Collision checking

As part of this exercise, you will write your very own collision checker. While this checker will only function in-simulation, it should give you a good idea of the complexity associated with detecting collisions in the real world.

We have defined a few data structures that you will use in this task.

Data structures and protocol

The data structures are defined in the `dt-protocols-daffy` package.

In particular, you can look in `collision_protocol.py` the data structures to use.

We **strongly** suggest opening the `collision_protocol.py` link/file in a separate window, and cross-referencing the information given here with the code definition given in the file.

The parameters for the collision checker is a `MapDefinition`, which specifies the `environment` and `body`. The `environment` is all of the shapes that the robot can collide with, and the `body` is all of the shapes that make up the robot's body. Therefore, both `environment` and `body` are lists of `PlacedPrimitive`s. However, in the validation tests, the robot list will only contain one `PlacedPrimitive`.

A `PlacedPrimitive` is a pair of a `FriendlyPose` and a `Primitive`, or a pose and a shape. Note that `theta_deg` in `FriendlyPose` starts at zero in the positive x-axis direction and ends at 359 degrees, moving in a counter-clockwise direction.

```
@dataclass
class PlacedPrimitive:
    pose: FriendlyPose
    primitive: Primitive


@dataclass
class FriendlyPose:
    x: float
    y: float
    theta_deg: float
```

A `FriendlyPose` is a handy pose representation containing a (x, y) coordinate along with an angle. How friendly!

A `Primitive` is either a `Rectangle` or a `Circle`. A circle's shape needs only be defined by a `radius`, while a Rectangle is defined by four values:

- `xmax` is the distance from the pose point to its side in the positive x direction (if theta_deg in `FriendlyPose` is zero, this side is on the right of the pose point).
- `xmin` is the same, but in the negative x direction
- `ymax` is the distance from the center point to its side in the positive y direction (if theta_deg in `FriendlyPose` is zero, this side is on the top of the pose point).
- `ymin` is the same, but in the negative y direction

`xmax`, `xmin`, `ymax`, and `ymin` are all given with respect to the robot/obstacle's coordinate system, and not the world coordinate system. Therefore, the theta_deg value of the `FriendlyPose` affects the rotation of the Rectangle.

```
@dataclass
class Circle:
    radius: float

@dataclass
class Rectangle:
    xmin: float
    ymin: float
    xmax: float
    ymax: float

Primitive = Union[Circle, Rectangle]
```

So, we represents shapes as the union of rototranslated `Rectangle`s and `Circle`s.

The class `CollisionChecker` in `collision_checker.py` first receives a message `MapDefinition` to define the environment and robot shape. It also contains a default pose for the robot (0, 0). Then, it recieves a sequence of `CollisionCheckQuery`s. The query contains a new pose for the robot, which you will need to cross-reference against the `MapDefinition` to detect collisions.

Therefore, the `CollisionChecker` must take the new pose from the `CollisionCheckQuery`, combine it with the default pose already contained the the robot's `PlacedPrimitive`s, and then see if it collides with any part of the environment. The result of this will go into a `CollisionCheckResult`. The `CollisionCheckResult` contains only a boolean: true means that it is in collision, false means that it is not in collision.

Tips for implementing the collision checker

There are multiple ways to implement the collision checker. Here are some tips, but feel free to follow your intuition.

Use decomposition

The first thing to note is that the problem can be *decomposed*.

You are asked to see whether the robot collides with the environment at a certain pose. The robot is a list of `PlacedPrimitive` s and the environment is a list of `PlacedPrimitive` s. Remember, a `PlacedPrimitive` is the combination of a pose and a primitive, or in other terms, a location and a shape. In pseudocode:

```
robot =  rp1 ∪ rp2 ∪ rp3 ∪ ...
environment =  obj1 ∪ obj2 ∪ obj3 ∪ ...
```

What you have to check is whether the intersection

```
robot ∩ environment
```

is empty. By substituting terms we obtain:

```
(rp1 ∪ rp2 ∪ ...) ∩ (obj1 ∪ obj2 ∪ ...)
```

Now, the intersection of unions is a union of intersection:

```
[rp1 ∩ (wc1 ∪ wc2 ∪ ...)]  ∪  [rp2 ∩ (wc1 ∪ wc2 ∪ ...)] ∪ ...
```

The above shows that you have to check whether any primitive of the robot collides with environment.

Further expanding the first term we obtain:

```
[rp1 ∩ (obj1 ∪ obj2 ∪ ...)] = (rp1 ∩ obj1) ∪ (rp2 ∩ obj2) ∪
...
```

This shows that in the end, you can reduce to problem to checking pairwise intersection of `PlacedPrimitives` . Therefore, using *decomposition*, we have simplified the problem of "Does the robot collide with the environment?" to asking "Does this part of the robot collide with this environmental object?". We ask this second question multiple times for each query. If the answer to this second question is ever yes, then we know that the robot collides with the environment.

This tip has already been partially implemented in `collision_checker.py` . In other words...

```
for each environment_shape in env:
    for each robot_part in robot:
        if collides:
            return True
return False
```

Pay attention to the poses

Both robot and environment are lists of `PlacedPrimitives`.

That is, we should rewrite the robot expression as:

```
robot = RT(pose1, primitive1) ∪ RT(pose2, primitive1) ∪ ...
```

where `RT()` rototranslates a primitive by a pose. Also note that for each query the robot changes pose. Let's call this pose `Q`. Note that we have:

Note that we have

```
robot at pose Q = RT(Q ∗ pose1, primitive1) ∪ RT(Q ∗ pose2,
primitive1) ∪ ...
```

where `Q ∗ pose` represent matrix multiplication.

The above says that you can "push inside" the global pose.

In the end, what is the core complexity?

Following the above tips, you should be able to get to the point where you are left with checking the collision of two rototranslated primitives.

Now notice that there are 3 cases:

- `Circle` vs `Circle`
- `Rectangle` vs `Circle`
- `Rectangle` vs `Rectangle`

Note that without loss of generality you can get to the point where you have one primitive at the origin (You put one primitive in the coordinate frame of the other). How would you go about it?

`Circle` vs `Circle` is easy: two circles intersects if the distance of the centers is less than the sum of the radii. (The validation tests don't actually ever use a circle shape on a robot, so this case may seem unncessary, but it's useful to leave it in for learning purposes).

For the others, you have to think about it... Use your robotic mind to engineer a solution!

Speeding things up using lower/upper bound heuristics

If you want to speed things up, consider the following method, which allows to introduce a fast heuristic phase using only circle-to-circle comparisons.

For each rectangle `R`, you can find `C1`, the largest circle that is contained in the rectangle, and `C2`, the smallest circle that contains the rectangle. These are an upper

bound and a lower bound to the shape.

```
C1 ⊆ R ⊆ C2
```

Now notice that:

- if `C1` collides with a shape, also `R` does. (but if it doesn't you cannot conclude anything)
- if `C2` does not collide with a shape, `R` does not as well. (but if it does, you cannot conclude anything)

Using this logic, you can implement a method that first checks quickly whether the circle approximations give already enough information to conclude collision/no-collision. Only if the first test is inconclusive you go to the more expensive component.

Speeding things up using bitmaps heuristics

Another approach is using bitmaps to convert the environment to an image, where a black pixel means "occupied", and a white pixel means "free".

Then you can do the same with the robot shape and obtain another bitmap.

Then you check whether the two bitmaps intersect

Advantages:

- reduces the problem of collision to drawing of shapes;
- cheaper if shapes are very complex.

Disadvantages:

- There are subtle issues regarding the approximations you are making. What exactly does a pixel represent? is it a point, or is it an area? is this an optimistic or pessimistic approximation? The semantics of painting is unclear.