Project 2 – Robotz

In this project, you are asked to implement functions that will enable you to create, organize, and simulate communication between stationary transceivers and mobile robots.

The objective of this project is to apply object-oriented design techniques such as the representation of data along with functions using classes, implementing abstraction to hide internal data and present necessary functions for a scalable API, utilize inheritance for borrowing common features, keep track of changes in multiple instances, and in general design a fully functional software project to show what you have learned throughout the class.

Background Information

In an uncertain future, humankind wants to control the surface of the planet Omicron Persei 8, located out of the alpha centauri star system. For this purpose, transceivers and certain type of robots are designed. Transceivers located to fixed locations, and robots are mobile and should be stay connected to the transceiver towers, otherwise, if robots cannot build a duplex connection (both transmitting and receiving signals) more than 60 seconds, it is assumed that the robot unit can be (or worse it is) captured by the aliens, and the units destroy themselves due to prevent any possible information leakage.

Your task is to create a simulator to test the possible algorithms for this mission. In the planet Omicron Persei 8, the power of the electromagnetic waves used for communication decay linearly and inversely proportional to the distance from transceivers or robot units. A battlefield is composed of transceivers and the mobile units (robots).

Transceiver:

Transceivers (**trans**mitter and re**ceiver**) (can also be think as base stations) are stationary and have fixed location. Every transceiver has a different transmitting power and minimum receiving power according to used components in them. Once a transceiver set up in the battlefield, then a local/internal time counter starts to run and count every second. Every transceiver should have a unique ID to build up separate communication channels.

Robots:

While transceivers are stationary (fixed), robots can move depending on the class of the unit. Robots has their unique IDs, different than transceivers. Their transmitting power and minimum receiving powers are all same: transmitting power is 1 W and minimum receiving power is 10 mW. Since they move, they should have a velocity in meters/seconds (m/s). They also have status flag that shows dead or alive, and a counter that counts the disconnected time, if the connection is lost. According to trajectories, there are three types:

Robot : Robot units share the properties of the Transceivers. They move with a constant speed in a constant direction, in other words they have a constant velocity. The velocity information is given when the robot unit is set up in the battlefield.

Guard : Guard units share the properties of the Robots. Moreover, guards move in a square trajectory and turn right with a given period. The default value for the period is set to 60 seconds.

Psycho : Psycho units share the properties of the Robots. However, their move and velocity are random, in addition, the changing frequency of the velocity is also random. The velocity a psycho can be 10 m/s ≥ $\{v_x, v_y\}$ ≥ -10 m/s.

Battlefield:

The battlefield should manage all the units, both transceivers and robots. Keep track of their status, position and other vital information related with the mission.

Transceiver class:

This is a base class that will hold the basic information about a unit, and basic methods for printing setting and getting information, and comparison.

Methods	**Explanation**

`__init__(x, y, tpower, rpower)`	`__init__` method should take x and y coordinates of the unit and transmitting and minimum receiving powers. The default value for minimum receiving power is 1 mW.
	
	`>>> t1 = Transceiver(0,0,100)`
	`>>> t2 = Transceiver(750,500,100,0.1)`

`get_coordinate_x()`	This method will **return** the x component of the location of the transceiver in meters as float or integer.
	
	`>>> t1.get_coordinate_x()`
	`0`
	`>>> t2.get_coordinate_x()`
	`750`

`get_coordinate_y()`	This method will **return** the y component of the location of the transceiver in meters as float or integer.
	
	`>>> t1.get_coordinate_y()`
	`0`
	`>>> t2.get_coordinate_y()`
	`500`

`get_coordinates()`	This method will **return** the x, y coordinates of the location of the transceiver in meters as tuple.
	
	`>>> t1.get_coordinates()`
	`(0, 0)`
	`>>> t2.get_coordinates()`
	`(750, 500)`

`get_tpower()`	This method will **return** the transmitting power of the transceiver in Watts as integer or float.
	
	`>>> t1.get_tpower()`
	`100`
	`>>> t2.get_tpower()`
	`100`

`get_rpower()`	This method will **return** the minimum receiving power of the transceiver in Watts as integer or float.
	
	`>>> t1.get_rpower()`
	`0.001`
	`>>> t2.get_rpower()`
	`0.1`

`get_id()`	This method will **return** the ID number of the transceiver as integer.
	
	`>>> t1.get_id()`
	`0`
	`>>> t2.get_id()`
	`1`

`get_localtime()`	This method will **return** the local time of the transceiver in seconds as integer.
	
	`>>> t1.get_localtime()`
	`0`
	`>>> t2.get_localtime()`
	`0`

`set_coordinates(x, y)`	This method will **set** the x, y coordinates of the transceiver in meters. The coordinates are float or integer.
	
	`>>> t1.set_coordinates(-250, 0)`
	`>>> t1.get_coordinates()`
	`(-250, 0)`

`set_transmitting_power(tpower)`	This method will **set** the transmitting power of the transceiver in Watts. Transmitting power can be float or integer.
	
	`>>> t1.set_transmitting_power(1000)`
	`>>> t1.get_tpower()`

	`1000`

`set_receiving_power(rpower)`	This method will **set** the minimum receiving power of the transceiver in Watts. Minimum Receiving power can be float or integer.
	
	`>>> t1.set_receiving_power(1.5)`
	`>>> t1.get_rpower()`
	`1.5`

`update_local_time()`	This method will **set** the local time of the transceiver. The local time should be integer in seconds.
	
	`>>> t1.update_local_time(50)`
	`>>> t1.get_localtime()`
	`50`

`distance(m)`	This method will **return** the distance of the transceiver to another unit in meters as float.
	
	`>>> t1.distance(t2)`
	`1118.033988749895`

`transmitted_power((x,y))`	This method will calculate and **return** the transmitted power from the transceiver to given coordinates. Method will take the coordinates as tuple and return the transmitted power in Watts as float or integer.
	
	`>>> t1.transmitted_power((-250,0))`
	`1000`
	`>>> t1.transmitted_power((50,25))`
	`3.3218191941495983`
	`>>>`
	`t1.transmitted_power(t2.get_coordinates()`
	`)`
	`0.8944271909999159`
	`>>>`
	`t1.transmitted_power(t1.get_coordinates()`
	`)`
	`1000`

`__eq__()`	This method will implement equal operation between two units. If the transmitted signals can be received from both sides it returns `True`, otherwise `False`.
	
	`>>> t1==t2`
	`False`

`__lt__()`	This method will implement less than operation between two units. If the transmitted signal from the first unit can be received from the other unit it returns `True`, otherwise `False`.
	
	`>>> t1<t2`
	`True`

`__gt__()`	This method will implement greater than operation between two units. If the transmitted signal from the second unit can be received from the first unit it returns `True`, otherwise `False`.
	
	`>>> t1>t2`
	`False`

`__str__()`	This method will **return** the following information in the following format as string. Note the difference of kW, W, and mW.
	
	`>>> print(t1)`
	`Class: Tower`
	`  Tower number: 0`
	`  Coordinates: <-250,0>`
	`  Transmitting Power: 1.0kW`
	`  Min. Receiving Power: 1.5W`
	`>>> print(t2)`
	`Class: Tower`
	`  Tower number: 1`
	`  Coordinates: <750,500>`
	`  Transmitting Power: 100W`
	`  Min. Receiving Power: 100.0mW`

Robot class:

This is a subclass that inherits from Transceiver class and will hold information about the robot unit.

- The Robot units have velocity data, i.e. x and y components of the velocity. The velocity of the Robot unit is constant.
- The transmitted power by the Robot unit is always 1 W.
- The minimum received power by the Robot unit is always 10 mW.
- Robot units have different IDs than Transceiver units.
- Robot units have a status flag, True for alive, False for dead.
- Robot units have a counter for measure disconnected time.
- The Robot class inherits all the methods from Transceiver class.
- The rest of the methods are given below:

Methods	**Explanation**

`__init__(x, y, vx,vy)`	`__init__` method should take x, y coordinates of the unit in meters and the x and y components of the velocity of the robot unit in meter/seconds.
	
	`>>> r1 = Robot(10,25,10,0)`
	`>>> r2 = Robot(500,750,-7.5,6.8)`

`get_velocity()`	This method will **return** the x and y components of the velocity of the robot unit in meters/seconds as float or integer.
	
	`>>> r1.get_velocity()`
	`(10, 0)`
	`>>> r2.get_velocity()`
	`(-7.5, 6.8)`

`get_status():`	This method will **return** the status of the robot unit as boolean. True for alive, False for dead.
	
	`>>> r1.get_status()`
	`True`
	`>>> r2.get_status()`
	`True`

`get_disconnet_time()`	This method will **return** the disconnected time of the robot unit in seconds as integer.
	
	`>>> r1.get_disconnet_time()`
	`0`
	`>>> r2.get_disconnet_time()`
	`0`

`set_velocity(vx,vy)`	This method will **set** the x and y components of the velocity of the robot unit in meters/seconds as float or integer.
	
	`>>> r1.set_velocity(-5,10)`
	`>>> r1.get_velocity()`
	`(-5, 10)`

`set_status(newstatus)`	This method will **set** the status of the robot unit as boolean. True for alive, False for dead.
	
	`>>> r1.set_status(False)`
	`>>> r1.get_status()`
	`False`

`update_disconnect()`	This method will increase the value of the disconnected time of the robot unit by 1 second.
	
	`>>> r1.update_disconnect()`
	`>>> r1.get_disconnet_time()`
	`1`
	`>>> r1.update_disconnect()`
	`>>> r1.get_disconnet_time()`
	`2`

`set_disconnet_time()`	This method will **set** the disconnected time of the robot unit to 0 seconds.
	
	`>>> r1.set_disconnet_time()`
	`>>> r1.get_disconnet_time()`
	`0`

`update_location()`	This method will increase the local time the robot unit by 1 and update the coordinates of the robot unit using velocity as 1 second passes.
	
	`>>> r1.update_location()`
	`>>> r1.get_coordinates()`
	`(5, 35)`
	`>>> r1.get_localtime()`
	`1`
	`>>> r1.update_location()`
	`>>> r1.get_coordinates()`
	`(0, 45)`
	`>>> r1.get_localtime()`
	`2`

`__str__()`	This method will shadow the __str__ method in Transceiver unit. This method will **return** the following information in the following format as string. Note the difference of kW, W, and mW. `>>> print(r1)`

	Class: Robot
	  Robot number: 0
	  Current Coordinates: <0,45>
	  Current Velocity: <-5,10>
	  Transmitting Power: 1W
	  Min. Receiving Power: 10.0mW
	  Status: Dead
	>>> print(r2)
	Class: Robot
	  Robot number: 1
	  Current Coordinates: <500,750>
	  Current Velocity: <-7.5,6.8>
	  Transmitting Power: 1W
	  Min. Receiving Power: 10.0mW
	  Status: Alive

Other Method Examples	>>> t3 = Transceiver(400,250,1000)
	>>> r1==r2
	False
	>>> t3==r1
	True
	>>> t3==r2
	True
	>>>
	t1.transmitted_power(r1.get_coordinates()
	)
	3.936733295894781
	>>>
	r1.transmitted_power(t1.get_coordinates()
	)
	0.003936733295894781
	>>> r1>t1
	True
	>>> r1<t1
	False
	>>> r1.get_coordinate_x()
	0
	>>> r1.get_coordinate_y()
	45
	>>> r1.get_coordinates()
	(0, 45)
	>>> r2.get_coordinates()
	(500, 750)
	>>> r1.get_tpower()
	1
	>>> r1.get_rpower()
	0.01
	>>> r1.get_id()
	0
	>>> r2.get_id()
	1

Guard class:

This is a subclass that inherits from Robot class and will hold information about the guard unit. When they setup in the battlefield, they guard a square region. They have an initial velocity and an initial local time (default is 0), then they turn right at the end of the period. The default value of the period is 60 s, and it can be updated.

- The Guard class inherits all the methods and data attributes from Robot class.
- The Guard units has a period data, which determines the turning time to right.
- Guard units are actually Robot units, therefore they should belong to the same ID cluster.
- The rest of the methods are given below:

Methods	**Explanation**

`__init__(x, y, vx,vy, period, localtime)`	`__init__` method should take x, y coordinates of the unit in meters and the x and y components of the velocity of the robot unit in meter/seconds, period in seconds, and localtime. Default value of the period is 60 s. Default value of the localtime is 0 s.
	
	`>>> r3 = Guard(40,58,10,12,120,50)`
	`>>> r4 = Guard(750,800,-10,0,100)`
	`>>> r5 = Guard(600,700,-10,10)`

`get_period()`	This method will **return** the period of the guard unit in seconds as integer.
	
	`>>> r3.get_period()`
	`120`
	`>>> r4.get_period()`
	`100`
	`>>> r5.get_period()`
	`60`

`set_period(period)`	This method will **set** the period of the guard unit in seconds as integer.
	
	`>>> r3.set_period(60)`
	`>>> r3.get_period()`
	`60`

`update_location()`	This method will shadow the same method in robot class. It will increase the local time of the guard unit by 1, check the period and local time and decide to turn right, if so update the velocity as it turns to right, and then update the coordinates of the guard unit using velocity as 1 second passes.
	
	`>>> r3.update_location()`
	`>>> r3.get_velocity()`
	`(10, 12)`
	`>>> r3.get_localtime()`
	`51`

	`>>> r3.set_period(3)`
	`>>> r3.get_period()`
	`3`
	`>>> r3.update_location()`
	`>>> r3.get_coordinates()`
	`(60, 82)`
	`>>> r3.get_velocity()`
	`(10, 12)`
	`>>> r3.get_localtime()`
	`52`
	`>>> r3.update_location()`
	`>>> r3.get_coordinates()`
	`(70, 94)`
	`>>> r3.get_velocity()`
	`(10, 12)`
	`>>> r3.get_localtime()`
	`53`
	`>>> r3.update_location()`
	`>>> r3.get_coordinates()`
	`(82, 84)`
	`>>> r3.get_velocity()`
	`(12, -10)`
	`>>> r3.get_localtime()`
	`54`

`__str__()`	This method will shadow the __str__ method in Robot class. This method will **return** the following information in the following format as string. Note the difference of kW, W, and mW.
	
	`>>> print(r3)`
	`Class: Guard`
	`  Robot number: 2`
	`  Current Coordinates: <82,84>`
	`  Current Velocity: <12,-10>`
	`  Transmitting Power: 1W`
	`  Min. Receiving Power: 10.0mW`
	`  Status: Alive`
	`>>> print(r4)`
	`Class: Guard`
	`  Robot number: 3`
	`  Current Coordinates: <750,800>`
	`  Current Velocity: <-10,0>`
	`  Transmitting Power: 1W`
	`  Min. Receiving Power: 10.0mW`
	`  Status: Alive`
	`>>> print(r5)`
	`Class: Guard`
	`  Robot number: 4`
	`  Current Coordinates: <600,700>`
	`  Current Velocity: <-10,10>`
	`  Transmitting Power: 1W`

	Min. Receiving Power: 10.0mW Status: Alive >>>

Other Method Examples	>>> r3.get_id()
	2
	>>> r4.get_id()
	3
	>>> r3.get_tpower()
	1
	>>> r3.get_rpower()
	0.01
	>>> r4.get_tpower()
	1
	>>> r4.get_rpower()
	0.01
	>>> r1==r3
	True
	>>> r3==r1
	True
	>>> r5==r2
	False
	>>> r5==t1
	False

Psycho class:

This is a subclass that inherits from Robot class and will hold information about the psycho unit. When they setup in the battlefield, they move to a random direction with a random velocity, and their velocity changes with a random period.

- The Psycho class inherits all the methods and data attributes from Robot class.
- The Psycho units has a random period data, which should be updated randomly at the end of the period. The value of the period should be in [0,100]
- The Psycho units has a random velocity, which should be updated randomly at the end of the period. x and y components of the velocity should be in [-10,10].
- Psycho units are actually Robot units, therefore they should belong to the same ID cluster.
- You can use **`random`** module to generate random velocity and random direction.
- The rest of the methods are given below:

Methods	**Explanation**

`__init__(x, y, vx,vy)`	`__init__` method should take x, y coordinates of the unit in meters.
	
	`>>> r6=Psycho(-10,60)`
	`>>> r7=Psycho(20,320)`

`update_location()`	This method will shadow the same method in robot class. It will increase the local time of the psycho unit by 1, check the period and local time, if so randomly update the velocity and the period, and then update the coordinates of the psycho unit using velocity as 1 second passes.
	
	`>>> r6.get_velocity()`
	`(9.990742922010703, -4.4504938374168095)`
	`>>> r6.get_localtime()`
	`0`
	`>>> r6.update_location()`
	`>>> r6.get_velocity()`
	`(9.990742922010703, -4.4504938374168095)`
	`>>> r6.get_localtime()`
	`1`
	`>>> r6.update_location()`
	`>>> r6.get_velocity()`
	`(9.990742922010703, -4.4504938374168095)`
	`>>> r6.get_localtime()`
	`2`
	`>>> r6.update_location()`
	`>>> r6.get_velocity()`
	`(9.990742922010703, -4.4504938374168095)`
	`>>> r6.get_localtime()`
	`3`
	`>>> r6.update_location()`
	`>>> r6.get_velocity()`

	(9.990742922010703, -4.4504938374168095)
	>>> r6.get_localtime()
	4
	>>> r6.update_location()
	>>> r6.get_velocity()
	(1.9176068977202725, 7.828466622364687)
	>>> r6.get_localtime()
	5
	>>> r6.update_location()
	>>> r6.get_velocity()
	(1.9176068977202725, 7.828466622364687)
	>>> r6.get_localtime()
	6

__str__()	This method will shadow the __str__ method in Robot class. This method will **return** the following information in the following format as string. Note the difference of kW, W, and mW.
	
	>>> print(r6)
	Class: Psycho
	  Robot number: 5
	  Current Coordinates:
	<33.79818548348336,57.85495789506214>
	  Current Velocity:
	<1.9176068977202725,7.828466622364687>
	  Transmitting Power: 1W
	  Min. Receiving Power: 10.0mW
	  Status: Alive
	>>> print(r7)
	Class: Psycho
	  Robot number: 6
	  Current Coordinates: <20,320>
	  Current Velocity: <-
	7.714931182267762,1.7294829426157765>
	  Transmitting Power: 1W
	  Min. Receiving Power: 10.0mW
	  Status: Alive

Other Method Examples	>>> r6.get_id()
	5
	>>> r7.get_id()
	6
	>>> r6.get_tpower()
	1
	>>> r6.get_rpower()
	0.01
	>>> r7.get_tpower()
	1
	>>> r7.get_rpower()
	0.01
	>>> r1==r6
	True

	>>> r7>r1
	False
	>>> r6<r2
	False
	>>> r6==t3
	True
	>>> r6>r5
	False
	>>> r6==r4
	False
	>>> r6.distance(t3)
	413.5498593406817

Battle_Field class:

This class will hold information about the battlefield, including the list of transceivers, robots, and dead robots. It advances the time and update the information of all units properly, track the global time in seconds, and if a mobile unit does not communicate more than 60 seconds, it updates the status info of the unit from alive to dead and move the unit to dead robots list. The rest of the methods are given below:

Methods	**Explanation**
`__init__()`	`__init__` method should not take any parameters, it should create an instance of the Battle_Field class. Important data for this class is global time, the list of transceivers, robots, and dead robots. Global time should start from 0. `>>> field=Battle_Field()`

`add_transceiver(x,y,tpower,rpower)`	This method should take x and y coordinates of the transceiver unit and transmitting and minimum receiving powers, then creates an instance of the Transceiver class and add it to the transceiver list. The default value for minimum receiving power is 1 mW.
	
	`>>> field.add_transceiver(0,0,10)`
	`>>> field.add_transceiver(1200,0,15)`
	`>>> field.add_transceiver(500,500,12)`

`add_robot(x, y, vx,vy)`	This method should take x, y coordinates of the robot unit in meters and the x and y components of the velocity of the robot unit in meter/seconds, then creates an instance of the Robot class and add it to the robot list.
	
	`>>> field.add_robot(10,25,10,0)`
	`>>> field.add_robot(500,750,-10,10)`
	`>>> field.add_robot(500,750,5,10)`

`add_guard(x,y,vx,vy,timer,period)`	This method should take x, y coordinates of the guard unit in meters and the x and y components of the velocity of the robot unit in meter/seconds, period in seconds, and localtime, then creates an instance of the Guard class and add it to the robot list. Default value of the period is 60 s. Default value of the localtime is 0 s.
	
	`>>> field.add_guard(40,58,10,10)`
	`>>> field.add_guard(1000,0,10,0)`
	`>>> field.add_guard(750,800,-10,0)`

`add_psycho(x, y)`	This method should take x, y coordinates of the psycho unit in meters, then creates an instance of the Guard class and add it to the robot list.
	
	`>>> field.add_psycho(-10,60)`
	`>>> field.add_psycho(20,320)`
	`>>> field.add_psycho(500,1020)`
	`>>> field.add_psycho(750,300)`
	`>>> field.add_psycho(-300,320)`
	`>>> field.add_psycho(0,0)`

`get_transceivers()`	This method is a **yield** method and will **return** the instances of the transceivers one by one as it is called.
	
	`>>> for e in field.get_transceivers():`
	`...  e.get_id()`
	`...  e.get_coordinates()`
	`...  e.get_tpower()`
	`...`
	`0`
	`(0, 0)`
	`10`
	`1`
	`(1200, 0)`
	`15`
	`2`
	`(500, 500)`
	`12`

`get_robots(self)`	This method is a **yield** method and will **return** the instances of the robots one by one as it is called.
	
	`>>> robotlist=[]`
	`>>> for e in field.get_robots():`
	`...  e.get_id()`
	`...  e.get_coordinates()`
	`...  e.get_velocity()`
	`...  e.get_tpower()`
	`...  e.get_status()`
	`...  robotlist.append(e)`
	`...`
	`0`
	`(10, 25)`
	`(10, 0)`
	`1`
	`True`
	`1`
	`(500, 750)`
	`(-10, 10)`

	1 True 2 (500, 750) (5, 10) 1 True 3 (40, 58) (10, 10) 1 True 4 (1000, 0) (10, 0) 1 True 5 (750, 800) (-10, 0) 1 True 6 (-10, 60) (1.3244743561870607, 8.184241301280395) 1 True 7 (20, 320) (2.0763702059282956, -0.9776277446454635) 1 True 8 (500, 1020) (7.539274400643535, -3.508767020834078) 1 True 9 (750, 300) (-3.0104151473410523, -3.503089191352558) 1 True 10 (-300, 320) (4.652739552333697, -9.488442857749595) 1 True 11 (0, 0) (-4.595577170637338, 9.824729446070624) 1 True

`kill_robot(robot)`	This method takes an instance of robot class, change its status to dead, moves the instance from robot list to dead robots list. `>>> field.kill_robot(robotlist[-1])`

`get_deadrobots(self)`	This method is a **yield** method and will **return** the instances of the dead robots one by one as it is called.
	
	`>>> for e in field.get_deadrobots():`
	`...  e.get_id()`
	`...  e.get_coordinates()`
	`...  e.get_velocity()`
	`...  e.get_tpower()`
	`...  e.get_status()`
	`...`
	`11`
	`(0, 0)`
	`(-8.640836566874487, -4.231309190073986)`
	`1`
	`False`
	`>>> plist=[]`
	`>>> for e in field.get_robots():`
	`...  plist.append(e.get_id())`
	`...`
	`>>> print(plist)`
	`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
	`>>>`

`remove_robot(robotid)`	This method takes the ID of a robot unit as integer and remove it from the battlefield. If the ID is not in the list of robots or the robot unit is already dead, it stops silently.
	
	`>>> field.remove_robot(4)`
	`>>>`
	`>>> plist=[]`
	`>>> for e in field.get_robots():`
	`...  plist.append(e.get_id())`
	`...`
	`>>> print(plist)`
	`[0, 1, 2, 3, 5, 6, 7, 8, 9, 10]`
	`>>>`
	`>>> plist=[]`
	`>>> for e in field.get_deadrobots():`
	`...  plist.append(e.get_id())`
	`...`
	`>>> print(plist)`
	`[11]`
	`>>>`

	`>>> field.remove_robot(12)`
	`>>>`
	`>>> plist=[]`
	`>>> for e in field.get_robots():`
	`...  plist.append(e.get_id())`
	`...`
	`>>> print(plist)`
	`[0, 1, 2, 3, 5, 6, 7, 8, 9, 10]`
	`>>>`
	`>>> plist=[]`
	`>>> for e in field.get_deadrobots():`
	`...  plist.append(e.get_id())`
	`...`
	`>>> print(plist)`
	`[11]`
	`>>>`
	`>>> field.remove_robot(11)`
	`>>>`
	`>>> plist=[]`
	`>>> for e in field.get_robots():`
	`...  plist.append(e.get_id())`
	`...`
	`>>> print(plist)`
	`[0, 1, 2, 3, 5, 6, 7, 8, 9, 10]`
	`>>>`
	`>>> plist=[]`
	`>>> for e in field.get_deadrobots():`
	`...  plist.append(e.get_id())`
	`...`
	`>>> print(plist)`
	`[11]`
	`>>>`

`remove_transceiver(trid)`	This method takes the ID of a transceiver unit as integer and remove it from the battlefield. If the ID is not in the list of transceivers, it stops silently.
	
	`>>> field.remove_transceiver(2)`
	`>>>`
	`>>> for e in field.get_transceivers():`
	`...  e.get_id()`
	`...`
	`0`
	`1`
	`>>>`

`progress_time()`	This method increases the global time by 1 s, and updates information of the all units in the battle field, e.g., location, velocity, status, etc. It should determine if a robot unit is disconnected to transceivers, count the disconnected time and kill the robot unit if it is necessary. (i.e. when the

disconnected time is more than 60 seconds.)

```
>>> for t in range(0,100):
...     field.progress_time()
...
>>> for e in field.get_robots():
...     e.get_id()
...     e.get_coordinates()
...     e.get_velocity()
...     e.get_localtime()
...
0
(1010, 25)
(10, 0)
100
3
(1020, 258)
(10, -10)
98
5
(160, 1210)
(0, 10)
100
6
(-467.98963784057054, 561.3554601689915)
(-2.1699251412935467, 2.2042403694388124)
100
7
(-355.31694453664517, 2.959241230324075)
(-9.005021674480796, 6.429716114905737)
100
9
(24.443531362925544, 825.8590014386745)
(-8.803137727443325, -0.2879545026354222)
99
10
(-424.31012125603405, 901.6672578627087)
(3.012620921307711, 0.8172815797196868)
100
>>> for e in field.get_deadrobots():
...     e.get_id()
...     e.get_coordinates()
...     e.get_velocity()
...     e.get_localtime()
...
11
(0, 0)
(-8.640836566874487, -4.231309190073986)
0
8
(513.1134371398435, 1036.0747414608418)
(0.21497437934167962, 0.2635203518170677)
```

	61 2 (845, 1440) (5, 10) 69 1 (-310, 1560) (-10, 10) 81 >>>

`__str__()`	This method will return the number of units in the following format as string.
	
	`>>> print(field)`
	`Number of transceivers: 2`
	`Number of robots (alive): 7`
	`Number of dead robots: 4`

`create_report()`	This method will **return** the global time information, number of units, then the status of transceiver, robot, and dead robot units in the following format as string.
	
	`>>> print(field.create_report())`
	`Time: 100 s`
	`Number of transceivers: 2`
	`Number of robots (alive): 7`
	`Number of dead robots: 4`
	`Class: Tower`
	`  Tower number: 0`
	`  Coordinates: <0,0>`
	`  Transmitting Power: 10W`
	`  Min. Receiving Power: 1.0mW`
	`Class: Tower`
	`  Tower number: 1`
	`  Coordinates: <1200,0>`
	`  Transmitting Power: 15W`
	`  Min. Receiving Power: 1.0mW`
	`Class: Robot`
	`  Robot number: 0`
	`  Current Coordinates: <1010,25>`
	`  Current Velocity: <10,0>`
	`  Transmitting Power: 1W`
	`  Min. Receiving Power: 10.0mW`
	`  Status: Alive`
	`Class: Guard`
	`  Robot number: 3`
	`  Current Coordinates: <1020,258>`
	`  Current Velocity: <10,-10>`
	`  Transmitting Power: 1W`
	`  Min. Receiving Power: 10.0mW`

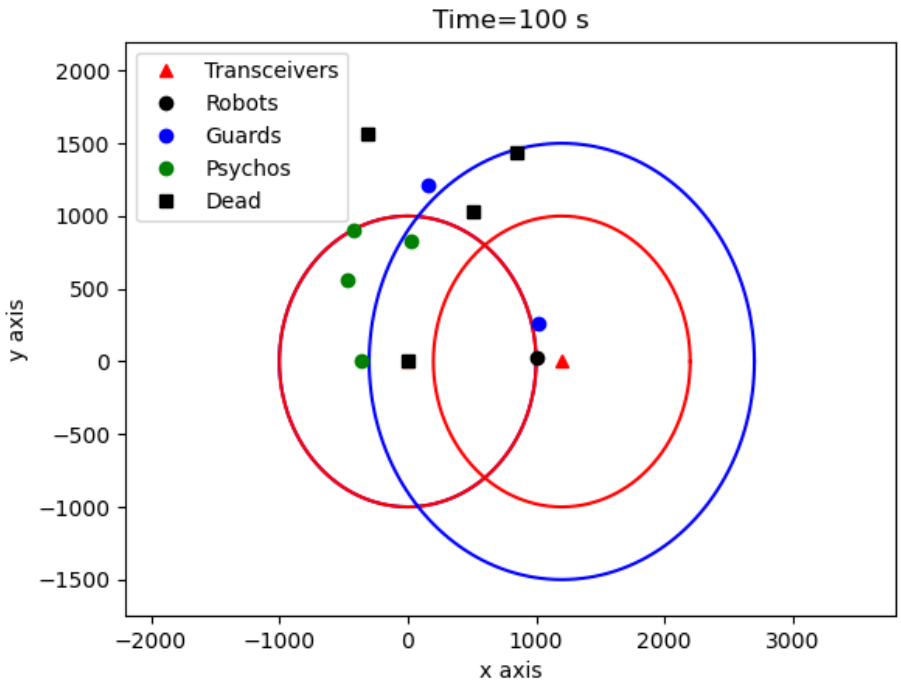
	Status: Alive Class: Guard Robot number: 5 Current Coordinates: <160,1210> Current Velocity: <0,10> Transmitting Power: 1W Min. Receiving Power: 10.0mW Status: Alive Class: Psycho Robot number: 6 Current Coordinates: <-467.98963784057054,561.3554601689915> Current Velocity: <-2.1699251412935467,2.2042403694388124> Transmitting Power: 1W Min. Receiving Power: 10.0mW Status: Alive Class: Psycho Robot number: 7 Current Coordinates: <-355.31694453664517,2.959241230324075> Current Velocity: <-9.005021674480796,6.429716114905737> Transmitting Power: 1W Min. Receiving Power: 10.0mW Status: Alive Class: Psycho Robot number: 9 Current Coordinates: <24.443531362925544,825.8590014386745> Current Velocity: <-8.803137727443325,-0.2879545026354222> Transmitting Power: 1W Min. Receiving Power: 10.0mW Status: Alive Class: Psycho Robot number: 10 Current Coordinates: <-424.31012125603405,901.6672578627087> Current Velocity: <3.012620921307711,0.8172815797196868> Transmitting Power: 1W Min. Receiving Power: 10.0mW Status: Alive Class: Psycho Robot number: 11 Current Coordinates: <0,0> Current Velocity: <-8.640836566874487,-4.231309190073986> Transmitting Power: 1W Min. Receiving Power: 10.0mW Status: Dead

	Class: Psycho
	  Robot number: 8
	  Current Coordinates:
	<513.1134371398435,1036.0747414608418>
	  Current Velocity:
	<0.21497437934167962,0.2635203518170677>
	  Transmitting Power: 1W
	  Min. Receiving Power: 10.0mW
	  Status: Dead
	Class: Robot
	  Robot number: 2
	  Current Coordinates: <845,1440>
	  Current Velocity: <5,10>
	  Transmitting Power: 1W
	  Min. Receiving Power: 10.0mW
	  Status: Dead
	Class: Robot
	  Robot number: 1
	  Current Coordinates: <-310,1560>
	  Current Velocity: <-10,10>
	  Transmitting Power: 1W
	  Min. Receiving Power: 10.0mW
	  Status: Dead

General Notes:

You should start implementing your classes one-by-one and test them individually before moving to the next class.

- **Transceiver, Robot, Guard, Psycho** classes should be very straight-forward as they are mostly similar to what we worked on in the class.
- While implementing **Psycho** class, you can use **random** module, specifically you can check the following methods
  - random.uniform(▪,▪)
  - random.randint(▪,▪)
- **You should not use any other module or library!!!**
- You can use **matplotlib.pyplot** for plotting, but do not forget to remove these lines from your submission. You can generate plots like the one given below. The red line shows the area that a transceiver can send signal, and blue shows the area that a transceiver can receive a signal from a robot unit.



- In general, you should NOT access internal class variables directly. We tried to include as many getters as possible. If you need and want, you can implement and add your own methods. These lists are the only ones that we will be testing.

Good luck, have fun!