MSR stands for **M**inimal **S**ufficient **R**efinements and for **M**inimal **S**tructural **R**epresentations

We want to create a library called `msrgym` which has sublibraries corresponding to different types of agents in different types of environments. For example `msrgym.robot_arm` and `msrgym.bot_in_a_maze`. Each of them has a subsublibrary `.ext` and `.int` (external and internal). I am thinking of such a "subsublibrary" as a class. Let us focus on the robot arm, because that's the first one to be implemented.

Robot arm: External

Initialization The class `msrgym.robot_arm.ext` is initialized with the following parameters:

1. n = number of joints. An integer > 0,
2. L = arm lengths. A list of floats. The length of this list is n.
3. O = obstacles.
4. p = initial position of the arm, a sequence of integers between 0 and 359. The length of this sequence is n.
5. `get_S()` = sensory distribution. This is a function which returns an integer. The answer cannot be different for the same values of n, L, O, p. The idea is that the sensory data depends on the (actual) position of the arm.

How to present obstacles? The simplest obstacle is a disk. It is given by a center point and a radius. A more complicated one is presented as a closed polygon. A polygon can be presented as a sequence points in a plane, i.e. pairs of floats. Assuming that the first joint of the robot arm is at $(0, 0)$. For example

O = `[[[1,1],[2,1],[2,1]],[[-1,-1],[-3,-1],[-2,-1]]]`

would be a set of two obstacles each of which is a triangle near the origin.

The initial position p_0 must be such that the arm doesn't touch the obstacles, otherwise an error should be thrown. This means that the piecewise linear curve which starts from $(0, 0)$ and which is defined by L and p_0 together doesn't intersect any of the polygonal areas defined in O.

Functions.

1. `_hit_obstacle(p_input)` this returns `True`, if the position defined by `p_input` is such that the arm intersects one of the polygonal obstacles. Otherwise return `False`.
2. `update(a)`. This function takes as an input an action a. The action a is an integer such that $0 \le a < 2n$. If $a = 2k$ for some k, then the k:th joint moves counterclockwise, and if $a = 2k + 1$ for some k, then the k:th joint moves clockwise. The move is successfull if no obstacle is in the way, otherwise the arm stays still:
```
p_new := p.copy()
if a is even:
  k := a/2
  p_new[k] := (p_new[k]+1) mod 360
else:
  k := (a-1)/2
  p_new[k] := (p_new[k]-1) mod 360
if not self._hit_obstacle(p_new):
  p := p_new
```
   The output of this function is `None` (or no output).
3. `get_sensory_data()`. Function with no input. The output is given by `get_S()`.

4. `get_position()`. This function returns p and also the actual geometry of the arm, i.e. the coordinates of the joints. This can be computed using p and L, assuming that the first joint is in the origin $(0, 0)$.

Note that `_hit_obstacle` is only an internal function which needs never be called from outside. From outside we only call `update`, `get_sensory_data`, and `get_position`.

Robot arm: Internal

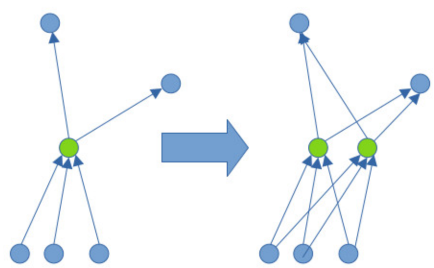
Initialization The class `msrgym.robot_arm.int` is initialized with the following parameters:

1. G = A transition multigraph given as a square matrix whose entries are sets of actions (integers between 0 and $2n$ where n should match the n above). Initialize as a (1×1)-matrix with the (only) set of actions being the set of all actions.

2. Current state m which is an integer between 0 and $d-1$ where d is the size of G, i.e. the dimensions of the matrix are $d \times d$. Initialized as 0.

Functions.

1. `split(n)` this function changes the transition graph G by splitting a node into two nodes. All the incoming and outgoing transitions are dublicated. An illutsration of the change in the graph:



In terms of the transition matrix this means that the number of both columns and rows is increased by one and the new column is a copy of the n:th column and the new row is a copy of the n:th row. The element a_{dd} is the same as a_{nn}. For example, if the original matrix is

$$A = \begin{bmatrix} \{0,1\} & \{1,2,3,4\} \\ \{1,3,4\} & \{0,2\} \end{bmatrix}$$

and the input is $n = 0$, the new matrix is

$$A = \begin{bmatrix} \{0,1\} & \{1,2,3,4\} & \{0,1\} \\ \{1,3,4\} & \{0,2\} & \{1,3,4\} \\ \{0,1\} & \{1,2,3,4\} & \{0,1\} \end{bmatrix}$$

2. `merge(n,m)` Opposite of `split`. Remove columns n, m and rows n, m and add instead one row and one column which are obtained by taking unions of elements of original rows/columns. Thus, if the matrix is

$$A = \begin{bmatrix} \{1\} & \{2,4\} & \{2\} \\ \{\} & \{\} & \{1\} \\ \{0\} & \{\} & \{\} \end{bmatrix}$$

and the input is $(n, m) = (0, 1)$, then the output is

$$A = \begin{bmatrix} \{1,2,4\} & \{1,2\} \\ \{0\} & \{\} \end{bmatrix}$$

3. `add(n,m,k)` add a connection from n to m with label k, meaning that $a_{nm} := a_{nm} \cup \{k\}$.

4. `del(n,m,k)` remove the connection from n to m, if it has label k, meaning that $a_{nm} := a_{nm} \setminus \{k\}$.

5. `transition(k)` This is a non-deterministic transition. If the state currently is n, then we go to one of the states in the set

$$\{m \mid k \in a_{nm}\},$$

meaning we go to one of the states to which there is a connection from n with label k. We can pick it randomly.