A Simple Robot Simulator: $py_{simulator.py}$

Roy M. Turner

Fall 2022

Contents

A Simple Robot Simulator: py$_{simulator.py}$

Roy M. Turner

Fall 2022

Description

This is a very simple "robot world" simulator for use by COS 470/570 students for the search assignment. It allows you to define a rectangular world and add some obstacles and/or one or more robots. It provides a base class for robots (`Robot`) that you can subclass to create your own robots. All you have to do is create a new class based on `Robot` and define a method `agent_program` to encode your agent's intelligence. `agent_program` accepts a single argument, `percept` (see below), and it returns a string naming the next action to take (see below). You can add your gent to the simulator, then use its `run` method to run the simulation.

There are also functions available give you all of the obstacle locations for when you implement (e.g.) your A* search, as well as a very simple method to show you an overview of the current world.

Loading the simulator

First, make sure that the file `py_simulator.py` is in Python's current working directory. You may also want to use things from these files, too, and if so, put them in the current working directory, too:

- `py_symbol.py` – a "symbol" (really, unique string) generator
- `py_messages.py` – a facility for writing messages to the user

You can load these in the normal Python way using `import`. I'd suggest either something like:

```
import py_simulator as sim
```

or

```
from py_simulator import *
```

for all three, just to make your typing life easier.

Creating a simulator

To create a simulator instance, do something like:

```
s = Simulator(size=[20,20])
```

assuming that you have loaded it with "from" as above, else do something like:

```
s = sim.Simulator(size=[20,20])
```

if you used the "import. . . as" form.

The default size is 10× 10, with no obstacles, which is what you get if you just do:

```
s = Simulator()
```

You can specify obstacles in one of two ways. First, using the `num_obstacles` keyword parameter:

```
s = Simulator(num_obstacles=10)
```

which will insert (in this case) 10 obstacles in random locations. If you want them in particular places, you would use the second method and say exactly where you want them:

```
s = Simulator(obstacle_locations=[[1,1], [3,4], [10,10]])
```

Note: The world coordinates 1-based and in `[row,column]` format, starting from the top for `row` and from the left for `column`.

Example

An example "random" robot is included, implemented by class `RandomRobot`. To see it in action, do:

```
from py_simulator import *

# Create the simulator:
s = Simulator()

# Add a robot:
s.add_robot(robot_type='RandomRobot'))

# Put in some obstacles -- could have done this when instantiating Simulator,
# too:
s.add_random_obstacles(number=10)

# Display the world:
s.draw()

# Run for one "clock tick"...
s.run()
# ...and show the world again:
s.draw()

# Run for 20 ticks, showing the world each time:
s.run(20,show_each=True)
```

Creating a new robot type

To run your agent code, you'll need to create a new kind of robot and add it to the simulator. I have provided a base class for you to use, `Robot`, whose `__init__` method defines instance variables for `name`, `location`, `orientation` ($\in$ {'north', 'south', 'east', 'west'} – i.e., *world* directions), the most recent percept calculated (`percept`), the next action the agent is requesting (`next_action`), the previous action taken (`prev_action`), and the status of the last action taken (`prev_action_status` = True or False, for successful or not, respectively).

Your agent program should not access these directly, of course, since even though they are in the agent (the `Robot` instance), they are really simulation variables, not information the agent program should have. Most of your agents, for example, will not know anything about the world except as revealed via the percepts, and they will "think" in terms of forward, backward, left, right, etc., not north, south, east, or west. Your model agent will have to keep its own model of the world in those terms, for example.

Your hill-climbing agent will need a heuristic function that does know about the world, but this should be opaque to your agent program: it should call the heuristic function with local references ("what is the square in front of me worth?") and the heuristic function would map that into world coordinates as needed to determine the value to return. Thus, your heuristic function is actually a kind of simulation function or interface to the simulator rather than an agent function per se, but should be defined as a method of the `Robot` (or its subclasses you're defining).

For your uniform cost and A^* agent, you can directly use methods of `World` (in the simulator's `world` instance variable) to build whatever kind of map or representation, as well as heuristic functions, you choose to implement. See the section below about the `World` class for methods that might be useful.

To run your code, you will need to create another robot class based on `robot` and define its `agent_program` method to run your code. (In fact, you will create a different robot class for each of the parts of the assignment, most likely.) I have provided a sample robot class, `RandomRobot` (see below) to see how to do this. This robot just wanders around randomly.

You want your agent program—i.e., your AI code—to be run automatically by the simulator at each "clock tick". The simulator runs in cycles referred to here as "clock ticks". Your agent will likely be fine just using the `Robot` class' `clock_tick` function.

Your code should be implemented in your class' `agent_program` method, which is called with a percept and which should return an action's name to be done next.

For example, suppose you are writing your reflex agent. All you need to do is something like this:

```
class ReflexAgent (Robot):
    def agent_program(self,percept):
        # your brilliant code goes here; let's say
        # the action you choose to take has been
        # placed in local variable "action"; then
        # you'd do:
        return action
```

Of course, for complex agents, you wouldn't want to put all the code in `agent_program`, since that would be rather bad style (right??), so you would break it down in a nice top-down manner, defining additional methods for `agent_program` to call.

Percept format

For the search assignment, the robots have a very limited repertoire of sensors: just a forward-looking sonar-type thing that can sense what is directly in front of the robot and four bump sensors, one on each side and in the front and rear, that can detect whether or not the robot bumped into something due to the *previous* command. An agent's `clock_tick` method calculates the current percept and passes it to the `agent_program` as an argument.

The percept is a Python dictionary with an entry for each sensor. The sensors are named `front_sensor`, `front_bump`, `right_bump`, `left_bump`, and `rear_bump`. Each will have a value of `True` or `False`, for example:

```
{"front_sensor": True, "front_bump": False,
 "right_bump"=False, "left_bump": True, "rear_bump": False"}
```

which means something in front of the robot and the robot bumped into something on its left when it tried to execute the previous action.

Adding new percept components

You won't need to do this for this assignment, but in case you decide to, here's how. You can add new percept components to robots you define based on `Robot`. The `Robot` class has an instance variable, `percept_map`, that contains a dictionary of the form:

```
{"front_sensor": "forward_sensor", ...}
```

That is, each kind of sensor (e.g., `"front_sensor"`) is linked to a method (e.g., `"forward_sensor"`) that is called to give the value. You can add your own key/value pairs to this as you need to; just don't forget to define the method called!

The method is called by `calculate_percept` (see the code below), a method of `Robot`. It takes no arguments, sets the instance variable `percept`, and returns the percept as well.

Adding new actions

You may also want to add actions for the robot that are not provided by the standard `Robot` class. Actions are carried out according to the `command_map` instance variable of the robot. A command map should be a dictionary of the form:

```
{"nop": "do_nop", "forward": "do_move_forward", ...}
```

where the key is the command name and the value is the name of the method to call when that command is to be carried out. The method takes no arguments. It should return `True` if it succeeds and `False` if not. The command methods are called by the `take_action` method (see the code below), which takes care of setting `prev_action` and `prev_action_status` (based on what the command method returns).

Adding your robot to the simulator

As shown in the example above, you can add your robot to the simulator's world using (assuming `s` contains a `Simulator` instance:

```
s.add_robot(type="MyRobot")
```

which will create a new instance of `MyRobot` for you. You can instead specify an existing instance by:

```
s.add_robot(robot=my_robot)
```

where `my_robot` contains an instance of (say) `MyRobot`.

The `add_robot` method has additional parameters to allow setting the location (`location`), orientation (`orientation`), and name (`name`, which defaults to a new symbol based on `robot`). If a location or orientation is not set, then your robot's `location` and `orientation` instance variables are used (which means that if you let the simulator create the robot instance for you, it will appear at the default location specified in your class or in `Robot` (`[1,1]`)

World methods

There are various methods that you can use to access the world as needed, many of which have corresponding "pass through" methods defined in `Simulator` that just call their `World` counterpart. Here are some useful ones (see the code for their parameters and return values, as well as whether they are methods of `Simulator`, `World`, or both):

- `add_object`, `add_random_obstacle`, `add_random_obstacles` – add objects
- `add_robot` – add a robot
- `find_object` – find an object, either by location or by the object instance itself (in which case, it's just a fancy "is this object in the world?" method)
- `remove_object` to get rid of an object, either by location or by the object instance itself
- `clear` – clears the world, or the simulator and the world if you call `Simulator`'s version
- `draw` – shows a view of the world
- `empty` – check if a location is empty
- `set_drawing_character` – change the characters used when drawing the world
- `random_location`, `random_empty_location` – return a random location (the second one ensures it's empty)
- `next_location` – given a direction and an orientation, the next location in that direction; orientation is in world coordinates, so don't use this inside your agent if it shouldn't know about that
- `opposite_direction` – given a direction, returns the opposite one
- `clockwise_direction`, `counterclockwise_direction` – given a direction, returns the direction just to the clockwise/counterclockwise.
- `objects` – world method that returns a list of object instances
- `object_locations` – returns a list of locations occupied objects
- `in_bounds` – given a location, returns True/False depending on if it is in-bounds or not

I can't stress enough, however, that you **must** take care to keep the information you can get from the world out of the hands of the agents that should not have access to it (looking at you, reflex agent!).

Exceptions

Some methods raise exceptions when there is a problem so you can use Python's exception-handling facilities (e.g., `try ...  =except`) to catch errors in your code. These exceptions are:

- `WorldException` – a problem with something having to do with the world; includes subclasses:
  - `OutOfBounds` – raised (e.g.) `add_object` when you try to put something outside of the world boundaries
  - `LocationOccupied`– raised (e.g.) by `add_object` when you try to put something where there is something already
- `DirectionError` – raised (e.g.) by `next_location` if you give it a bad direction

Using the messaging methods

The file `py_messages.py` defines a class, `MessageHandler`, and the methods `msg`, `vmsg`, `dmsg`, and `vdmsg` (yes, I know what that sounds like) to allow you to control the verbosity of messages printed by your code. To use these, do something like:

```
from py_messages import *
```

and then instantiate `MessageHandler`, e.g.:

```
m = MessageHandler()
```

By default, the verbosity of output is set so that only `msg` methods produce output. You can control this by setting the verbosity of the message handler, e.g.,

```
m.set_verbosity(verb)
```

where `verb` is one of these strings:

- `'silent'` – turn off all messages
- `'normal'` – only `msg` produces output
- `'verbose'` – in addition to `msg`, `vmsg` also produced output
- `'debugging'` – in addition to the above, `dmsg` produces output
- `'verbose_debugging'` – `vdmsg` also outputs stuff at this level

As you can see in the code below, I usually define instance variables and methods of my classes to make it easier to use the message methods and to avoid dependencies on a global variable holding the `MessageHandler` instance; this also allows each object to have different verbosities, since each has their own `MessageHandler` instance. For example:

```
class MyClass():

    def __init__(self):
        self.mh = MessageHandler()

    def msg(self,m):
```

```
        self.mh.msg(m)
    def dmsg(self,m):
        self.mh.dmsg(m)
    def vmsg(self,m):
        self.mh.vmsg(m)
    def vdmsg(self,m):
        self.mh.vdmsg(m)
```

This way, from methods of `MyClass` can do:

```
self.msg('hi there')
```

Something that is very useful is (Python 3 only) string interpolation, too, e.g.:

```
self.msg(f'The objects are {self.objects()}.')
```

Simulating your work

The major function to use to run your simulation is just `run`. (Original, no?) This has two optional parameters:

- `ticks` – how many clock-ticks to run for
- `show_each` – show the state of the world after each clock tick

So if you want to run it for 10 seconds (if that's what you want clock ticks to represent, and assuming `s` contains a `Simulator` instance):

```
s.run(ticks=10,show_each=True)
```

I have provided a (very) simple way to show the world, the `draw` methods of `Simulator` and `World`. These have keyword arguments that allow you to change what characters look like, or use `set_drawing_character` to do that.

Here is an example of what the world looks like for a 10×10 world:

```
++++++++++++
+...<......+
+..........+
+..........+
+..........+
+..........+
+..........+
+..........+
+..........+
+..........+
+..........+
++++++++++++
```

#+end$_{\text{verbatim}}$ Not pretty, but functional.

The character output for each object is obtained by this method by calling each object's `icon` method, which should return a single character. The `Robot` version of this outputs a pointer-like symbol to indicate its orientation. You can change this for your agents if you like.

Code

Module setup

Here is the module setup; see above for how to load simulator. Note that this documentation is being produced from an Org Mode literate programming file that contains both Python and Lisp versions of the simulator. Feel free to ignore the Lisp code (I know you will want to!).

```
from py_symbol import *
from py_messages import *
from random import randint
```

Now create a global symbol generator for all objects to use:

```
symbolGen = SymbolGenerator()
```

Object class: Simulated objects

The `Object` class represents simulation objects, for example, obstacles. Robots and other objects can be built on this class.

```
class Object():
```

This initializes several instance variables based on the (optional, keyword) parameters to the instantiation function:

```
    def __init__(self,name=None,location=[1,1], orientation="north",icon='@'):
        self.name = name if name else symbolGen.new_symbol("obj")
        self.location = location
        self.orientation = orientation
        self.icon_char = icon
        self.world = None
        self.mh = MessageHandler()

```

Along with the `mh` instance variable, these methods allow using the messaging functions by just using other methods of the object, e.g., `self.msg('hi')` passes calls the corresponding method of `MessageHandle`.

```
    def msg(self,m):
        self.mh.msg(m)
    def dmsg(self,m):
        self.mh.dmsg(m)
    def vmsg(self,m):
        self.mh.vmsg(m)
    def vdmsg(self,m):
        self.mh.vdmsg(m)
```

Define a `clock_tick` method that is just a placeholder for those defined for subclasses.

```
    def clock_tick(self):
        pass
```

This lets `World`'s `draw` method know what this object's icon should be.

```
    def icon(self):
        return self.icon_char
```

World class

The `World` class holds a representation of the current state of the world. Before defining those, though, we first define the exception classes used by the `World` when there are problems

```
class WorldException(Exception):
    pass
class OutOfBounds(WorldException):
    pass
class LocationOccupied(WorldException):
    pass

class DirectionError(WorldException):
    pass
```

Here is the class and its `__init__` method. The world can be initialized with different sizes, numbers of obstacles automatically created in random locations, or obstacles placed at particular locations. The class variables provide some default characters to use when drawing the world. These can be overridden (see below).

```
class World():
    empty_char='.'
```

```
    side_wall_char=’+’
    top_bottom_char=’+’

    def __init__(self,size=[10,10],num_obstacles=0,
                 obstacle_locations=None):
        self.size = size
        self.num_obstacles = num_obstacles
        self.obstacle_locations = obstacle_locations

        self.objects = []

        self.mh = MessageHandler()
```

Set up messaging methods.

```
    def msg(self,m):
        self.mh.msg(m)
    def dmsg(self,m):
        self.mh.dmsg(m)
    def vmsg(self,m):
        self.mh.vmsg(m)
    def vdmsg(self,m):
        self.mh.vdmsg(m)
```

Use this method to set the drawing character(s) for the sides, top and bottom, and/or empty spaces.

```

    def set_drawing_character(self,empty=None,side_wall=None,
                              top_bottom=None):
        self.empty_char = empty if empty else World.empty_char
        self.side_wall_char = side_wall if side_wall \
            else World.side_wall_char
        self.top_bottom_char = top_bottom if top_bottom else \
            World.top_bottom_char

```

Return `True` if the location passed is empty.

```
    def empty(self,location):
        if not self.in_bounds(location):
            return False
        else:
            for object in self.objects:
                if object.location == location:
                    return False
            return True
```

Return `True` if the location passed is inside the world's boundaries.

```
    def in_bounds(self,loc):
        (x,y) = loc
        (max_x,max_y) = self.size
        return False if x < 1 or y < 1 or x > max_x or y > max_y else True
```

Add an object to the world. If you specify a location (a tuple or list), then this will insert an instance of `Object` at that location. If you pass an object (e.g., a robot, obstacle, etc.), then that will be put into the world at the location specified in its `location` instance variable.

If the location is out of bounds or the location is occupied, this raises an exception.

Note that this adds (or at least, sets) the added object's `world` instance variable so that other methods can access the world. So after this is called, a method of the object can call, e.g., `self.world.next_location([5,5],'north')` to find the location to the North of the given location.

```
    def add_object(self,object):
        if type(object) == list or type(object) == tuple:
            object = Object(location=object)

        self.vdmsg(f'(adding object {object.name} to world)')

        object.world = self                  # so it can do its own percepts

        if not self.in_bounds(object.location):
            raise OutOfBounds()
        elif not self.empty(object.location):
            raise LocationOccupied
        else:
            self.objects.append(object)
```

This clears the world of obstacles.

```
    def clear(self):
        self.vdmsg('(clearing world)')
        self.objects = []
```

This returns a list of locations at which there are objects in the world. Note that this will return any robots' locations, too. For a list all objects, use the `World` instances' `objects` instance variable directly.

```
    def object_locations(self):
        return [obj.location for obj in self.objects]
```

These two methods do the same thing: just remove an object from the world. Which object to remove can be specified either as a location (tuple or list) or as the actual object to be removed.[1]

[1] Yes, I'm aware I could have just had a class variable for `delete_object` set to `remove_object`. I just chose not to do it.

```
    def delete_object(self,object):
        return self.remove_object(object)

    def remove_object(self,object):
        object = self.find_object(object)
        if not object:
            self.vdmsg(f'(remove_object: object {object.name} not found)')
            return None
        else:
            i = self.objects.index(object)
            self.objects = self.objects[0:i] + self.objects[i+1:]
            self.vdmsg(f'(remove_object: removed {object.name})')
            return object
```

Find an object in the world and return it. If you give a location (tuple, list), then this will return the object at that location, if one is there. If you give it an object instance, it will return the object if it is in the world's list of objects—in other words, this can double as an "is this object in the world?" method.

```
    def find_object(self,description):
        if type(description) == list:
            return self.find_object_by_location(description)
        else:
            for obj in self.objects:
                if obj is description:
                    return obj
            return None

    def find_object_by_location(self,loc):
        for obj in self.objects:
            if loc == obj.location:
                return obj
        return None
```

Draw a simple depiction of the world.

```
    def draw(self):
        self.draw_line(self.top_bottom_char)
        self.draw_rows(self.empty_char,self.side_wall_char)
        self.draw_line(self.top_bottom_char)

```

```
    def draw_line(self,char):
        print((self.size[1]+2)*char)

    def draw_rows(self,empty,wall):
        for i in range(self.size[0]):
            print(wall,end='')
            self.draw_row(i+1,empty)
            print(wall)

    def draw_row(self,row,empty):
        for col in range(self.size[1]):
            obj = self.find_object([row,col+1])
            if obj:
                print(obj.icon(),end='')
            else:
                print(empty,end='')

```

This returns a random empty location in the world.

The method could be improved, since it just tries to find an empty location randomly, and returns if it hasn't found one after trying once for every location in the world—so there are times it may not find one, even if one is available. We could (should?) change this to first make a list of all empty location, then return a random element of that list, thus guaranteeing we find one. The trade-off is time: for sparsely-populated, large worlds, this will be much quicker.

```
    # return empty location
    def empty_location(self):
        for i in range(self.size[0]*self.size[1]):
            loc = [randint(1,self.size[0]),randint(1,self.size[0])]
            if self.empty(loc):
                return loc
        self.dmsg('No empty squares found after row*column tries.')
        return None

```

These methods: find the next location in the given orientation; find the direction opposite the one given; and find the direction just to clockwise or counterclockwise of the given direction. If you give one of them an invalid direction, they will raise an exception.

```
    # Note: we're going w/ row,column rather than x,y now:
    def next_location(self,location,direction):
        if direction == 'north':
            return [location[0]-1,location[1]]
```

```
        elif direction == 'south':
            return [location[0]+1,location[1]]
        elif direction == 'east':
            return [location[0],location[1]+1]
        elif direction == 'west':
            return [location[0],location[1]-1]
        else:
            raise DirectionError()

    def opposite_direction(self,direction):
        if direction == 'north':
            return 'south'
        elif direction == 'south':
            return 'north'
        elif direction == 'east':
            return 'west'
        elif direction == 'west':
            return 'east'
        else:
            raise OrientatioError()

    def clockwise_direction(self,direction):
        if direction == 'north':
            return 'east'
        elif direction == 'south':
            return 'west'
        elif direction == 'east':
            return 'south'
        elif direction == 'west':
            return 'north'
        else:
            raise DirectionError()

    def counterclockwise_direction(self,direction):
        return self.opposite_direction(self.clockwise_direction(direction))

```

This allows you to set the location for the object by calling the corresponding method of `World`.

```
    def set_drawing_character(self,empty=None,side_wall=None,
                              top_bottom=None):
        self.world(set_drawing_character(empty=empty,side_wall=side_wall,
                                         top_bottom=top_bottom))

```

Simulator class

This is the class that represents the simulator itself. It creates and contain an instance of `World`. You can set the world's size and initial obstacle content by passing the appropriate parameters to the instantiation as well; for details, see `World`'s `__init__` method.

```
class Simulator():
    def __init__(self,size=[10,10],num_obstacles=0,obstacle_locations=None):
        self.time = 0
        self.world = World(size=size,num_obstacles=num_obstacles,
                           obstacle_locations=obstacle_locations)
        self.mh = MessageHandler()
```

Set up messaging methods for this object.

```
    def msg(self,m):
        self.mh.msg(m)
    def dmsg(self,m):
        self.mh.dmsg(m)
    def vmsg(self,m):
        self.mh.vmsg(m)
    def vdmsg(self,m):
        self.mh.vdmsg(m)

```

Clear the world (`clear`) or clear the world and reset the timer (`reset`).

```
    def clear(self):
        self.world.clear()
        self.msg('Cleared.')

    def reset(self):
        self.clear()
        self.time = 0
```

Methods for adding objects.

`add_obstacles` just calls `add_objects`, which calls `World`'s `add_object` method for each object specified (see that method for details about object specification).

`add_random_obstacles` adds multiple obstacles in random locations. You can specify the number to add, the maximum to add, and the minimum to add. If you don't specify a number, this creates a random number (between the minimum and the maximum, inclusive) of obstacles.

```
    def add_obstacles(self,loc_list):
        return self.add_objects(loc_list)

    # "loc_list" can be a list of locations or actual object instances:
    def add_objects(self,loc_list):
        for loc in loc_list:
            self.world.add_object(loc)

    def add_object(self,loc_or_obj):
        return self.world.add_object(loc_or_obj)

    def add_random_obstacles(self,number=None,max=20,min=1):
        if number == None:
            number = randint(min,max)
        for i in range(number):
            self.add_random_obstacle()

    def add_random_obstacle(self):
        self.world.add_object(self.world.empty_location())

    def add_robot(self,robot=None,name=None,location=None,orientation=None,
                  robot_type=’Robot’):
        if location and not self.empty(location):
            self.msg(f"Can’t add robot at {location}: not empty or out of bounds.")
            return False
        if robot is None:
            robot = eval(f’{robot_type}()’)
            robot.location = location if location else self.world.empty_location()
            robot.orientation = location if location else directions[randint(0,3)]
        else:
            if location:
                robot.location = location
            if orientation:
                robot.orientation = orientation

        self.dmsg(f’Adding robot {robot.name} at {robot.location}, orientation {robot.orientation}’)
        return self.add_object(robot)
```

These are methods that just call their counterparts of `World`; see the description for those methods.

```
    def find_object(self,description):
        return self.world.find_object(description)
```

```
    def delete_object(self,object):
        self.world.delete_object(object)

    def remove_object(self,object):
        self.world.delete_object(object)

    def random_location(self):
        return [randint(1,self.world.size[0]),randint(1,self.world.size[1])]

    def random_empty_location(self):
        self.world.empty_location()


    def draw(self,empty_char='.',side_wall_char='+',top_bottom_char='+'):
        self.world.draw()
```

This runs the simulator. By default, it runs for a single "clock tick" and does not draw the world. You can set `ticks` to the number of ticks you would like it to run, and you can set `show_each` to `True` to have it draw the world after each clock tick.

```
    def run(self,ticks=1,show_each=False):
        self.msg(f'Running for {ticks} ticks.')
        for i in range(ticks):
            self.clock_tick()
            if show_each:
                self.draw()

```

This just calls each object's `clock_tick` method, then increments the simulated time.

```
    def clock_tick(self):
        self.dmsg('.')
        for object in self.world.objects:
            object.clock_tick()
        self.time += 1
```

Robot class

This is the base class you should use for your agents.

```
class Robot(Object):
```

The commands and percepts the `Robot` knows about are defined as class variables, which `__init__` then copies to corresponding instance variables if no different ones are specified when the object is instantiated. These are described above.

```
    command_map = {"nop": "do_nop",
                   "forward": "do_move_forward",
                   "backward": "do_move_backward",
                   "left": "do_move_left",
                   "right": "do_move_right",
                   "turn_right": "do_turn_clockwise",
                   "turn_left": "do_turn_counterclockwise"}

    percept_map = {"front_sensor": "forward_sensor",
                   "front_bump": "front_bump_sensor",
                   "rear_bump": "rear_bump_sensor",
                   "right_bump": "right_bump_sensor",
                   "left_bump": "left_bump_sensor"}

```

You can specify the location, orientation, name, and the command and percepts the robot will have here. By default, the class variables for the commands and percepts are used, the location is [1,1], and the robot is oriented toward North. If name is not given `'robot'` is used as the base, with the first robot being named ='robot1', etc.

```
    def __init__(self,command_map=None,percept_map=None,
                 location=[1,1],orientation='north',
                 name=None):
        super().__init__(location=location, orientation=orientation)
        self.percept = None
        self.next_action = None
        self.prev_action = None
        self.prev_action_success = None

        self.command_map = command_map if command_map else \
            Robot.command_map
        self.percept_map = percept_map if percept_map else \
            Robot.percept_map

        self.name = name if name else symbolGen.new_symbol('robot')


```

This is called by `clock_tick` to calculate the agent's current percept; it sets the `percept` instance variable accordingly, as well as returning the percept.

```
    def calculate_percept(self):
        percept = []
        for sensor in self.percept_map:
```

```
            func = self.percept_map[sensor]
            self.vdmsg(f'(calculate_percept({self.name}): calculating {sensor} value)')
            percept.append([sensor, eval(f'self.{func}()')])
        self.percept = percept
        return percept
```

Set the icon used; called by `World`'s `draw` function. The icon is meant to indicate the orientation.

```

    def icon(self):
        if self.orientation == 'north':
            return '^'
        elif self.orientation == 'south':
            return 'v'
        elif self.orientation == 'east':
            return '>'
        elif self.orientation == 'west':
            return '<'
        else:
            return '?'

```

The `clock_tick` method calculates the percept, calls the agent program, then takes the action requested.

```

    def clock_tick(self):
        self.calculate_percept()
        self.next_action = self.agent_program(self.percept)
        self.take_action()
        return True

```

This is a placeholder `agent_program`—by default, since `Robot` isn't meant to really do anything by itself, it just always requests no operation (`'nop'`).

```

    def agent_program(self,percept):
        self.msg(f'{self.name}: Dummy agent_program({percept}) called.')
        return 'nop'

```

Here are the default sensor methods. The standard ones supplied provide the outputs of the forward sensor and bump sensors.

```
    def forward_sensor(self):
        if self.world.empty(self.world.next_location(self.location,
                                                      self.orientation)):
            return False
        else:
            return True

    def front_bump_sensor(self):
        return self.bump_sensor('forward',self.orientation)
    def rear_bump_sensor(self):
        return self.bump_sensor('backward',self.world.opposite_direction(self.orientation))
    def left_bump_sensor(self):
        return self.bump_sensor('left', self.world.counterclockwise_direction(self.orientation))
    def right_bump_sensor(self):
        return self.bump_sensor('right', self.world.clockwise_direction(self.orientation))

    def bump_sensor(self,which,direction):
        return self.prev_action == which and \
            not self.prev_action_success and \
            not self.world.empty(self.world.next_location(self.location, direction))

    ## Action methods:
    def take_action(self):
        if not self.next_action in self.command_map:
            self.msg(f'take_action for {self.name}: unknown action {self.next_action}; ' + \
                'doing nothing')
            self.next_action = "nop"
            self.prev_action_success = False
        else:
            method = self.command_map[self.next_action]
            self.msg(f'{self.name}: Performing action {self.next_action}')
            self.dmsg(f'(take_action: calling method {method})')
            self.prev_action_success = eval(f'self.{method}()')

        self.prev_action = self.next_action
        self.next_action = None
        return self.prev_action_success
```

These are the methods that are called to accomplish the commands `agent_program` requests. See above for a description of what they do.

```

    ## actions implementation:
    def do_nop(self):
        return True
```

```

    def do_move_forward(self):
        world = self.world
        return self.move(world.next_location(self.location,self.orientation))

    def do_move_backward(self):
        world = self.world
        return \
            self.move(world.next_location(self.location,
                                          world.opposite_direction(self.orientation)))

    def do_move_left(self):
        world = self.world
        return \
            self.move(world.next_location(self.location,
                                          world.counterclockwise_direction(self.orientation)))

    def do_move_right(self):
        world = self.world
        return \
            self.move(world.next_location(self.location,
                                          world.clockwise_direction(self.orientation)))

    def move(self,location):
        if not self.world.empty(location):
            self.msg(f'{self.name}: Tried and failed to move to {location}.')
            return False
        else:
            self.location = location
            self.msg(f'{self.name} Moving to {location}.')
            return True

    def do_turn_clockwise(self):
        self.orientation = self.world.clockwise_direction(self.orientation)
        self.msg(f'{self.name}: Turning right to {self.orientation}.')
        return True

    def do_turn_counterclockwise(self):
        self.orientation = self.world.counterclockwise_direction(self.orientation)
        self.msg(f'{self.name}: Turning left to {self.orientation}.')
        return True
```

`create_simulator` function

A function is provided to create a simulator, but really, just instantiating the `Simulator` class is just as good.

```
def create_simulator(size=[10,10],num_obstacles=0,obstacle_locations=None):
    return Simulator(size=size,num_obstacles=num_obstacles,obstacle_locations=obstacle_locations)
```

Example: `RandomRobot`

Here is an example to help you figure out how to set up your agents. This one is **not** one of the ones you will create, but rather just wanders around the world.

```
class RandomRobot(Robot):
    def __init__(self,command_map=None,percept_map=None,
                 location=[1,1],orientation='north',
                 name=None):
```

This calls the `Robot` class' `__init__` method to have it set up most of the robot for you.

```
        super().__init__(command_map=command_map, percept_map=percept_map,
                                    location=location, orientation=orientation,
                                    name=symbolGen.new_symbol('randrob'))
```

An example agent program. It also shows how you can use the variable verbosity messaging code from inside methods of your robot.

```
    def agent_program(self,percept):
        # Just wander around:
        keys = list(self.command_map.keys())
        self.next_action = keys[randint(0,len(keys)-1)]

        # here is how you can use msg, dmsg, etc.:
        self.dmsg(f'{self.name}: next action={self.next_action}.')

        return self.next_action              # must do this!!
```