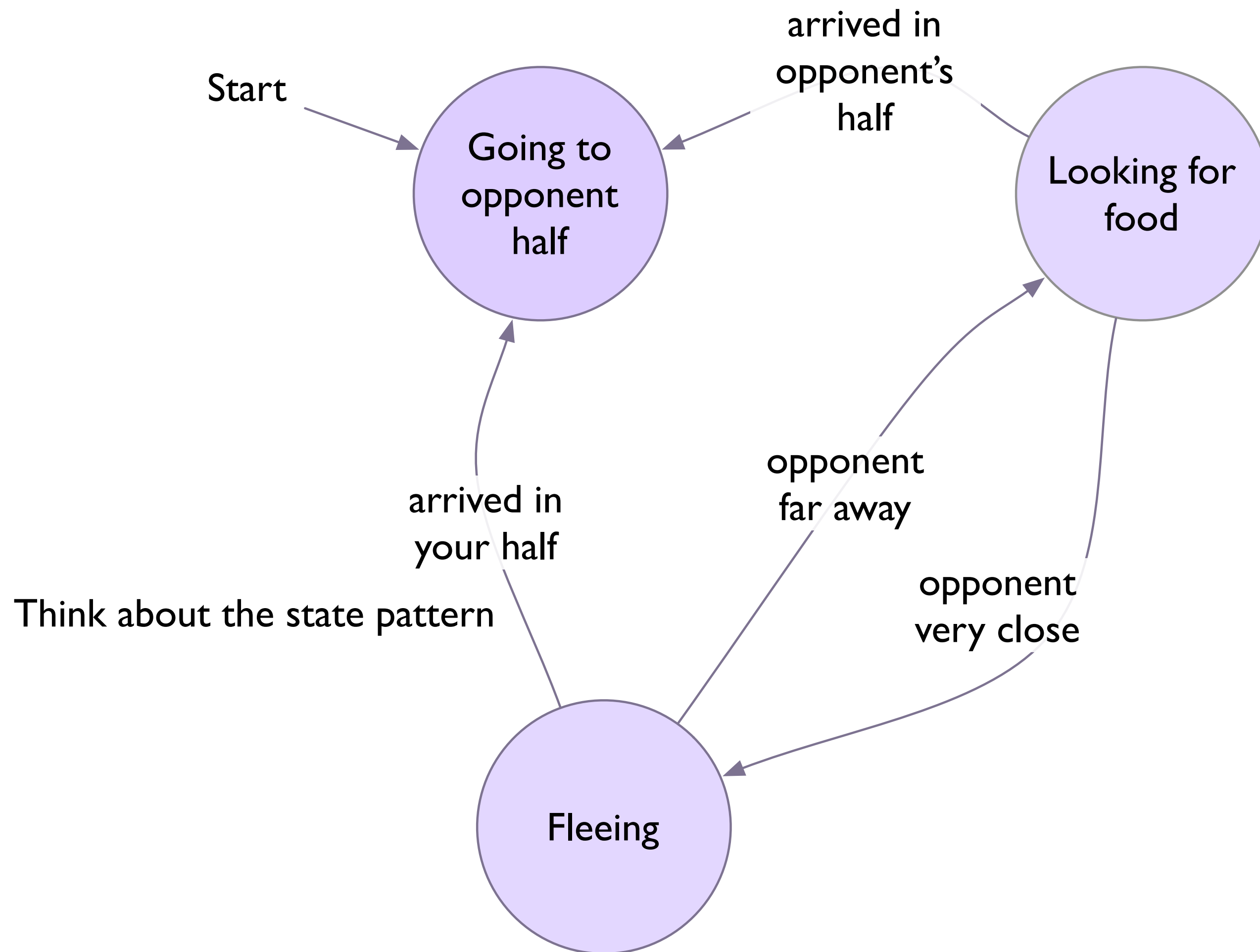
- In your get_move method, information about the current universe and food situation is available. See the documentation for more details.
- self.current_pos
  - Where am I?
- self. me
  - Which bot am I controlling?
- self. enemy_bots
  - Who and where are the other bots?
- self. enemy_food
  - Which are the positions of the food pellets?
- self. current_uni
  - Retrieve the universe you live in.
- self. current_uni.maze
  - How does my world look like?
- self. legal_moves
  - Where can I go?

Writing Players 101 — Testing Players

- **Very useful**
  - The alternative is to run games, hope that the Players end up in the right situation, guess from looking at the screen if it behaved correctly
- **More sophisticated testing scenario**
  - Write a test layout and check that your Player behaves correctly, e.g. for the Player always moving west:
  - Create a file and run the script with it.
  - See documentation for more information

```
############
#0  .     1#
#     ##   #
#2  . #   3#
############
```

Basic Player behaviors — Finite State Machines



Think about the state pattern

Basic Player behaviors — Value-maximizer

- Player has a function that gives a value to a given game state according to several criteria, e.g.
  - $value(game_state) = -1 \times distance_from_nearest_food + 100 \times score$
- At each turn:
  - get the legal actions Player.legal_moves
  - request the future universe, given one of the actions self.current_uni.copy().move_bot(self._index, direction)
  - compute the value of future states
  - pick the action that leads to the state with the highest value

Learning

- Plenty of opportunities for learning
  - Adapt parameters according to final score
  - Reinforcement Learning (similar to learning weights in the value-maximizing Player)
  - Collect statistics on opponents
  - Ambitious: Genetic Programming
  - ...

Things that we've found to be useful

- Shortest-path algorithm
- Algorithm to keep track of opponents
- Communication between Players (requires investigating the SimpleTeam initialisation in the factory method)
- …

- Code re-use is encouraged
- More important than fancy strategies is the quality of your code: Is it well tested? Does it conform to standards? Apply agile development techniques

Let's start!

- Form 5 teams of 6 people (wiki)
- Test that you can write and run matches with simple players
  - set up your project directory:
    - clone the game files
    - clone your group repository
  - copy a random Player and corresponding Player's factory, try to have a few matches with different layouts
  - write a Player that picks a random direction at junctions
- Organize team work
- Have fun!