

---

# **Domination Game Documentation**

*Release 1.6.2*

**Thomas van den Berg and Tim Doolan**

October 26, 2015



<b>1</b>	<b>Intro</b>	<b>1</b>
<b>2</b>	<b>Contents</b>	<b>3</b>
2.1	Running a Game . . . . .	3
2.2	Creating Agents . . . . .	6
2.3	Using Scenarios . . . . .	10
2.4	Customizing the Field . . . . .	12
2.5	Utilities . . . . .	13
2.6	Third Party Libraries . . . . .	15
<b>3</b>	<b>Quickstart</b>	<b>25</b>
<b>4</b>	<b>Indices and tables</b>	<b>27</b>
	<b>Python Module Index</b>	<b>29</b>



### Intro

---

The domination game is a game played by two teams of agents. They will combat one another and accumulate points through capturing control points on the map. The team with the most agents on a control point will capture that control point. These control points remain captured by the same team even when left alone. Agents are capable of picking up ammo, that spawns at designated positions on the map, and use it to shoot other agents. Upon death, agents will respawn in their teams' designated spawn areas. Agents can freely roam the map, but are unable to walk through walls or other agents.

Within one iteration an agent can turn, change its speed, and shoot (in that order). To assure that simulations can terminate in reasonable time, there is a reaction time limit per iteration per agent. Simply, if the agent exceeds this limit it will not do anything. Map layouts (walls, control points and such) are known at the start of the game, but other info are not commonly known and have to be observed by the agents (ammopacks and agents).



## 2.1 Running a Game

In order to run a game, you need to import the `domination` module, and either create a `Scenario`, or create a `Game` object directly.

### 2.1.1 Creating a Game object directly

The simplest way you can use the game object, is to just instantiate it and call its `run()` method. This will run a game with all its default settings:

```
from domination import core
core.Game(rendered=True).run() # Set rendered=False if you don't have pygame.
```

However, creating a game object directly is useful mainly if you want to do some fiddling with its internals, so we recommend skipping right to [Creating Agents](#) or [Using Scenarios](#).

If we like, we can mess around a bit with the game object and its properties:

```
from domination import core

# Make it a short game
settings = core.Settings(max_steps=20)

# Initialize a game
game = core.Game('domination/agent.py', 'domination/agent.py',
                 record=True, rendered=False, settings=settings)

# Will run the entire game.
game.run()

# And now let's see the replay!
replay = game.replay
playback = core.Game(replay=replay)
playback.run()
```

### 2.1.2 Game

The `Game` class has the following specification.

```
class domination.core.Game (red=<open file '/home/docs/checkouts/readthedocs.org/user_builds/domination-
game/checkouts/latest/domination/agent.py', mode 'r'>, blue=<open
file      '/home/docs/checkouts/readthedocs.org/user_builds/domination-
game/checkouts/latest/domination/agent.py', mode 'r'>, red_init={},
blue_init={}, settings=Settings(), field=None, record=False, re-
play=None, rendered=True, verbose=True, hard_errors=False,
step_callback=None)
```

The main game class. Contains game data and methods for simulation.

Constructor for Game class

### Parameters

- **red** – Descriptor of the red agent. Can be either a path, an open file, a string with the class definition, or an instance of `Team`
- **blue** – Descriptor of the blue agent
- **red\_init** – A dictionary of keyword arguments passed to the red agent constructor.
- **blue\_init** – Like `red_init`.
- **settings** – Instance of the settings class.
- **field** – An instance of `Field` to play this game on, or a generator.
- **record** – Store all actions in a game replay.
- **replay** – Pass a game replay to play it.
- **rendered** – Enable/disable the renderer.
- **verbose** – Print game log to output.
- **hard\_errors** – Enable to make agent errors interrupt the game.
- **step\_callback** – Function that is called on every step. Useful for debugging.

**log = None**

The game log as an instance of class: `~domination.core.GameLog`

**replay = None**

The replay object, can be accessed after game has run

**stats = None**

Instance of `GameStats`.

**red = None**

Instance of `Team`.

**blue = None**

Instance of `Team`.

**run ()**

Start and loop the game.

```
class domination.core.GameStats
```

**score\_red = None**

The number of points scored by red

**score\_blue = None**

The number of points scored by blue



**score = None**

The final score as a float (red/total)

**steps = None**

Number of steps the game lasted

**ammo\_red = None**

Number of ammo packs that red picked up

**ammo\_blue = None**

Idem for blue

**deaths\_red = None**

Number red agents that got shot

**deaths\_blue = None**

Number blue agents that got shot

**think\_time\_red = None**

Total time in seconds that red took to compute actions

**think\_time\_blue = None**

Idem for blue

### 2.1.3 Replays

Running replays is easy, first you need to unpack them:

```
>>> import pickle
>>> from domination import core
>>> rp = pickle.load(open('replay20120215-1341_t2v1_vs_t6v1.pickle', 'rb'))
>>> print rp
<domination.core.ReplayData object at 0x10fca5fd0>
```

Then you call the play method:

```
>>> rp.play()
```

**class** `domination.core.ReplayData` (*game*)

Contains the replaydata for a game.

**play** ()

Convenience method for setting up a game to play this replay.

### 2.1.4 Settings

**class** `domination.core.Settings` (*max\_steps=600, max\_score=400, max\_turn=1.0471975511965976, max\_speed=40, max\_range=60, max\_see=100, field\_known=True, ammo\_rate=20, ammo\_amount=3, agent\_type='tank', spawn\_time=10, tilesize=16, think\_time=0.01, capture\_mode=2, end\_condition=1*)

Constructor for Settings class

#### Parameters

- **max\_steps** – How long the game will last at most
- **max\_score** – If either team scores this much, the game is finished
- **max\_speed** – Number of game units each tank can drive in its turn

- **max\_turn** – The maximum angle that a tank can rotate in a turn
- **max\_range** – The shooting range of tanks in game units
- **max\_see** – How far tanks can see (vision is a square with sides that are 2x this value)
- **field\_known** – Whether the agents have knowledge of the field at game start
- **ammo\_rate** – How long it takes for ammo to reappear
- **ammo\_amount** – How many bullets there are in each ammo pack
- **agent\_type** – Type of the agents ('tank' or 'vacubot')
- **spawn\_time** – Time that it takes for tanks to respawn
- **think\_time** – How long the tanks have to do their computations (in seconds)
- **capture\_mode** – One of the CAPTURE\_MODE constants.
- **end\_condition** – One of the ENDGAME flags. Use bitwise OR for multiple.
- **tilesize** – How big a single tile is (game units), change at risk of massive bugginess

The `Settings.capture_mode` can be one of:

```
domination.core.CAPTURE_MODE_NEUTRAL = 0
```

Controlpoints are neutral when occupied by both teams

```
domination.core.CAPTURE_MODE_FIRST = 1
```

Controlpoints stay in control of first team that captures them

```
domination.core.CAPTURE_MODE_MAJORITY = 2
```

Controlpoints are owned by the team with the most occupiers

The `Settings.end_condition` can be one of:

```
domination.core.ENDGAME_NONE = 0
```

End game when time expires

```
domination.core.ENDGAME_SCORE = 1
```

End game when either team has 0 score

```
domination.core.ENDGAME_CRUMBS = 2
```

End game when all crumbs are picked up

## 2.2 Creating Agents

Writing agents consists of creating a Python class that implements *five* methods, some of which are optional. The agents are imported using Python's `exec` method, after which the class named *Agent* is extracted. It is probably easiest to refer to and modify the *default agent*. But there is a quick rundown of the functions below as well.

The first thing you need to do is create a new file with a class named *Agent* that contains these 5 methods:

```
class Agent(object):

    NAME = "my_agent" # Replay filenames and console output will contain this name.

    def __init__(self, id, team, settings=None, field_rects=None, field_grid=None, nav_mesh=None, **kwargs):
        pass

    def observe(self, observation):
        pass
```

```

def action(self):
    return (0,0,False)

def debug(self, surface):
    pass

def finalize(self, interrupted=False):
    pass

```

### 2.2.1 Initialize

It needs to implement an `__init__` method that accepts a number of setup arguments. This method will be called for each agent at the beginning of each game.

```

Agent.__init__(id, team, settings=None, field_rects=None, field_grid=None, nav_mesh=None,
               blob=None, **kwargs)

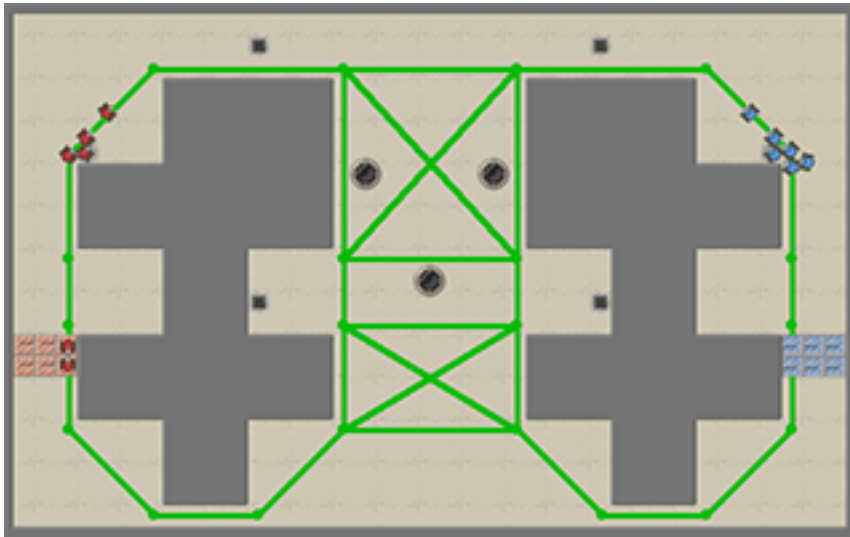
```

Each agent is initialized at the beginning of each game. The first agent (`id==0`) can use this to set up global variables. Note that the properties pertaining to the game field might not be given for each game.

The `settings` object is an instance of `Settings`, and contains all the game settings such as game length and maximum score. The `field_rects`, `field_grid`, and `nav_mesh` arguments provide some information about the map that the game will be played on. The first contains a list of walls on the map as `(x,y,width,height)` tuples, the second contains the same information, but as a 2D binary array instead.

#### Navigation Mesh

Also passed to the agent constructor is a ‘navigation mesh’. This is a directed graph containing **the set of points from which all points on the map can be seen**, and the straight lines connecting them. You can use it in conjunction with `find_path()` to plan paths.



It is structured as a dictionary where the keys are `(x, y)` tuples defining connectivity and distances. All connections are in this dictionary *two times*, both `A → B` and `B → A` are in there. The example below shows a point at `(0, 0)` connected to two other points, at `(1, 0)` and `(0, 2)`:

```

{(0, 0): {(1, 0): 1.0,
          (0, 2): 2.0},

```

```
(1, 0): {(0, 0): 1.0},
(0, 2): {(0, 0): 2.0}}
```

## Agent Parameters

Finally, you can provide extra arguments to “parametrize” your agents. You can set these arguments when you start a new game. For example, if your initialization looks as follows:

```
def __init__(self, id, team, settings, field_rects, field_grid, nav_mesh, aggressiveness=0.0):
```

Then you can set this parameter to different values when you start the game:

```
MyScenario('my_agent.py', 'opponent.py', red_init={'aggressiveness':10.0}).run()
MyScenario('my_agent.py', 'opponent.py', red_init={'aggressiveness':20.0}).run()
```

## 2.2.2 Observe

The second method you need to implement is `observe`. This method is passed an *observation* of the current game state, depending on the settings, agents usually don’t observe the entire game field, but only a part of it. Agents use this function to update what they know about the game, e.g. computing the most likely locations of enemies. The properties of the *Observation* object are listed below.

Agent `.observe` (*observation*)

Each agent is passed an observation using this function, before being asked for an action. You can store either the observation object or its properties to use them to determine your action. Note that the observation object is modified in place.

```
class Observation(object):
    def __init__(self):
        self.step = 0 # Current timestep
        self.loc = (0,0) # Agent's location (x,y)
        self.angle = 0 # Current angle in radians
        self.walls = [] # Visible walls around the agent: a 2D binary array
        self.friends = [] # All/Visible friends: a list of (x,y,angle)-tuples
        self.foes = [] # Visible foes: a list of (x,y,angle)-tuples
        self.cps = [] # Controlpoints: a list of (x,y,TEAM_RED/TEAM_BLUE)-tuples
        self.objects = [] # Visible objects: a list of (x,y,type)-tuples
        self.ammo = 0 # Ammo count
        self.score = (0,0) # Current game score
        self.collided = False # Whether the agent has collided in the previous turn
        self.respawn_in = -1 # How many timesteps left before this agent can move again.
        self.hit = None # What the agent hit with its last shot. Can be None/TEAM_RED/TEAM_BLUE
        # The following properties are only set when
        # the renderer is enabled:
        self.selected = False # Indicates if the agent is selected in the UI
        self.clicked = [] # A list of mouse-clicks, tuples of (x, y, shift, selected)
        self.keys = [] # A list of all keys pressed in the previous turn

    def __str__(self):
        items = sorted(self.__dict__.items())
        maxlen = max(len(k) for k,v in items)
        return "== Observation ==\n" + "\n".join('%s : %r'%(k.ljust(maxlen), v)) for (k,v) in items)
```

### 2.2.3 Action

This is the most important function you have to implement. It should return a tuple containing a representation of the action you want the agent to perform. In this game, the action tuples are supposed to look like `(turn, speed, shoot)`.

- **Turn** indicates how much your tank should spin around it's center.
- **Speed** indicates how much you want your tank to drive forward after it has turned.
- **Shoot** is set to True if you want to fire a projectile in this turn.

**Turn** is given in radians, and **Speed** is given in game units (corresponding to pixels in the renderer). Note that any exceptions raised by your agent are ignored, and the agent simply loses it's turn. Turn and speed are capped by the game settings.

`Agent.action()`

This function is called every step and should return a tuple in the form: `(turn, speed, shoot)`

### 2.2.4 Debug

Allows the agents to draw on the game UI, refer to the pygame reference to see how you can [draw on a pygame.surface](#). The given surface is not cleared automatically. Additionally, this function will only be called when the renderer is active, and it will only be called for the active team.

`Agent.debug(surface)`

Allows the agents to draw on the game UI, Refer to the pygame reference to see how you can draw on a `pygame.surface`. The given surface is not cleared automatically. Additionally, this function will only be called when the renderer is active, and it will only be called for the active team.

### 2.2.5 Finalize

This method gives your agent an opportunity to store data or clean up after the game is finished. Learning agents could store their Q-tables, which they load up in `__init__`.

`Agent.finalize(interrupted=False)`

This function is called after the game ends, either due to time/score limits, or due to an interrupt (CTRL+C) by the user. Use it to store any learned variables and write logs/reports.

### 2.2.6 Communication

The recommended way to establish communication between agents is to define [static attributes](#) in the `Agent` class definition. Static attributes are variables that are identical for every instance of the class, essentially, they are attributes of the `class`, not of the instances.

In Python, static variables can be defined in the class body, and accessed through the class definition. Be careful, setting `Agent.attribute` is quite different from setting `my_agent = Agent(); my_agent.attribute`:

```
class Agent:
    shared_knowledge = 1

    def __init__(self, etc):
        print Agent.shared_knowledge
        # is identical to
        print self.__class__.shared_knowledge
```

```
# BUT THIS IS DIFFERENT:
self.shared_knowledge = 5
```

## 2.2.7 (Binary) Data

You might want to supply your agent with additional (binary) data, for example a Q/value table, or some kind of policy representation. The convention for doing this is to pass an open file-pointer to the agent's constructor:

```
Game(..., red_init={'blob': open('my_q_table', 'rb')})
```

This is also the way that your data will be passed to the agent in the web app. If you have stored your data as a pickled file, you can simply read it using `file.read()`, then unpickle its contents from a string using `pickle.loads()`

```
# In class Agent
def __init__(..., blob=None):
    if blob is not None:
        my_data = pickle.loads(blob.read())
        blob.seek(0) #: Reset the filepointer for the next agent.
                    # if you omit this, the next agent will raise an EOFError
```

Of course, the way you store your data in this file is up to you, you can store it in any format, and even read it line-by-line if you want.

## 2.3 Using Scenarios

Because most usage of the game will be more or less the same, some stuff has been automated in the form of a Scenario. Scenarios offer a way to define settings and score conditions, and automatically save the results of repeated runs.

For example, we subclass the Scenario module from `domination.scenarios`:

```
import domination

class MyScenario(domination.scenarios.Scenario):
    REPEATS = 10
    SETTINGS = core.Settings()
    FIELD = core.FieldGenerator().generate()

    def before_each():
        #: Regenerate the field before each game.
        self.FIELD = core.FieldGenerator().generate()
```

We can now run our scenario and save the results:

```
MyScenario.one_on_one('agent_one.py', 'agent_two.py', output_folder='results')
```

When a tournament is run, using `domination.scenarios.Scenario.tournament()` a `MatchInfo` object is passed to the agent constructor.

### 2.3.1 Reference

**class** `domination.scenarios.Scenario`

A scenario is used to run multiple games under the same conditions.

**SETTINGS = Settings()**

The settings with which these games will be played

**GENERATOR = <domination.core.FieldGenerator object>**

Will generate FIELD before each game if defined

**FIELD = None**

Will play on this field if GENERATOR is None

**REPEATS = 4**

How many times to repeat each game

**SWAP\_TEAMS = True**

Repeat each run with blue/red swapped

**setup ()**

Function is called once before any games

**before\_game ()**

Function that is run before each game. Use it to regenerate the map, for example.

**after\_game (game)**

Function that is run after each game.

**Parameters game** – The previous game

**classmethod test (red, blue)**

Test this scenario, this will run a single game and render it, so you can verify the FIELD and SETTINGS.

**Parameters**

- **red** – Path to red agent
- **blue** – Path to blue agent

**classmethod one\_on\_one (output\_folder, red, blue, rendered=False, verbose=False)**

Runs the set amount of REPEATS and SWAP\_TEAMS if desired, between two given agents.

**Parameters output\_folder** – Folder in which results will be stored

**classmethod tournament (output\_folder, folder=None, agents=None, rendered=False, verbose=False)**

Runs a full tournament between the agents specified, respecting the REPEATS and SWAP\_TEAMS settings.

**Parameters**

- **agents** – A list of paths to agents
- **folder** – A folder that contains all agents, overrides the agents parameter.
- **output\_folder** – Folder in which results will be stored.

**class domination.scenarios.MatchInfo (num\_games, current, match\_id, score\_weight)**

An instance of this object is passed to agents to let them know that they are participating in a match consisting of multiple games, and which game they are currently in.

Constructor for MatchInfo

**Parameters**

- **num\_games** – The total number of games in this match
- **current** – The current game with 1 being the first game.
- **match\_id** – A unique id of the opponent the agent is playing against.

- **score\_weight** – How much weight is assigned to the score of the current match.

## 2.4 Customizing the Field

Game fields are based on a tilemap where each tile can only be occupied by a single object. This means they can be represented conveniently by an ASCII representation. You can instantiate fields from these ASCII representations as well. Suppose we create a file *field.txt* with the following contents:

```

w w w w w w w w w w w w w w w
w _ _ _ _ _ _ _ _ _ _ _ _ _ w
w R _ _ _ _ _ _ C _ _ _ _ _ B w
w _ _ _ _ _ w _ _ _ _ _ w _ _ _ w
w _ _ _ _ _ w w w w w w w _ _ _ w
w _ _ _ _ _ w _ _ _ _ _ w _ _ _ w
w R _ _ _ _ _ A _ _ _ _ _ B w
w _ _ _ _ _ _ _ _ _ _ _ _ _ w
w w w w w w w w w w w w w w w
    
```

We can then load it up using the `domination.core.Field.from_string()` function, the map defined above looks like the image below:

```

field = core.Field.from_string(open('field.txt').read())
core.Game(field=field).run()
    
```



The default maps are randomly generated using the `FieldGenerator` class, it has a number of parameters for generating maps.

```

class domination.core.FieldGenerator(width=41, height=24, tilesize=16, mirror=True,
                                     num_red=6, num_blue=6, num_points=3, num_ammo=6,
                                     num_crumbsource=0, wall_fill=0.4, wall_len=(3, 7),
                                     wall_width=4, wall_orientation=0.5, wall_gridsize=6)
    
```

Generates field objects from random distribution

Create a `FieldGenerator` object with certain parameters for a random distribution of fields.

### Parameters

- **width** – The width of the field in tiles
- **height** – The height of the field in tiles
- **tilesize** – The size of each tile (don't change from 16)
- **mirror** – Make a symmetrical map
- **num\_blue** – The number of blue spawns



- **num\_red** – The number of red spawns
- **num\_points** – The number of controlpoints
- **num\_ammo** – The number of ammo locations on the map
- **num\_crumbsource** – The number of crumb fountains
- **wall\_fill** – What portion of the map is occupied by walls
- **wall\_len** – A range for the length of wall sections (min, max)
- **wall\_width** – The width of each wall section
- **wall\_orientation** – The probability that each wall will be placed horizontally i.e. that the walls length will be along a horizontal axis
- **wall\_gridsize** – Place walls only at every n-th tile with their top-left

**generate()**

Generates a new field using the parameters for random distribution set in the constructor.

**Returns** A `Field` instance.

## 2.5 Utilities

This module holds functions, exceptions and constants that are or might be used by both the game, renderer and perhaps the agents. By putting this code in a separate module, each of them can access it without requiring the other modules.

`domination.utilities.frange` (*limit1*, *limit2=None*, *increment=1.0*)

Like `xrange`, but for real numbers.

`domination.utilities.mean` (*iterable*)

Returns mean of given list or generator.

`domination.utilities.stdev` (*iterable*)

Returns standard deviation of given list or generator.

```
>>> stdev([1, 2, 3])
1.0
```

`domination.utilities.point_add` (*a*, *b*)

Add the coordinates of two points (Inline this if you can, function calls are slow)

`domination.utilities.point_sub` (*a*, *b*)

Subtract two 2d vectors (Inline this if you can, function calls are slow)

`domination.utilities.point_mul` (*a*, *f*)

Multiply a vector by a scalar (Inline this if you can, function calls are slow)

`domination.utilities.point_dist` (*a*, *b*)

Distance between two points.

`domination.utilities.line_intersects_rect` (*p0*, *p1*, *r*)

Check where a line between *p1* and *p2* intersects given axis-aligned rectangle *r*. Returns `False` if no intersection found. Uses the Liang-Barsky line clipping algorithm.

```
>>> line_intersects_rect((1.0, 0.0), (1.0, 4.0), (0.0, 1.0, 4.0, 1.0))
((0.25, (1.0, 1.0)), (0.5, (1.0, 2.0)))
```

```
>>> line_intersects_rect((1.0,0.0), (3.0,0.0), (0.0,1.0,3.0,1.0))
False
```

domination.utilities.**line\_intersects\_circ**((p0x,p0y), (p1x,p1y), (cx,cy), r)

Computes intersections between line and circle. The line runs between (p0x,p0y) and (p1x,p1y) and the circle is centered at (cx,cy) with a radius r. Returns False if no intersection is found, and one or two intersection points otherwise. Intersection points are (t, (x, y)) where t is the distance along the line between 0-1. (From [stackoverflow.com/questions/1073336/circle-line-collision-detection](http://stackoverflow.com/questions/1073336/circle-line-collision-detection))

```
>>> line_intersects_circ((0,0), (4,0), (2,0), 1)
[(0.25, (1.0, 0.0)), (0.75, (3.0, 0.0))]
```

```
>>> line_intersects_circ((0,0), (2,0), (2,0), 1)
[(0.5, (1.0, 0.0))]
```

```
>>> line_intersects_circ((0,1), (2,1), (1,0), 1)
[(0.5, (1.0, 1.0))]
```

```
>>> line_intersects_circ((0,0), (0,1), (2,0), 1)
False
```

domination.utilities.**line\_intersects\_grid**((x0,y0), (x1,y1), grid, grid\_cell\_size=1)

Performs a line/grid intersection, finding the “super cover” of a line and seeing if any of the grid cells are occupied. The line runs between (x0,y0) and (x1,y1), and (0,0) is the top-left corner of the top-left grid cell.

```
>>> line_intersects_grid((0,0), (2,2), [[0,0,0], [0,1,0], [0,0,0]])
True
```

```
>>> line_intersects_grid((0,0), (0.99,2), [[0,0,0], [0,1,0], [0,0,0]])
False
```

domination.utilities.**rect\_contains\_point**(rect, point)

Check if rectangle contains a point.

domination.utilities.**rect\_offset**(rect, offset)

Offsets (grows) a rectangle in each direction.

domination.utilities.**rect\_corners**(rect)

Returns cornerpoints of given rectangle.

```
>>> rect_corners((1,2,1,3))
((1, 2), (2, 2), (2, 5), (1, 5))
```

domination.utilities.**rects\_bound**(rects)

Returns a rectangle that bounds all given rectangles

```
>>> rects_bound([(0,0,1,1), (3,3,1,1)])
(0, 0, 4, 4)
```

domination.utilities.**rects\_merge**(rects)

Merge a list of rectangle (xywh) tuples. Returns a list of rectangles that cover the same surface. This is not necessarily optimal though.

```
>>> rects_merge([(0,0,1,1), (1,0,1,1)])
[(0, 0, 2, 1)]
```

domination.utilities.**angle\_fix**(theta)

Fixes an angle to a value between -pi and pi.

```
>>> angle_fix(-2*pi)
0.0
```

`domination.utilities.reachable` (*grid*, (*x*, *y*), *border=1*)

Performs a ‘flood fill’ operation to find reachable areas on given tile map from (*x*,*y*). Returns as binary grid with 1 for reachable.

**Parameters** **border** – can be a value or a function indicating borders of region

```
>>> reachable([[0,1,0],[0,1,0]], (0,0))
[[1, 0, 0], [1, 0, 0]]
```

`domination.utilities.make_nav_mesh` (*walls*, *bounds=None*, *offset=7*, *simplify=0.001*, *add\_points=[]*)

Generate an almost optimal navigation mesh between the given walls (rectangles), within the world bounds (a big rectangle). Mesh is a dictionary of dictionaries:

```
mesh[point1][point2] = distance
```

`domination.utilities.find_path` (*start*, *end*, *mesh*, *grid*, *tilesize=16*)

Uses astar to find a path from *start* to *end*, using the given mesh and tile grid.

```
>>> grid = [[0,0,0,0,0],[0,0,0,0,0],[0,0,1,0,0],[0,0,0,0,0],[0,0,0,0,0]]
>>> mesh = make_nav_mesh([(2,2,1,1)], (0,0,4,4),1)
>>> find_path((0,0),(4,4),mesh,grid,1)
[(4, 1), (4, 4)]
```

## 2.6 Third Party Libraries

Included in the domination package are a number of third party libraries.

### 2.6.1 A-Star

#### Introduction

The Munkres module provides an implementation of the Munkres algorithm (also called the Hungarian algorithm or the Kuhn-Munkres algorithm), useful for solving the Assignment Problem.

#### Assignment Problem

Let *C* be an *n*×*n* matrix representing the costs of each of *n* workers to perform any of *n* jobs. The assignment problem is to assign jobs to workers in a way that minimizes the total cost. Since each worker can perform only one job and each job can be assigned to only one worker the assignments represent an independent set of the matrix *C*.

One way to generate the optimal set is to create all permutations of the indexes necessary to traverse the matrix so that no row and column are used more than once. For instance, given this matrix (expressed in Python):

```
matrix = [[5, 9, 1],
          [10, 3, 2],
          [8, 7, 4]]
```

You could use this code to generate the traversal indexes:

```
def permute(a, results):
    if len(a) == 1:
        results.insert(len(results), a)

    else:
        for i in range(0, len(a)):
            element = a[i]
            a_copy = [a[j] for j in range(0, len(a)) if j != i]
            subresults = []
            permute(a_copy, subresults)
            for subresult in subresults:
                result = [element] + subresult
                results.insert(len(results), result)

results = []
permute(range(len(matrix)), results) # [0, 1, 2] for a 3x3 matrix
```

After the call to `permute()`, the results matrix would look like this:

```
[[0, 1, 2],
 [0, 2, 1],
 [1, 0, 2],
 [1, 2, 0],
 [2, 0, 1],
 [2, 1, 0]]
```

You could then use that index matrix to loop over the original cost matrix and calculate the smallest cost of the combinations:

```
n = len(matrix)
minval = sys.maxint
for row in range(n):
    cost = 0
    for col in range(n):
        cost += matrix[row][col]
    minval = min(cost, minval)

print minval
```

While this approach works fine for small matrices, it does not scale. It executes in  $O(n!)$  time: Calculating the permutations for an  $n \times n$  matrix requires  $n!$  operations. For a 12x12 matrix, that's 479,001,600 traversals. Even if you could manage to perform each traversal in just one millisecond, it would still take more than 133 hours to perform the entire traversal. A 20x20 matrix would take 2,432,902,008,176,640,000 operations. At an optimistic millisecond per operation, that's more than 77 million years.

The Munkres algorithm runs in  $O(n^3)$  time, rather than  $O(n!)$ . This package provides an implementation of that algorithm.

This version is based on <http://www.public.iastate.edu/~ddoty/HungarianAlgorithm.html>.

This version was written for Python by Brian Clapper from the (Ada) algorithm at the above web site. (The Algorithm: :Munkres Perl version, in CPAN, was clearly adapted from the same web site.)

## Usage

Construct a Munkres object:

```
from munkres import Munkres

m = Munkres()
```

Then use it to compute the lowest cost assignment from a cost matrix. Here's a sample program:

```
from munkres import Munkres, print_matrix

matrix = [[5, 9, 1],
          [10, 3, 2],
          [8, 7, 4]]
m = Munkres()
indexes = m.compute(matrix)
print_matrix(matrix, msg='Lowest cost through this matrix:')
total = 0
for row, column in indexes:
    value = matrix[row][column]
    total += value
    print '(%d, %d) -> %d' % (row, column, value)
print 'total cost: %d' % total
```

Running that program produces:

```
Lowest cost through this matrix:
[5, 9, 1]
[10, 3, 2]
[8, 7, 4]
(0, 0) -> 5
(1, 1) -> 3
(2, 2) -> 4
total cost=12
```

The instantiated Munkres object can be used multiple times on different matrices.

## Non-square Cost Matrices

The Munkres algorithm assumes that the cost matrix is square. However, it's possible to use a rectangular matrix if you first pad it with 0 values to make it square. This module automatically pads rectangular cost matrices to make them square.

Notes:

- The module operates on a *copy* of the caller's matrix, so any padding will not be seen by the caller.
- The cost matrix must be rectangular or square. An irregular matrix will *not* work.

## Calculating Profit, Rather than Cost

The cost matrix is just that: A cost matrix. The Munkres algorithm finds the combination of elements (one from each row and column) that results in the smallest cost. It's also possible to use the algorithm to maximize profit. To do that, however, you have to convert your profit matrix to a cost matrix. The simplest way to do that is to subtract all elements from a large value. For example:

```
from munkres import Munkres, print_matrix

matrix = [[5, 9, 1],
          [10, 3, 2],
          [8, 7, 4]]
```

```
cost_matrix = []
for row in matrix:
    cost_row = []
    for col in row:
        cost_row += [sys.maxint - col]
    cost_matrix += [cost_row]

m = Munkres()
indexes = m.compute(cost_matrix)
print_matrix(matrix, msg='Highest profit through this matrix:')
total = 0
for row, column in indexes:
    value = matrix[row][column]
    total += value
    print '(%d, %d) -> %d' % (row, column, value)

print 'total profit=%d' % total
```

Running that program produces:

```
Highest profit through this matrix:
[5, 9, 1]
[10, 3, 2]
[8, 7, 4]
(0, 1) -> 9
(1, 0) -> 10
(2, 2) -> 4
total profit=23
```

The `munkres` module provides a convenience method for creating a cost matrix from a profit matrix. Since it doesn't know whether the matrix contains floating point numbers, decimals, or integers, you have to provide the conversion function; but the convenience method takes care of the actual creation of the cost matrix:

```
import munkres

cost_matrix = munkres.make_cost_matrix(matrix,
                                       lambda cost: sys.maxint - cost)
```

So, the above profit-calculation program can be recast as:

```
from munkres import Munkres, print_matrix, make_cost_matrix

matrix = [[5, 9, 1],
          [10, 3, 2],
          [8, 7, 4]]
cost_matrix = make_cost_matrix(matrix, lambda cost: sys.maxint - cost)
m = Munkres()
indexes = m.compute(cost_matrix)
print_matrix(matrix, msg='Lowest cost through this matrix:')
total = 0
for row, column in indexes:
    value = matrix[row][column]
    total += value
    print '(%d, %d) -> %d' % (row, column, value)
print 'total profit=%d' % total
```

## References

1. <http://www.public.iastate.edu/~ddoty/HungarianAlgorithm.html>
2. Harold W. Kuhn. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83-97, 1955.
3. Harold W. Kuhn. Variants of the Hungarian method for assignment problems. *Naval Research Logistics Quarterly*, 3: 253-258, 1956.
4. Munkres, J. Algorithms for the Assignment and Transportation Problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32-38, March, 1957.
5. [http://en.wikipedia.org/wiki/Hungarian\\_algorithm](http://en.wikipedia.org/wiki/Hungarian_algorithm)

## Copyright and License

This software is released under a BSD license, adapted from <<http://opensource.org/licenses/bsd-license.php>>

Copyright (c) 2008 Brian M. Clapper All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name “clapper.org” nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

**class** `domination.libs.munkres.Munkres`

Calculate the Munkres solution to the classical assignment problem. See the module documentation for usage.

Create a new instance

**static** `make_cost_matrix` (*profit\_matrix*, *inversion\_function*)  
**DEPRECATED**

Please use the module function `make_cost_matrix()`.

**pad\_matrix** (*matrix*, *pad\_value=0*)

Pad a possibly non-square matrix to make it square.

### Parameters

**matrix** [list of lists] matrix to pad

**pad\_value** [int] value to use to pad the matrix

**Return type** list of lists

**Returns** a new, possibly padded, matrix

**compute** (*cost\_matrix*)

Compute the indexes for the lowest-cost pairings between rows and columns in the database. Returns a list of (row, column) tuples that can be used to traverse the matrix.

**Parameters**

**cost\_matrix** [list of lists] The cost matrix. If this cost matrix is not square, it will be padded with zeros, via a call to `pad_matrix()`. (This method does *not* modify the caller's matrix. It operates on a copy of the matrix.)

**WARNING:** This code handles square and rectangular matrices. It does *not* handle irregular matrices.

**Return type** list

**Returns** A list of (row, column) tuples that describe the lowest cost path through the matrix

`domination.libs.munkres.make_cost_matrix` (*profit\_matrix*, *inversion\_function*)

Create a cost matrix from a profit matrix by calling 'inversion\_function' to invert each value. The inversion function must take one numeric argument (of any type) and return another numeric argument which is presumed to be the cost inverse of the original profit.

This is a static method. Call it like this:

For example:

**Parameters**

**profit\_matrix** [list of lists] The matrix to convert from a profit to a cost matrix

**inversion\_function** [function] The function to use to invert each entry in the profit matrix

**Return type** list of lists

**Returns** The converted matrix

## 2.6.2 Hungarian Algorithm

This is an algorithm for solving the assignment problem.

### Introduction

The Munkres module provides an implementation of the Munkres algorithm (also called the Hungarian algorithm or the Kuhn-Munkres algorithm), useful for solving the Assignment Problem.

### Assignment Problem

Let  $C$  be an  $n \times n$  matrix representing the costs of each of  $n$  workers to perform any of  $n$  jobs. The assignment problem is to assign jobs to workers in a way that minimizes the total cost. Since each worker can perform only one job and each job can be assigned to only one worker the assignments represent an independent set of the matrix  $C$ .

One way to generate the optimal set is to create all permutations of the indexes necessary to traverse the matrix so that no row and column are used more than once. For instance, given this matrix (expressed in Python):



```
matrix = [[5, 9, 1],
          [10, 3, 2],
          [8, 7, 4]]
```

You could use this code to generate the traversal indexes:

```
def permute(a, results):
    if len(a) == 1:
        results.insert(len(results), a)

    else:
        for i in range(0, len(a)):
            element = a[i]
            a_copy = [a[j] for j in range(0, len(a)) if j != i]
            subresults = []
            permute(a_copy, subresults)
            for subresult in subresults:
                result = [element] + subresult
                results.insert(len(results), result)

results = []
permute(range(len(matrix)), results) # [0, 1, 2] for a 3x3 matrix
```

After the call to permute(), the results matrix would look like this:

```
[[0, 1, 2],
 [0, 2, 1],
 [1, 0, 2],
 [1, 2, 0],
 [2, 0, 1],
 [2, 1, 0]]
```

You could then use that index matrix to loop over the original cost matrix and calculate the smallest cost of the combinations:

```
n = len(matrix)
minval = sys.maxint
for row in range(n):
    cost = 0
    for col in range(n):
        cost += matrix[row][col]
    minval = min(cost, minval)

print minval
```

While this approach works fine for small matrices, it does not scale. It executes in  $O(n!)$  time: Calculating the permutations for an  $n \times n$  matrix requires  $n!$  operations. For a  $12 \times 12$  matrix, that's 479,001,600 traversals. Even if you could manage to perform each traversal in just one millisecond, it would still take more than 133 hours to perform the entire traversal. A  $20 \times 20$  matrix would take 2,432,902,008,176,640,000 operations. At an optimistic millisecond per operation, that's more than 77 million years.

The Munkres algorithm runs in  $O(n^3)$  time, rather than  $O(n!)$ . This package provides an implementation of that algorithm.

This version is based on <http://www.public.iastate.edu/~ddoty/HungarianAlgorithm.html>.

This version was written for Python by Brian Clapper from the (Ada) algorithm at the above web site. (The Algorithm::Munkres Perl version, in CPAN, was clearly adapted from the same web site.)

### Usage

Construct a Munkres object:

```
from munkres import Munkres

m = Munkres()
```

Then use it to compute the lowest cost assignment from a cost matrix. Here's a sample program:

```
from munkres import Munkres, print_matrix

matrix = [[5, 9, 1],
          [10, 3, 2],
          [8, 7, 4]]
m = Munkres()
indexes = m.compute(matrix)
print_matrix(matrix, msg='Lowest cost through this matrix:')
total = 0
for row, column in indexes:
    value = matrix[row][column]
    total += value
    print '(%d, %d) -> %d' % (row, column, value)
print 'total cost: %d' % total
```

Running that program produces:

```
Lowest cost through this matrix:
[5, 9, 1]
[10, 3, 2]
[8, 7, 4]
(0, 0) -> 5
(1, 1) -> 3
(2, 2) -> 4
total cost=12
```

The instantiated Munkres object can be used multiple times on different matrices.

### Non-square Cost Matrices

The Munkres algorithm assumes that the cost matrix is square. However, it's possible to use a rectangular matrix if you first pad it with 0 values to make it square. This module automatically pads rectangular cost matrices to make them square.

Notes:

- The module operates on a *copy* of the caller's matrix, so any padding will not be seen by the caller.
- The cost matrix must be rectangular or square. An irregular matrix will *not* work.

### Calculating Profit, Rather than Cost

The cost matrix is just that: A cost matrix. The Munkres algorithm finds the combination of elements (one from each row and column) that results in the smallest cost. It's also possible to use the algorithm to maximize profit. To do that, however, you have to convert your profit matrix to a cost matrix. The simplest way to do that is to subtract all elements from a large value. For example:

```

from munkres import Munkres, print_matrix

matrix = [[5, 9, 1],
          [10, 3, 2],
          [8, 7, 4]]
cost_matrix = []
for row in matrix:
    cost_row = []
    for col in row:
        cost_row += [sys.maxint - col]
    cost_matrix += [cost_row]

m = Munkres()
indexes = m.compute(cost_matrix)
print_matrix(matrix, msg='Highest profit through this matrix:')
total = 0
for row, column in indexes:
    value = matrix[row][column]
    total += value
    print '(%d, %d) -> %d' % (row, column, value)

print 'total profit=%d' % total

```

Running that program produces:

```

Highest profit through this matrix:
[5, 9, 1]
[10, 3, 2]
[8, 7, 4]
(0, 1) -> 9
(1, 0) -> 10
(2, 2) -> 4
total profit=23

```

The `munkres` module provides a convenience method for creating a cost matrix from a profit matrix. Since it doesn't know whether the matrix contains floating point numbers, decimals, or integers, you have to provide the conversion function; but the convenience method takes care of the actual creation of the cost matrix:

```

import munkres

cost_matrix = munkres.make_cost_matrix(matrix,
                                       lambda cost: sys.maxint - cost)

```

So, the above profit-calculation program can be recast as:

```

from munkres import Munkres, print_matrix, make_cost_matrix

matrix = [[5, 9, 1],
          [10, 3, 2],
          [8, 7, 4]]
cost_matrix = make_cost_matrix(matrix, lambda cost: sys.maxint - cost)
m = Munkres()
indexes = m.compute(cost_matrix)
print_matrix(matrix, msg='Lowest cost through this matrix:')
total = 0
for row, column in indexes:
    value = matrix[row][column]
    total += value
    print '(%d, %d) -> %d' % (row, column, value)

```

```
print 'total profit=%d' % total
```

### References

1. <http://www.public.iastate.edu/~ddoty/HungarianAlgorithm.html>
2. Harold W. Kuhn. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83-97, 1955.
3. Harold W. Kuhn. Variants of the Hungarian method for assignment problems. *Naval Research Logistics Quarterly*, 3: 253-258, 1956.
4. Munkres, J. Algorithms for the Assignment and Transportation Problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32-38, March, 1957.
5. [http://en.wikipedia.org/wiki/Hungarian\\_algorithm](http://en.wikipedia.org/wiki/Hungarian_algorithm)

### Copyright and License

This software is released under a BSD license, adapted from <<http://opensource.org/licenses/bsd-license.php>>

Copyright (c) 2008 Brian M. Clapper All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name “clapper.org” nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

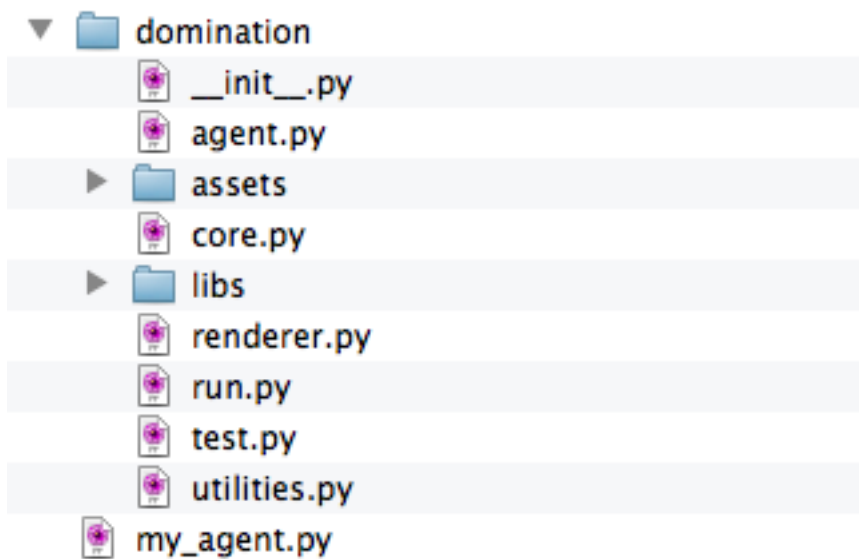
---

## Quickstart

---

If you're not going to read any of the other documentation, just do the following.

1. Copy and modify the basic agent found in the source code (`agent.py`).
2. Make sure your folder structure looks like this (you only need the *domination* module):



3. Create another file, put the following code in there, and run it:

```
from domination import core

# Setup
my_settings = core.Settings(max_steps=100)
my_game = core.Game(red='my_agent.py', blue='domination/agent.py', settings=my_settings)
# Run it
my_game.run()
```



---

## Indices and tables

---

- `genindex`
- `search`





**d**

`domination.libs.munkres`, 20  
`domination.utilities`, 13



## Symbols

`__init__()` (domination.agent.Agent method), 7

### A

`action()` (domination.agent.Agent method), 9  
`after_game()` (domination.scenarios.Scenario method), 11  
`ammo_blue` (domination.core.GameStats attribute), 5  
`ammo_red` (domination.core.GameStats attribute), 5  
`angle_fix()` (in module domination.utilities), 14

### B

`before_game()` (domination.scenarios.Scenario method), 11  
`blue` (domination.core.Game attribute), 4

### C

`CAPTURE_MODE_FIRST` (in module domination.core), 6  
`CAPTURE_MODE_MAJORITY` (in module domination.core), 6  
`CAPTURE_MODE_NEUTRAL` (in module domination.core), 6  
`compute()` (domination.libs.munkres.Munkres method), 20

### D

`deaths_blue` (domination.core.GameStats attribute), 5  
`deaths_red` (domination.core.GameStats attribute), 5  
`debug()` (domination.agent.Agent method), 9  
`domination.libs.munkres` (module), 15, 20  
`domination.utilities` (module), 13

### E

`ENDGAME_CRUMBS` (in module domination.core), 6  
`ENDGAME_NONE` (in module domination.core), 6  
`ENDGAME_SCORE` (in module domination.core), 6

### F

`FIELD` (domination.scenarios.Scenario attribute), 11  
`FieldGenerator` (class in domination.core), 12

`finalize()` (domination.agent.Agent method), 9  
`find_path()` (in module domination.utilities), 15  
`frange()` (in module domination.utilities), 13

### G

`Game` (class in domination.core), 3  
`GameStats` (class in domination.core), 4  
`generate()` (domination.core.FieldGenerator method), 13  
`GENERATOR` (domination.scenarios.Scenario attribute), 11

### L

`line_intersects_circ()` (in module domination.utilities), 14  
`line_intersects_grid()` (in module domination.utilities), 14  
`line_intersects_rect()` (in module domination.utilities), 13  
`log` (domination.core.Game attribute), 4

### M

`make_cost_matrix()` (domination.libs.munkres.Munkres static method), 19  
`make_cost_matrix()` (in module domination.libs.munkres), 20  
`make_nav_mesh()` (in module domination.utilities), 15  
`MatchInfo` (class in domination.scenarios), 11  
`mean()` (in module domination.utilities), 13  
`Munkres` (class in domination.libs.munkres), 19

### O

`observe()` (domination.agent.Agent method), 8  
`one_on_one()` (domination.scenarios.Scenario class method), 11

### P

`pad_matrix()` (domination.libs.munkres.Munkres method), 19  
`play()` (domination.core.ReplayData method), 5  
`point_add()` (in module domination.utilities), 13  
`point_dist()` (in module domination.utilities), 13  
`point_mul()` (in module domination.utilities), 13  
`point_sub()` (in module domination.utilities), 13

### R

reachable() (in module domination.utilities), 15  
rect\_contains\_point() (in module domination.utilities), 14  
rect\_corners() (in module domination.utilities), 14  
rect\_offset() (in module domination.utilities), 14  
rects\_bound() (in module domination.utilities), 14  
rects\_merge() (in module domination.utilities), 14  
red (domination.core.Game attribute), 4  
REPEATS (domination.scenarios.Scenario attribute), 11  
replay (domination.core.Game attribute), 4  
ReplayData (class in domination.core), 5  
run() (domination.core.Game method), 4

### S

Scenario (class in domination.scenarios), 10  
score (domination.core.GameStats attribute), 4  
score\_blue (domination.core.GameStats attribute), 4  
score\_red (domination.core.GameStats attribute), 4  
Settings (class in domination.core), 5  
SETTINGS (domination.scenarios.Scenario attribute), 10  
setup() (domination.scenarios.Scenario method), 11  
stats (domination.core.Game attribute), 4  
stdev() (in module domination.utilities), 13  
steps (domination.core.GameStats attribute), 5  
SWAP\_TEAMS (domination.scenarios.Scenario attribute), 11

### T

test() (domination.scenarios.Scenario class method), 11  
think\_time\_blue (domination.core.GameStats attribute), 5  
think\_time\_red (domination.core.GameStats attribute), 5  
tournament() (domination.scenarios.Scenario class method), 11