

## Chapter 7

### Creating repeating patterns of parts/models – For loops

In the previous chapter you used if statements to control whether some part of your design should be created or not. In this chapter you are going to find out how you can create multiple parts or objects when they form a specific pattern.

Given the following car model as an example you are going to learn how to create such patterns.

use <vehicle\_parts.scad>

```
$fa=1;
```

```
$fs=0.4;
```

```
// Variables
```

```
track = 30;
```

```
wheelbase=40;
```

```
// Body
```

```
body();
```

```
// Front left wheel
```

```
translate([-wheelbase/2,-track/2,0])rotate([0,0,0])simple_wheel();
```

```
// Front right wheel
```

```
translate([-wheelbase/2,track/2,0])rotate([0,0,0])simple_wheel();
```

```
// Rear left wheel
```

```
translate([wheelbase/2,-track/2,0])rotate([0,0,0])simple_wheel();
```

```
// Rear right wheel
```

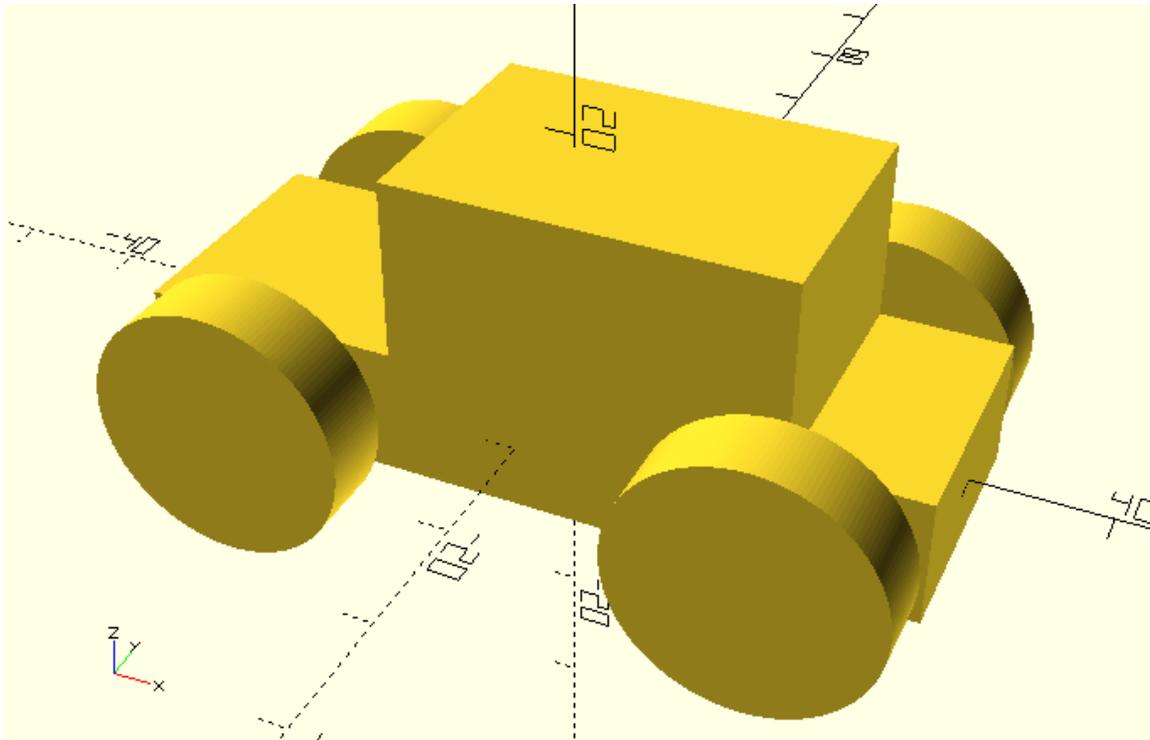
```
translate([wheelbase/2,track/2,0])rotate([0,0,0])simple_wheel();
```

```
// Front axle
```

```
translate([-wheelbase/2,0,0])axle(track=track);
```

```
// Rear axle
```

```
translate([wheelbase/2,0,0])axle(track=track);
```



Take the above car example and modify it in order to create another car. To avoid duplicating the code that creates the car you should create a car module. The module should have two input parameters, the track and the wheelbase of the car. The default value of the track and the wheelbase should be 30 and 40 units respectively. The first car should be positioned at the origin as the example above, the second car should be translated along the positive direction of the Y axis by 50 units.

```
use <vehicle_parts.scad>
```

```
$fa=1;
```

```
$fs=0.4;
```

```
module car(track=30, wheelbase=40) {
```

```
    // Body
```

```
    body();
```

```
    // Front left wheel
```

```
    translate([-wheelbase/2,-track/2,0])rotate([0,0,0])simple_wheel();
```

```
    // Front right wheel
```

```
    translate([-wheelbase/2,track/2,0])rotate([0,0,0])simple_wheel();
```

```

// Rear left wheel
translate([wheelbase/2,-track/2,0])rotate([0,0,0])simple_wheel();

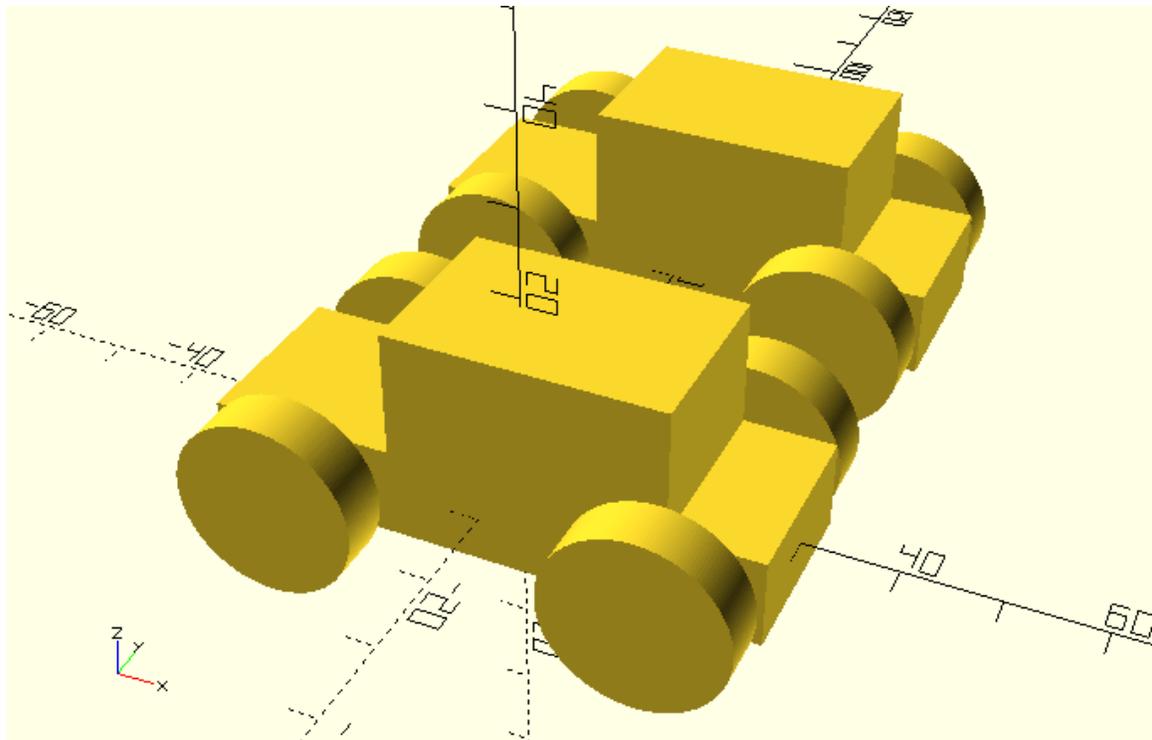
// Rear right wheel
translate([wheelbase/2,track/2,0])rotate([0,0,0])simple_wheel();

// Front axle
translate([-wheelbase/2,0,0])axle(track=track);

// Rear axle
translate([wheelbase/2,0,0])axle(track=track);
}

car();
translate([0,50,0])car();

```

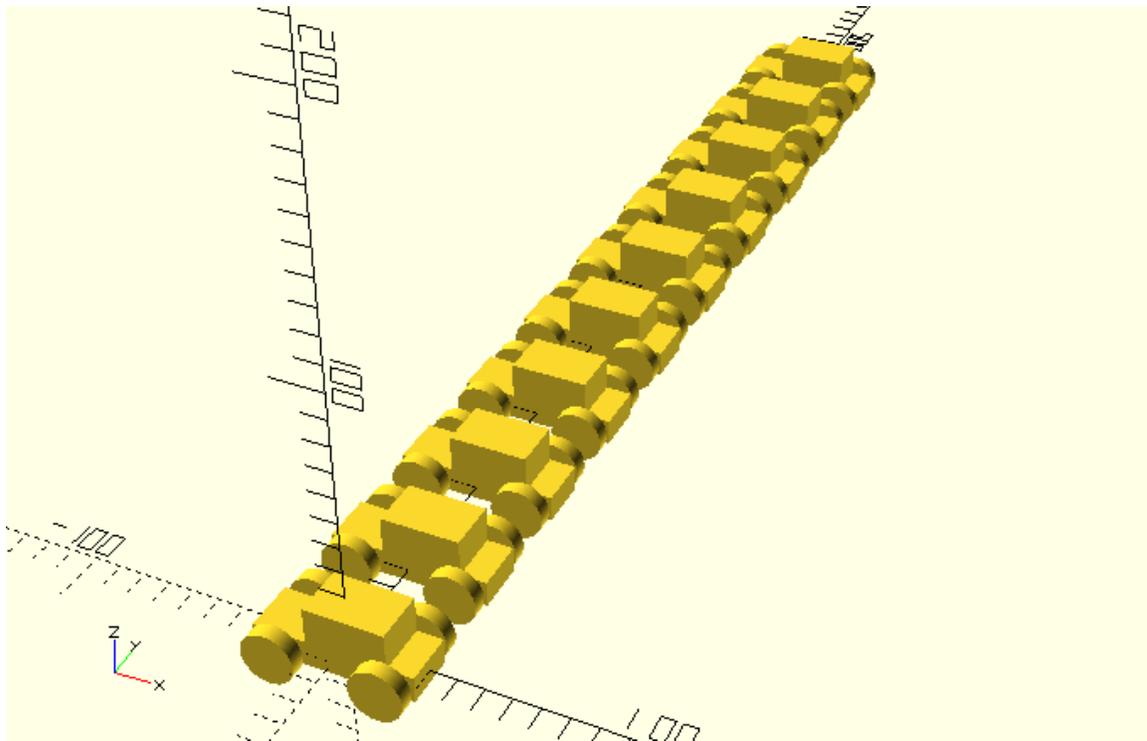


Create eight additional cars along the positive direction of the Y axis so that you have ten cars in total. Every next car should be translated 50 units along the positive direction of the Y axis in comparison to the previous car.

...

```
car();  
translate([0,50,0])car();  
translate([0,100,0])car();  
translate([0,150,0])car();  
translate([0,200,0])car();  
translate([0,250,0])car();  
translate([0,300,0])car();  
translate([0,350,0])car();  
translate([0,400,0])car();  
translate([0,450,0])car();
```

...



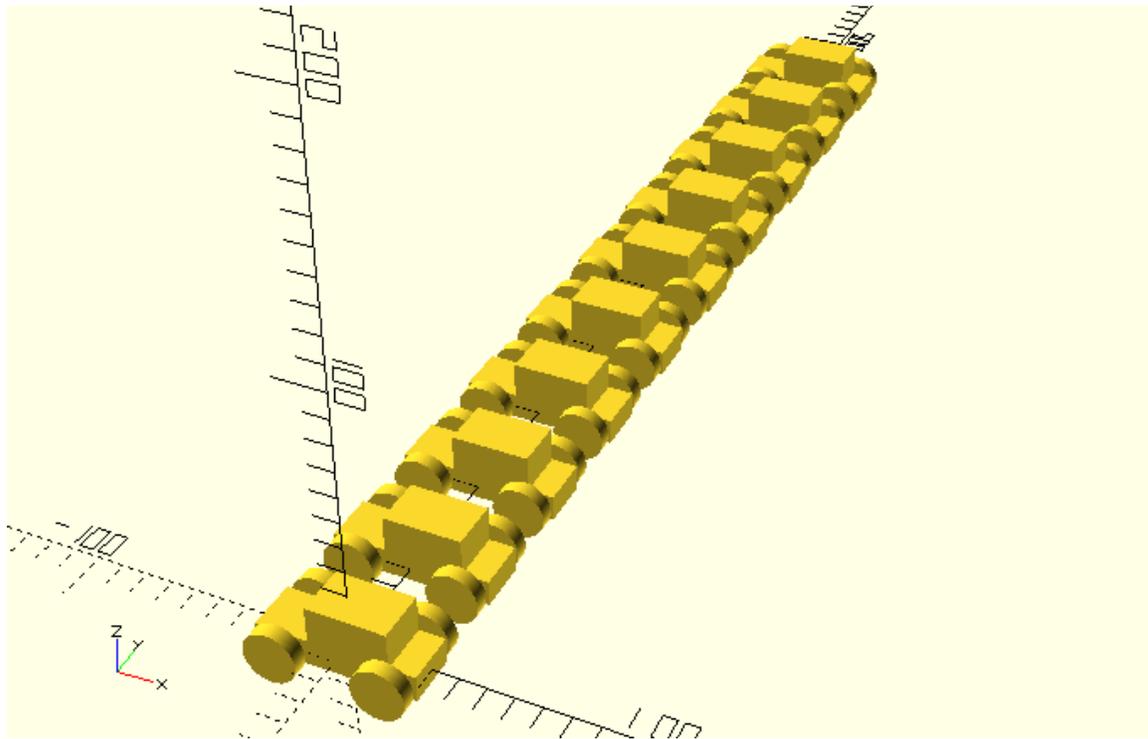
From the above exercise you have probably realized that creating a pattern of cars in this manner is not very efficient, since you must write a new statement for every car and there is also a lot of code duplication in your script. Likely you can use a for loop to achieve the same results a lot easier. The for loop is way to execute almost identical statements multiple times without having to write each individual statement in your script. Take a look at the following example.

...

```

for (dy=[0:50:450]) {
    translate([0,dy,0])car();
}
...

```

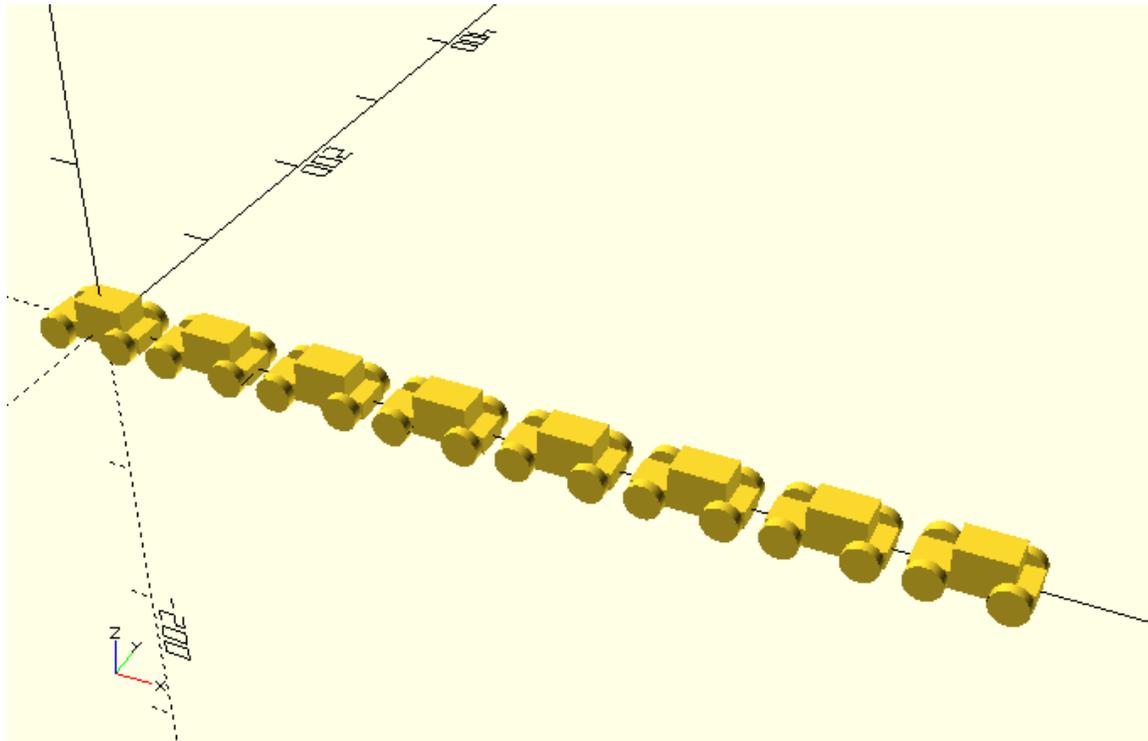


There are a few things you should notice about the syntax of the for loop. First the keyword for is typed out followed by a pair of parentheses. Inside the parentheses the variable of the for loop is defined. It's advisable to give a descriptive name to the for loop variables when applicable. In this case the variable is named dy because it represents the number of units that each car needs to be translated along the Y axis. The definition of the variable indicates that its first value will be 0 units and all following values will be incremented by 50 units each until the value 450 is reached. This means that the variable dy will take ten different values in total throughout the execution of the for loop repetitions. These values are 0, 50, 100, 150, 200, 250, 300, 350, 400 and 450. These values form a vector, which in contrast to a single value is a sequence of values. In the first repetition the variable will take the first value of the vector, which is 0. In the second repetition the second value, which is 50. And so forth. The different consecutive values that the for loop variable takes throughout the repetitions of the for loop is the key concept which makes the for loop suitable for creating patterns of multiple parts or models. Finally, after the closing parenthesis follows a pair of curly brackets. Inside the curly brackets exist the statements that will be executed repeatedly as many times as the number of values of the for loop variable. In this case the single statement inside the curly brackets will be executed 10 times which is the number of values that the dy variable will take. To avoid creating 10 cars that are completely overlapping the amount of translation along the Y axis on each

repetition of the for loop is parameterized using the dy variable. The dy variable has a different value on each repetition of the for loop thus creating the desired pattern.

Use a for loop to create a pattern of cars. The first car should be centered at the origin. Every next car should be placed behind the previous car. Specifically, every next car should be translated 70 units along the positive direction of the X axis in comparison to the previous car. The pattern should be consisted out of 8 cars in total. The for loop variable should be named dx.

```
...  
for (dx=[0:70:490]) {  
    translate([dx,0,0])car();  
}  
...
```



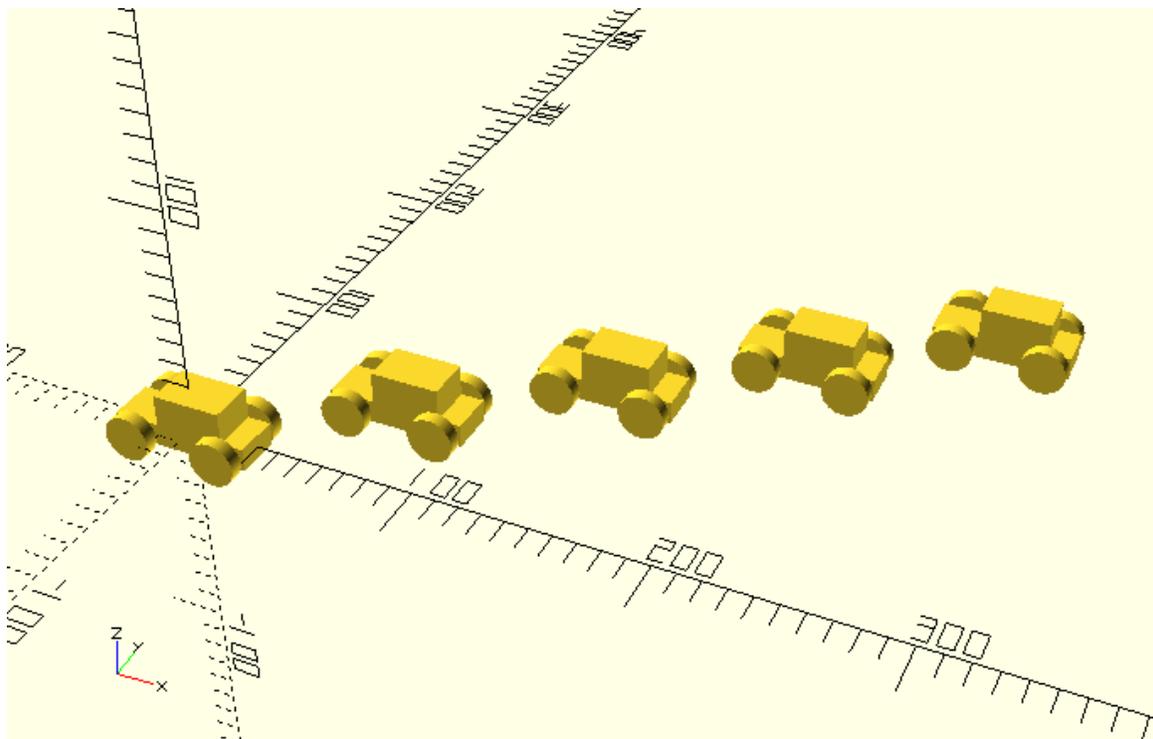
### Creating more complex patterns

In the previous examples the for loop variable dy was used directly to modify some aspect of each individual model that composes the pattern. The only aspect that was modified was the translation of each model along the Y or X axis. On each repetition the value of the dy variable was equal to the desired translation on each model.

When more than one aspect of the model needs to be modified it's a better practice for the for loop variable to take integer values 0, 1, 2, 3 etc. The required values for modifying different

aspects of the model (ex. translation along some axis, scaling of some part) are then produced from those integer values that the for loop variable takes. In the following example this concept is used to simultaneously translate each car 50 and 70 units along the positive direction of the Y and X axis.

```
...  
for (i=[0:1:4]) {  
    translate([i*70,i*50,0])car();  
}  
...
```



There are a few things you should notice. The for loop variable is now named *i*. When the for loop variable is used in this way it's usually called index and given the name *i*. Since the for loop variable takes integer values you need to multiply it by a proper number to produce the desired amount of translation along each axis. Specifically, the desired amount of translation along the Y axis is produced by multiplying the for loop variable by 50. Similarly the desired amount of translation along the X axis is produced by multiplying the for loop variable by 70.

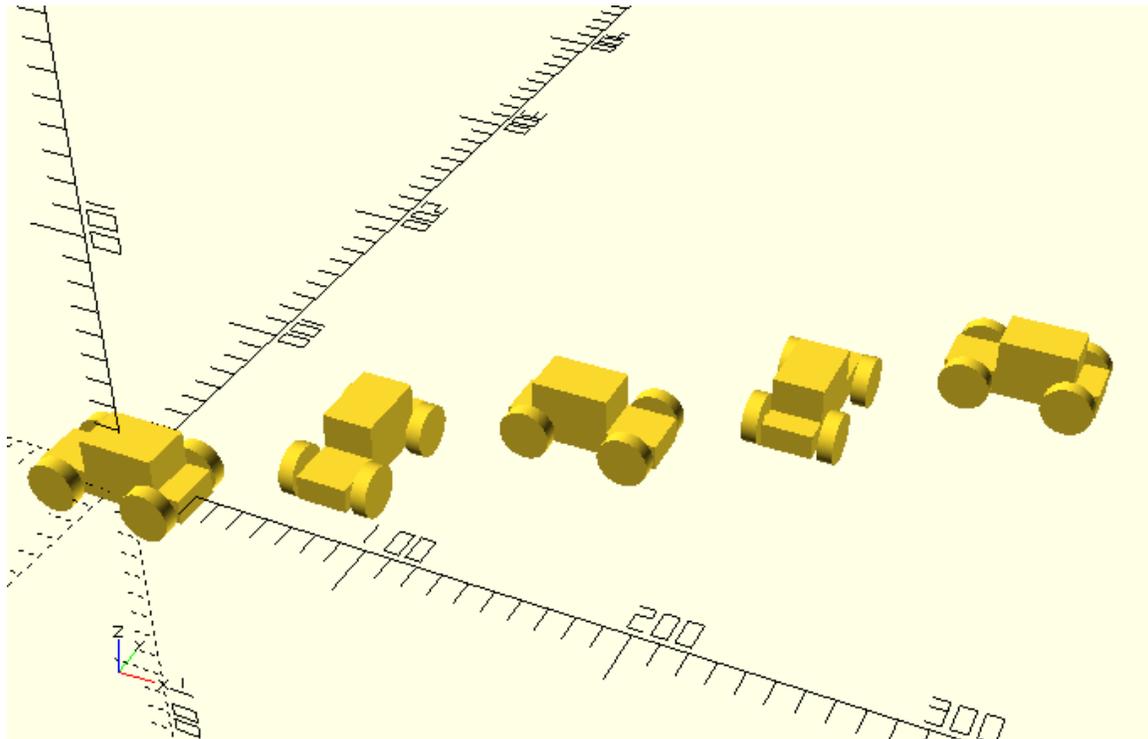
Add a rotation transformation to the previous example in order to turn the car around the Z axis. The first car shouldn't be turned at all. Each next car should be turned by 90 degrees around the positive direction of rotation of the Z axis in comparison to the previous car. The positive direction of rotation of the Z axis is the one that would rotate the X axis towards the Y axis. Does the rotation transformation need to be applied before or after the translation transformation in order to keep the cars in the same position?

\*The rotation transformation needs to be applied before the translation transformation.

...

```
for (i=[0:1:4]) {  
    translate([i*70,i*50,0])rotate([0,0,i*90])car();  
}
```

...

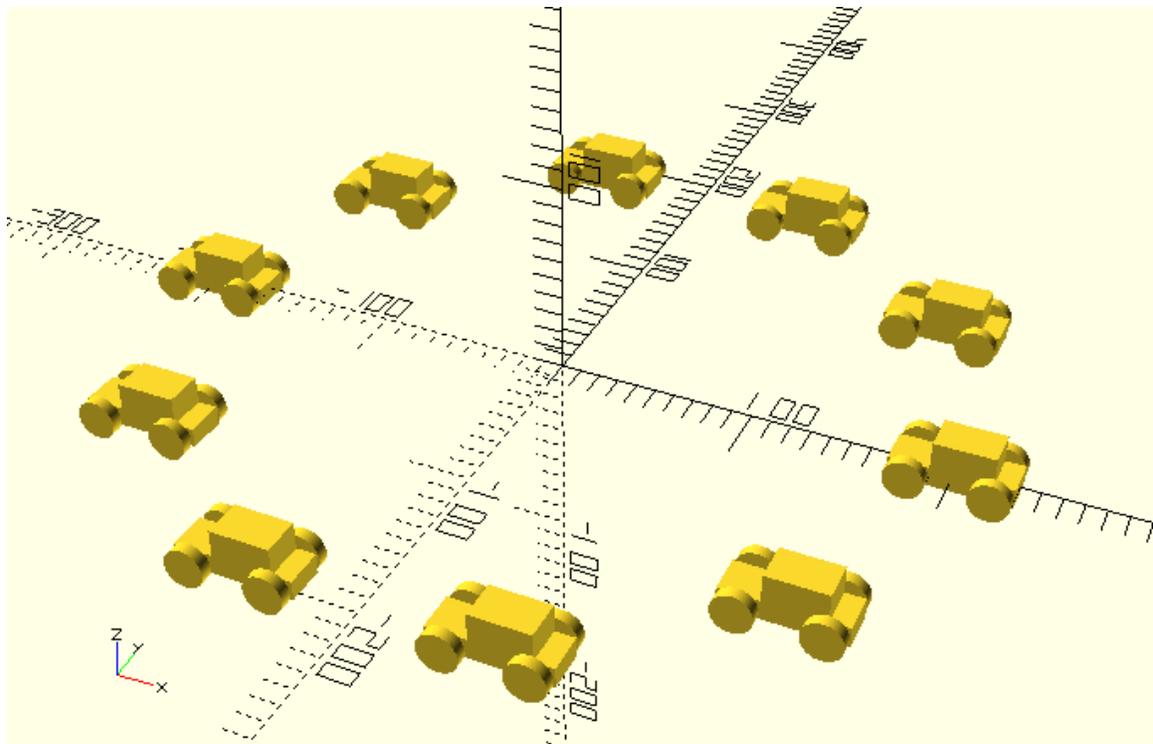


The patterns you are creating are already becoming cooler, here is an interesting one.

...

```
r = 200;  
for (i=[0:36:359]) {  
    translate([r*cos(i),r*sin(i),0])car();  
}
```

...



In the above pattern the cars have been placed at equally spaced points at the circumference of a perfect circle that has a radius of 200 units. There are a few important points you should pay attention to if you wish to create such patterns.

The first thing is that in order to create a circular pattern you need to use polar coordinates. Depending on your background you may have noticed the use of polar coordinates just by glancing at the code or you may have no idea what it is. In the later case, the only thing that you need to know is that polar coordinates is a way to produce the X and Y coordinates of a given point of a circle when you know the radius of the circle and the angle that corresponds to that point. The angle 0 corresponds to the point of the circle that belongs to the positive direction X axis. The positive counting direction of the angle is from the X to the Y axis. This means the positive Y axis corresponds to 90 degrees, the negative X axis to 180 degrees, the negative Y axis to 270 degrees and if you complete the circle the positive X axis to 360 degrees. According to the polar coordinates the X coordinate can be calculated by multiplying the radius of the circle by the cosine of the angle, while the Y coordinate can be calculated by multiplying the radius of the circle by the sine of the angle. This is how the desired amount of translation along the X and Y axis is produced.

The second thing you should notice are the values that the for loop variable *i* takes. The variable *i* starts from 0 and is incremented by 36 on each repetition in order to position 10 equally spaced cars on the circle ( $360/10 = 36$ ). The first car that is created at 0 angle and the car that correspond to 360 degrees would be exactly overlapping. In order to avoid this, you need to instruct the for loop variable to stop incrementing before it reaches 360. If you are lazy calculating  $360 - 36 = 324$ , you can just put the limit at 359. This will work fine because the for

loop variable will only take the values 0, 36, 72, 108, 144, 180, 216, 252, 288 and 324, since incrementing by another 36 units would result in 360 which exceeds 359.

By using additional variables and naming them properly you can make your scripts more descriptive and usable so that it's easier for anyone (or even you at a later point in time) to understand what they are doing and to use them. For example, the previous script can take the following form.

```
...  
r = 200; // pattern radius  
n = 10; // number of cars  
step = 360/n;  
for (i=[0:step:359]) {  
    angle = i;  
    dx = r*cos(angle);  
    dy = r*sin(angle);  
    translate([dx,dy,0])car();  
}
```

...

On the above script it is self-explanatory that the for loop variable *i* corresponds to the angle. It is also clearer what the amount of translation along each axis is. In addition, it is easy to customize this pattern by changing the radius and/or the number of cars.

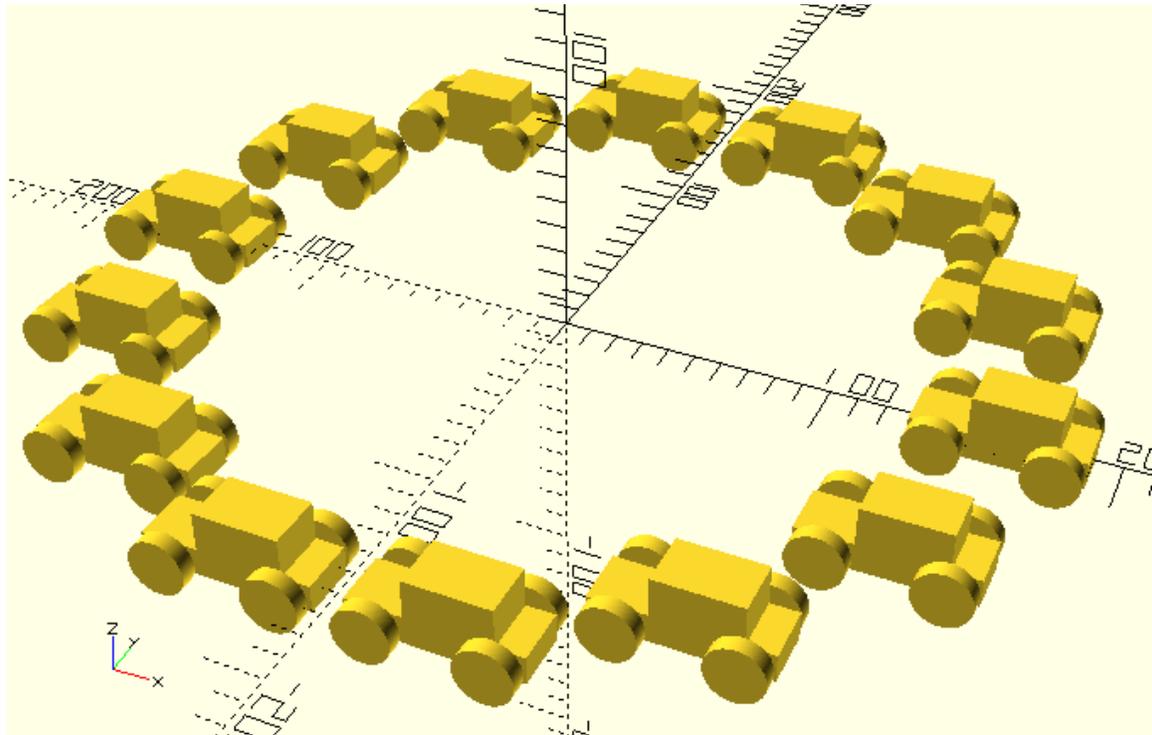
Modify the appropriate values on the above script in order to create a pattern of 14 equally spaced cars on the circumference of circle with radius of 160 units.

...

```
r = 160; // pattern radius  
n = 14; // number of cars  
step = 360/n;  
for (i=[0:step:359]) {  
    angle = i;  
    dx = r*cos(angle);  
    dy = r*sin(angle);  
    translate([dx,dy,0])car();
```

```
}
```

```
...
```



If you are not familiar with polar coordinates play around with the following script. Try assigning different values to the radius and the angle variables and see what the resulting position of the car is.

```
...
```

```
radius = 100;
```

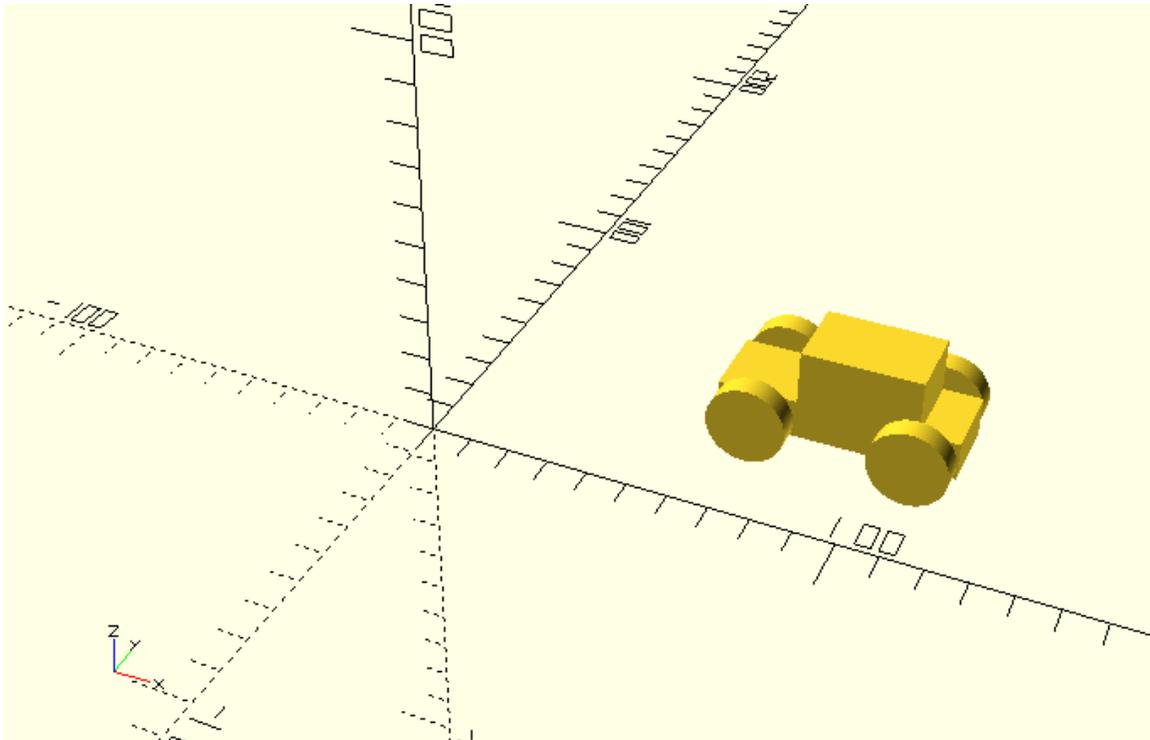
```
angle = 30; // degrees
```

```
dx = radius*cos(angle);
```

```
dy = radius*sin(angle);
```

```
translate([dx,dy,0])car();
```

```
...
```



### Challenge

...

```
r = 160; // pattern radius
```

```
n = 14; // number of cars
```

```
step = 360/n;
```

```
for (i=[0:step:359]) {
```

```
    angle = i;
```

```
    dx = r*cos(angle);
```

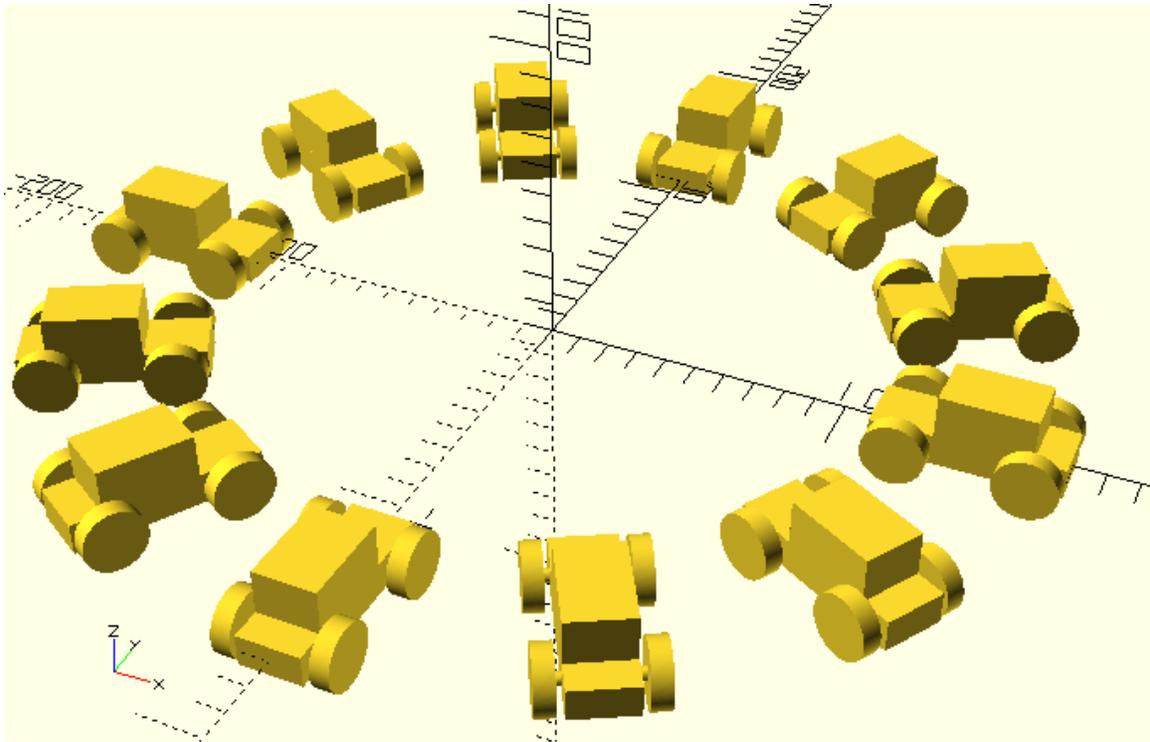
```
    dy = r*sin(angle);
```

```
    translate([dx,dy,0])car();
```

```
}
```

...

The above script was used to create a circular pattern of cars. Modify the above script by adding a rotation transformation in order to make all car face the origin. Use your modified script to create a pattern that has 12 cars and a radius of 140 units.



...

```
r = 140; // pattern radius
```

```
n = 12; // number of cars
```

```
step = 360/n;
```

```
for (i=[0:step:359]) {
```

```
    angle = i;
```

```
    dx = r*cos(angle);
```

```
    dy = r*sin(angle);
```

```
    translate([dx,dy,0])rotate([0,0,angle])car();
```

```
}
```

...

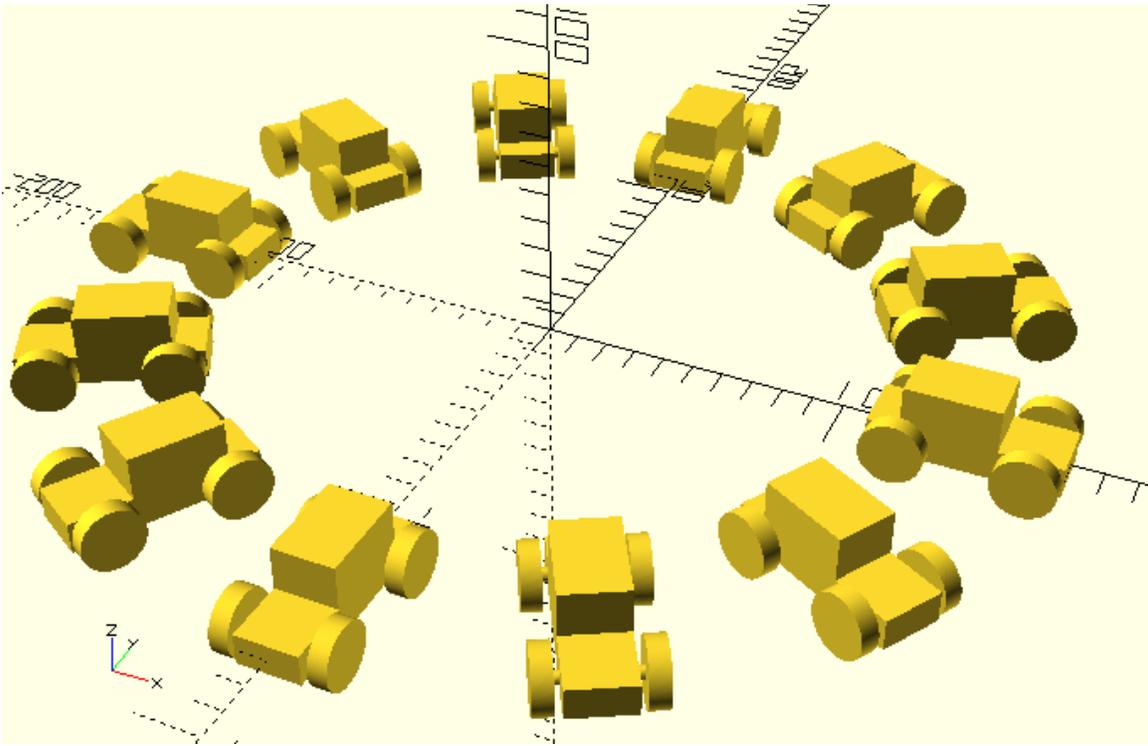
Make appropriate changes to the above script to create: i) one pattern where all cars are facing away from the origin ii) one pattern where all cars are oriented tangentially as if driving counter clockwise around a circle iii) and one as if driving clockwise around a circle

\*facing away from the origin

...

```
translate([dx,dy,0])rotate([0,0,angle - 180])car();
```

...

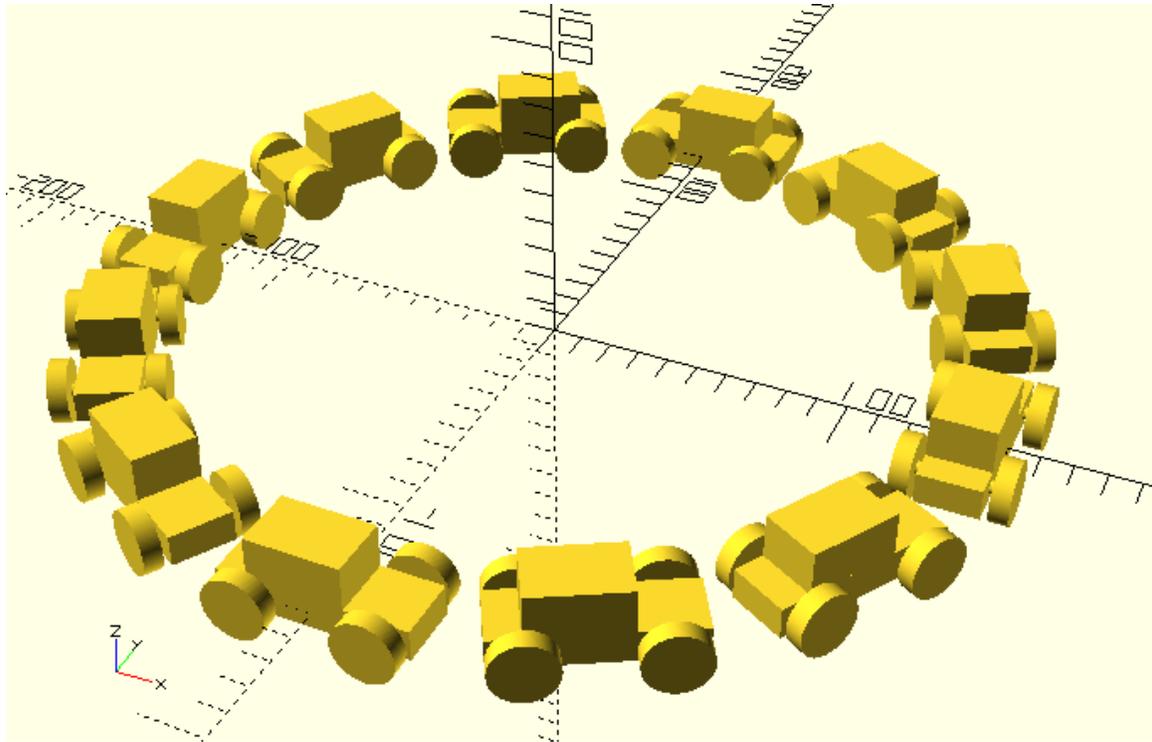


\*driving counterclockwise

...

```
translate([dx,dy,0])rotate([0,0,angle - 90])car();
```

...

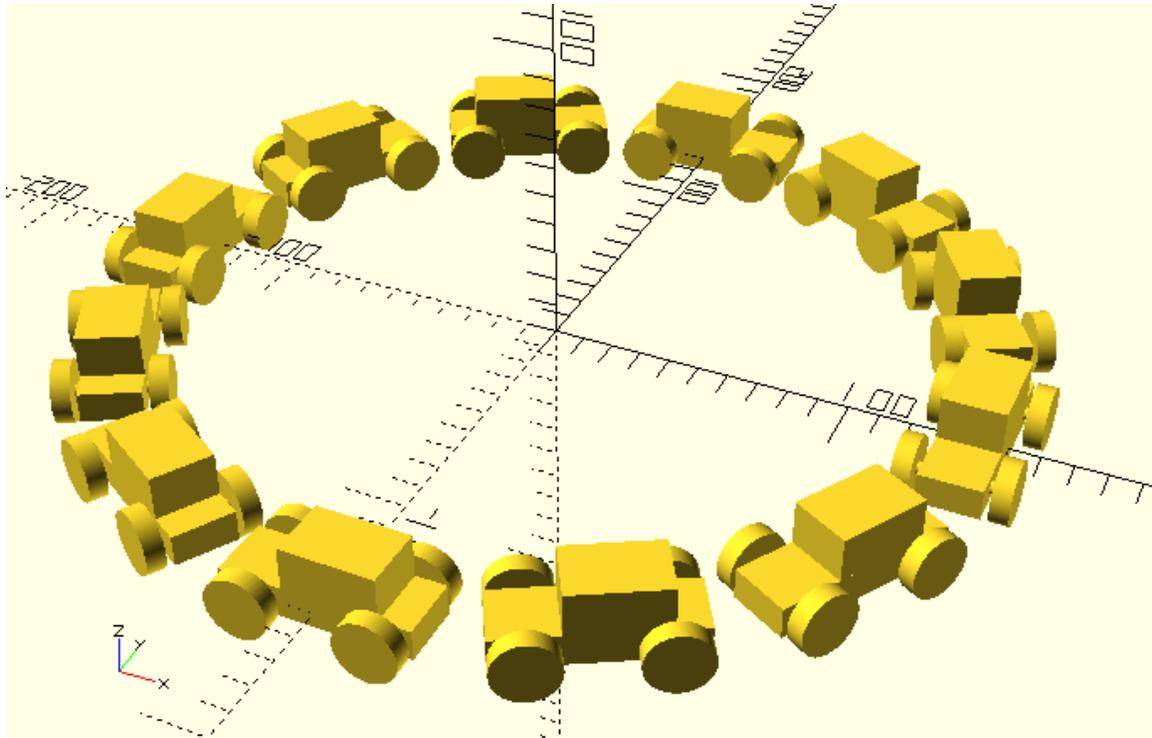


\*driving clockwise

...

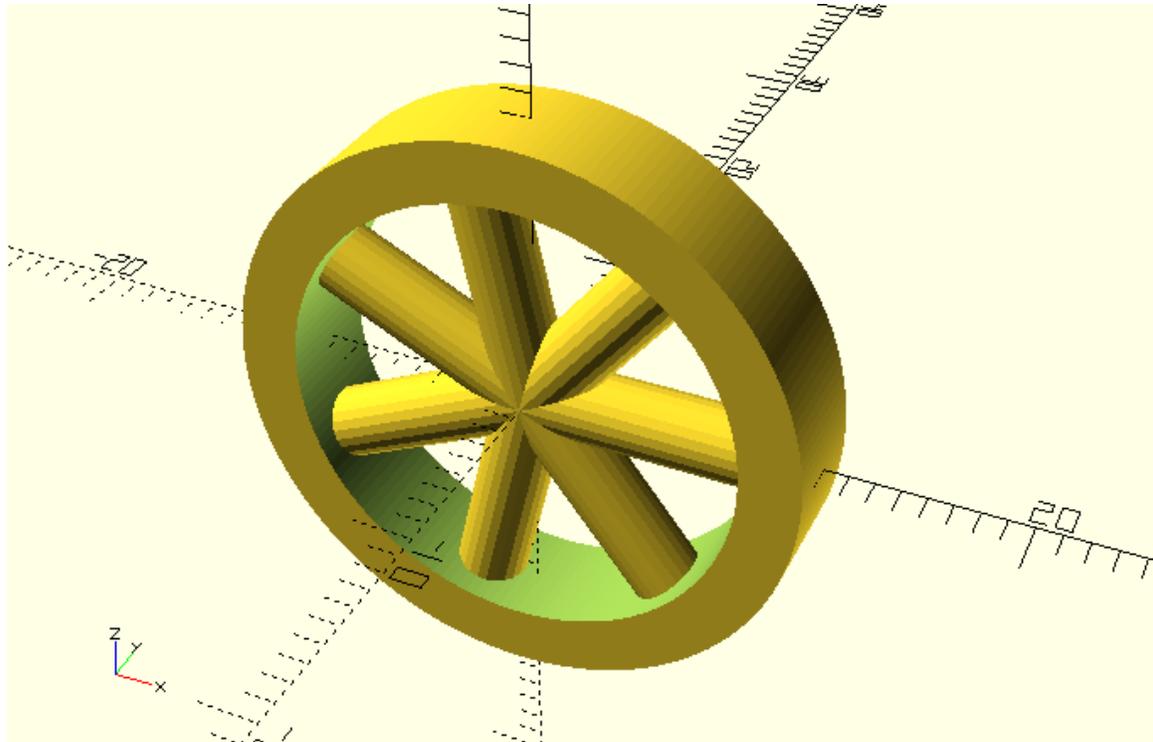
```
translate([dx,dy,0])rotate([0,0,angle - 90])car();
```

...

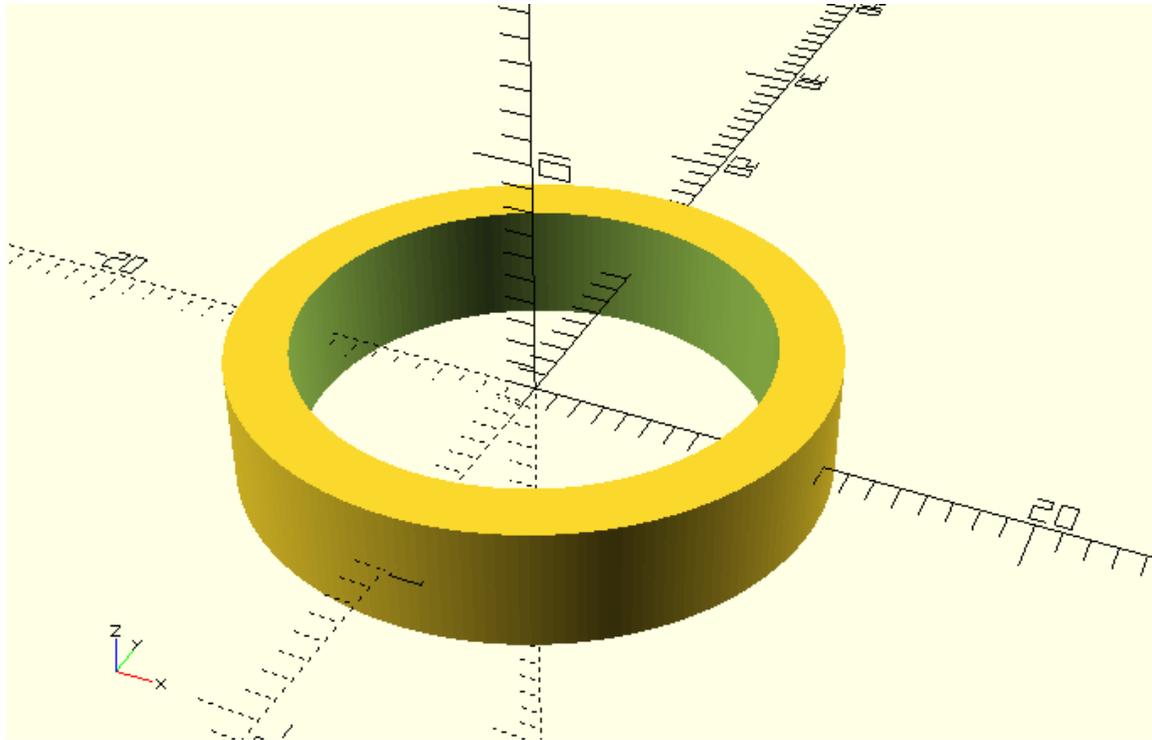


Now that you are getting a hold of using for loops to create patterns it's time to put your new skills in the development of a more sophisticated wheel design!

If you feel confident with your OpenSCAD skills or if you would like to experiment a bit more, try to building a new module named `spoked_wheel` that creates the following wheel design. If you would like some more guidance with creating this module go through the following exercises instead.



If you feel more comfortable with some additional guidance that's fine. Create a new module named `spoked_wheel` that has 5 input parameters. The input parameters should be named `radius`, `width`, `thickness`, `number_of_spokes` and `spoke_radius`. Give these variables the default values of 12, 5, 5, 7 and 1.5 respectively. Use the `cylinder` and `difference` commands to create the ring of the wheel by subtracting a small cylinder from a bigger one. The model that you need to create can be seen on the following image. For this step you are only going to use the `radius`, `width` and `thickness` variables. Remember that when you are subtracting one object from another it needs to clear the face of the other object to avoid any errors. Keep this in mind when defining the height of the small cylinder. You will also need to calculate the radius of the small cylinder from the `radius` and `thickness` variables. You can use a variable named `inner_radius` to store the result of the appropriate calculation and then use it to define the radius of the smaller cylinder.



...

```
module spoked_wheel(radius=12, width=5, thickness=5, number_of_spokes=7,
spoke_radius=1.5) {
```

```
    // Ring
```

```
    inner_radius = radius - thickness/2;
```

```
    difference() {
```

```
        cylinder(h=width,r=radius,center=true);
```

```
        cylinder(h=width + 1,r=inner_radius,center=true);
```

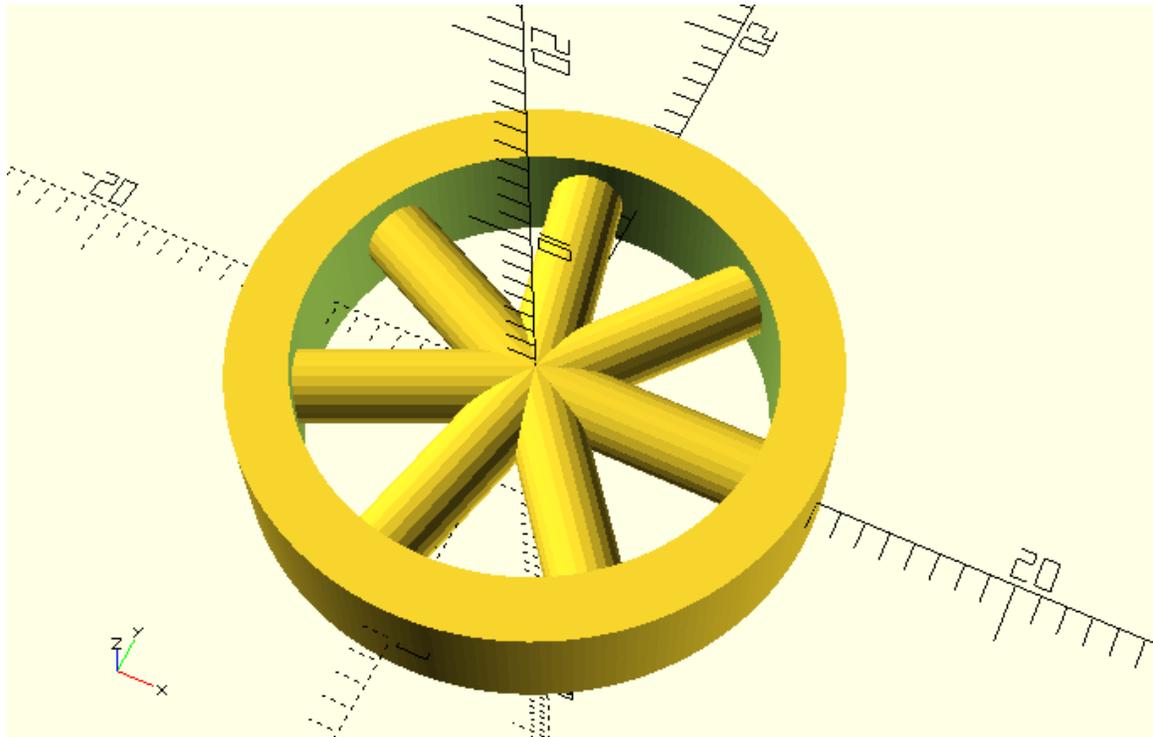
```
    }
```

```
}
```

```
spoked_wheel();
```

...

Extend the previous module to additionally create the spokes of the wheel as seen on the following image. The spokes of the wheel need to be cylindrical. The length of the spokes needs to be appropriate so that each spoke spans from the center of the ring to its half thickness. You will have to use a for loop to create the spokes as a pattern. Feel free to review previous for loop example that can help you with this.



...

```
module spoked_wheel(radius=12, width=5, thickness=5, number_of_spokes=7,  
spoke_radius=1.5) {
```

```
    // Ring
```

```
    inner_radius = radius - thickness/2;
```

```
    difference() {
```

```
        cylinder(h=width,r=radius,center=true);
```

```
        cylinder(h=width + 1,r=inner_radius,center=true);
```

```
    }
```

```
    // Spokes
```

```
    spoke_length = radius - thickness/4;
```

```
    step = 360/number_of_spokes;
```

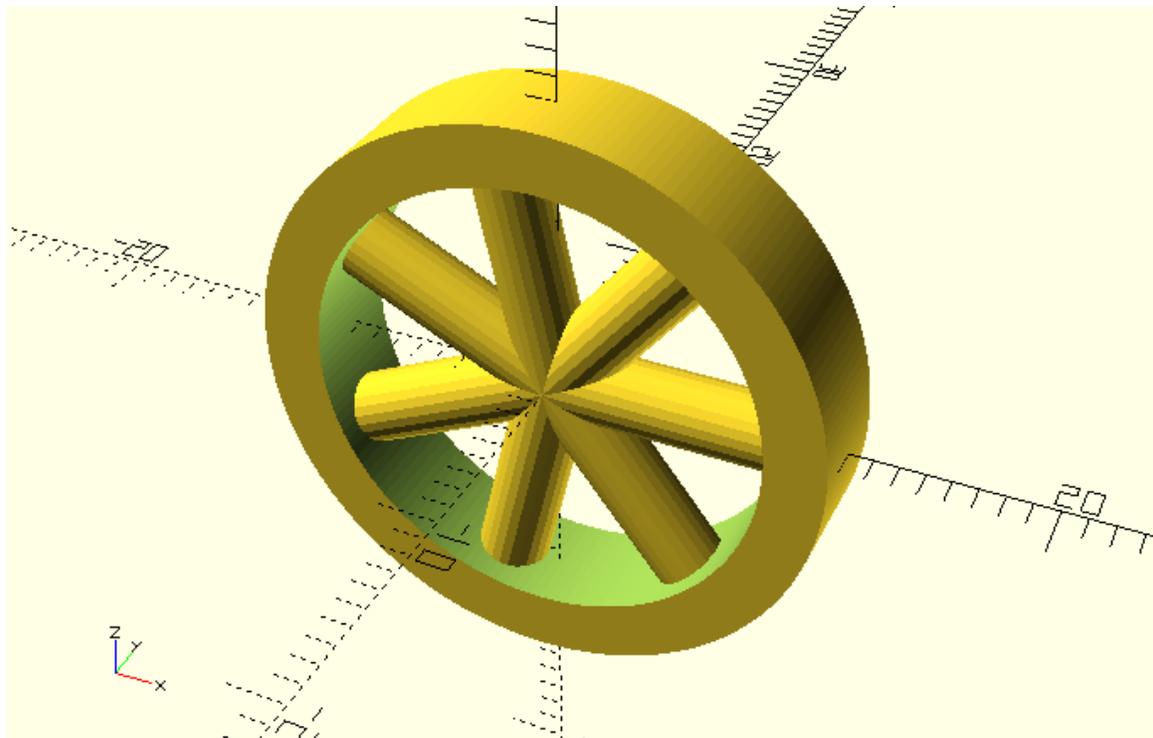
```
    for (i=[0:step:359]) {
```

```

    angle = i;
    rotate([0,90,angle])cylinder(h=spoke_length,r=spoke_radius);
  }
}
spoked_wheel();
...

```

For the new wheel design to be compatible with the existing wheel designs and modules that you have created throughout the tutorial, it needs to be rotated to stand straight as in the following image. Add an appropriate rotation transformation to do so.



```

...
module spoked_wheel(radius=12, width=5, thickness=5, number_of_spokes=7,
spoke_radius=1.5) {

  rotate([90,0,0]) {

    // Ring
    inner_radius = radius - thickness/2;
    difference() {

```

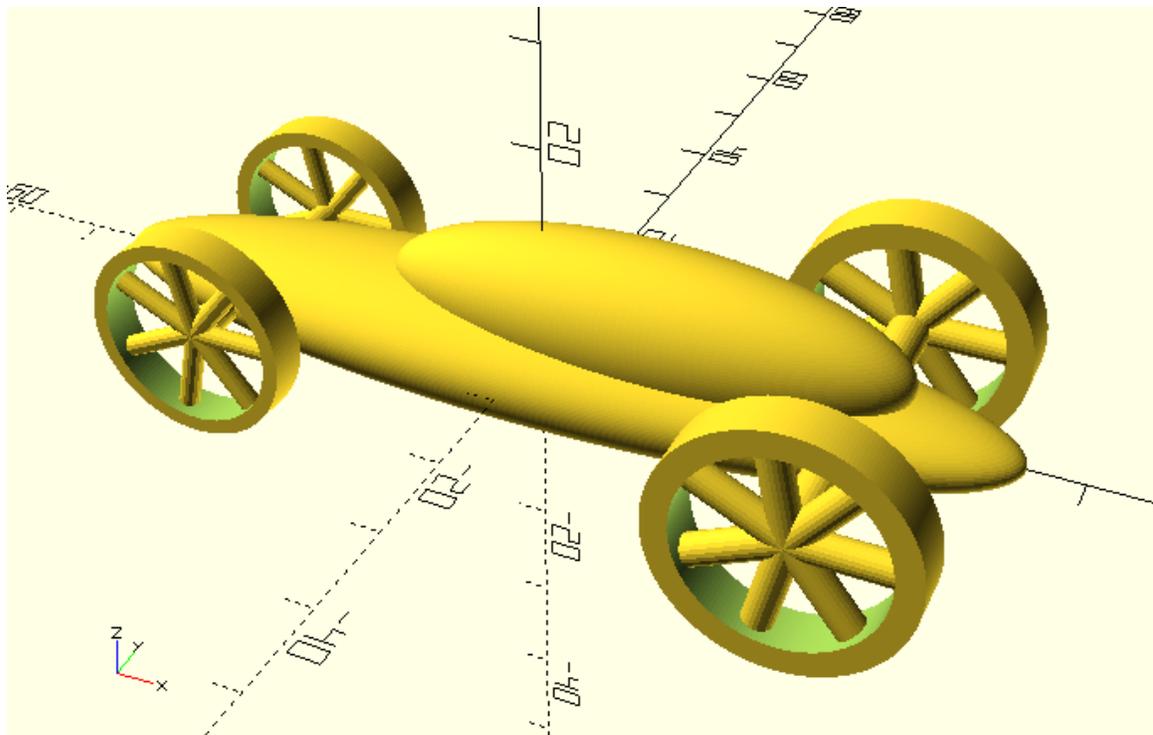
```

cylinder(h=width,r=radius,center=true);
cylinder(h=width + 1,r=inner_radius,center=true);
}

// Spokes
spoke_length = radius - thickness/4;
step = 360/number_of_spokes;
for (i=[0:step:359]) {
    angle = i;
    rotate([0,90,angle])cylinder(h=spoke_length,r=spoke_radius);
}
}
}
...
spoked_wheel();
...

```

Add the `spoked_wheel` module on the `vehicle_parts.scad` script. Use the new wheel design in one of your car models. If you don't have any ideas you can try replicating the following model.



use <vehicle\_parts.scad>

```
$fa = 1;
```

```
$fs = 0.4;
```

```
front_track = 24;
```

```
rear_track = 34;
```

```
wheelbase = 60;
```

```
front_wheels_radius = 10;
```

```
front_wheels_width = 4;
```

```
front_wheels_thickness = 3;
```

```
front_spoke_radius = 1;
```

```
front_axle_radius = 1.5;
```

```

// Round car body
resize([90,20,12])sphere(r=10);
translate([10,0,5])resize([50,15,15])sphere(r=10);

// Wheels
translate([-wheelbase/2,-front_track/2,0])spoked_wheel(radius=front_wheels_radius,
width=front_wheels_width, thickness=front_wheels_thickness,
spoke_radius=front_spoke_radius);

translate([-wheelbase/2,front_track/2,0])spoked_wheel(radius=front_wheels_radius,
width=front_wheels_width, thickness=front_wheels_thickness,
spoke_radius=front_spoke_radius);

translate([wheelbase/2,-rear_track/2,0])spoked_wheel();
translate([wheelbase/2,rear_track/2,0])spoked_wheel();

// Axles
translate([-wheelbase/2,0,0])axle(track=front_track, radius=front_axle_radius);
translate([wheelbase/2,0,0])axle(track=rear_track);

```

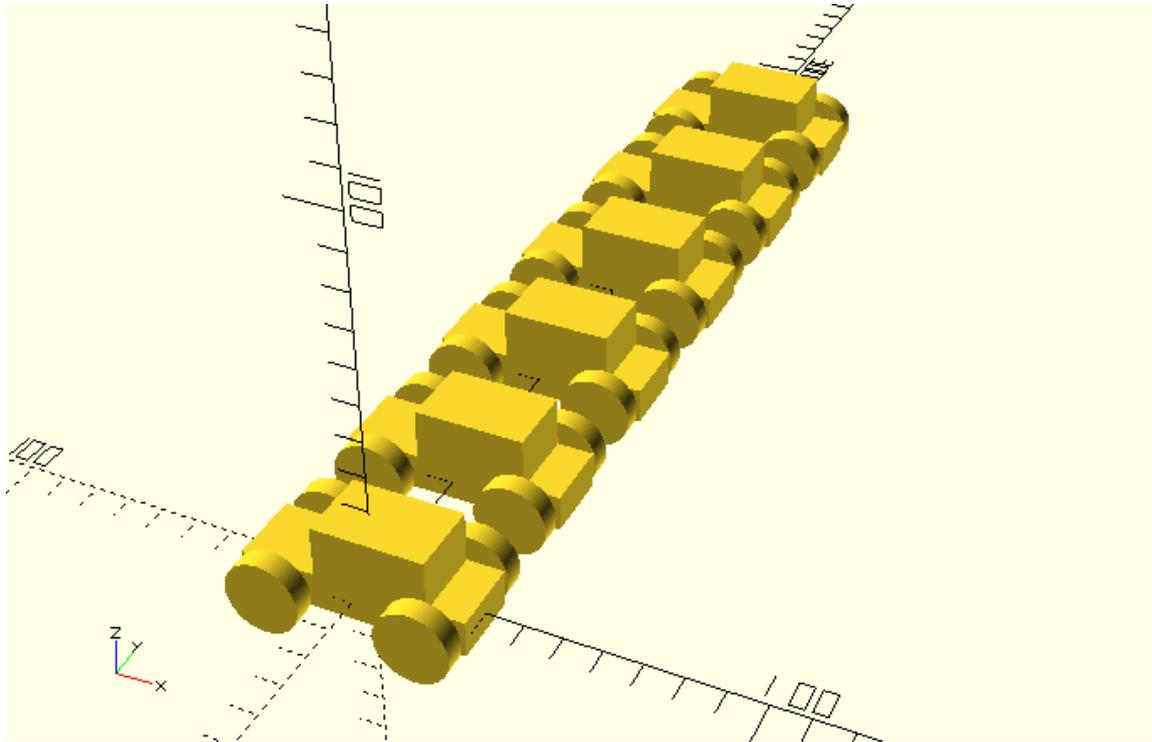
### **Creating patterns of patterns – Nested for loops**

The following script creates a row of cars along the Y axis.

```

...
n = 6; // number of cars
y_spacing = 50;
for (dy=[0:spacing:n*spacing-1]) {
    translate([0,dy,0])car();
}
...

```



Modify the above script to create 4 additional rows of cars. Each row should be translated by 70 units along the positive direction of the X axis in comparison to the previous row.

```

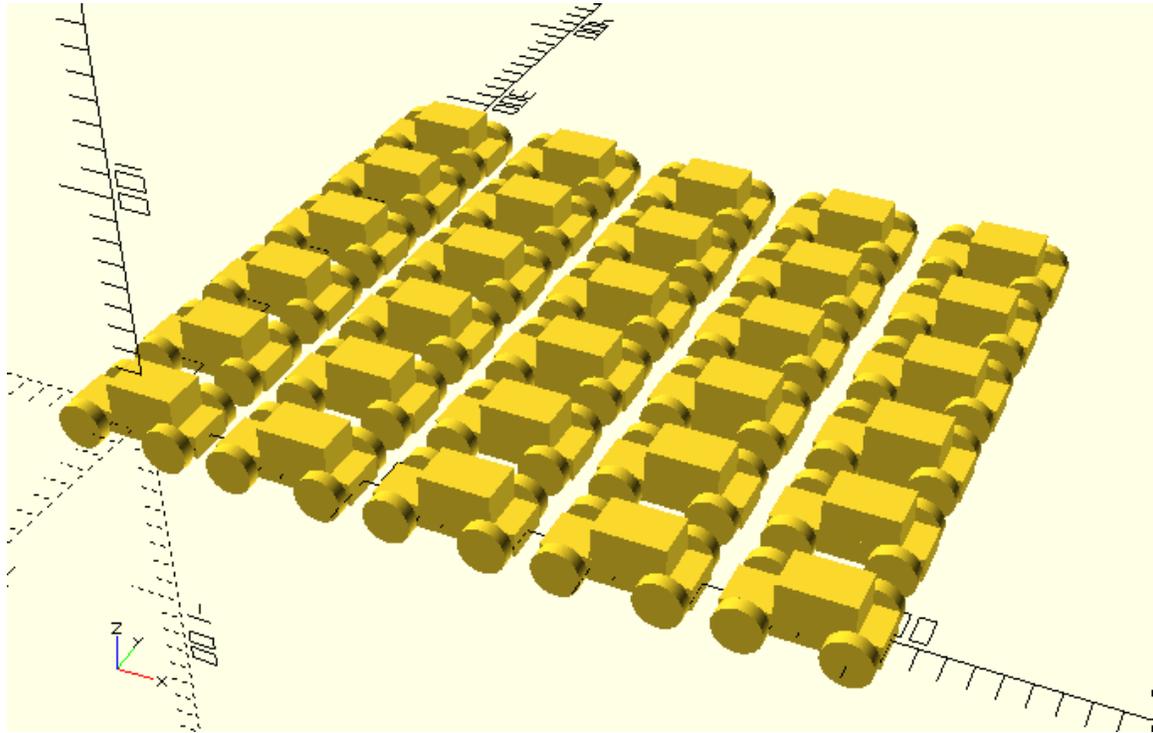
...
n = 6; // number of cars
y_spacing = 50;
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([0,dy,0])car();
}
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([70,dy,0])car();
}
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([140,dy,0])car();
}
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([210,dy,0])car();
}

```

```

}
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([280,dy,0])car();
}
...

```



If you have been paying close attention to the tutorial so far you may have probably thought that the script above is not very efficient. It has a lot of code duplication and the number of rows can't be easily modified. You faced a similar situation in the beginning of this chapter when you wanted to create a row of cars. What you did to solve the problem was to place the statement that creates a piece of the pattern (a single car) inside a for loop in order to generate the whole pattern (a row of cars) without having to type out a statement for each single car. The same principle can be applied in the current problem. In this case the whole pattern you want to create is many rows of cars, while a single piece of the pattern is one row of cars. Following the same process, the statements that create a row of cars need to be placed inside a for loop in order to create a pattern of rows of cars. This results in a for loop being placed inside another for loop. For loops that are used in this way are called nested. This is how you can implement this.

```

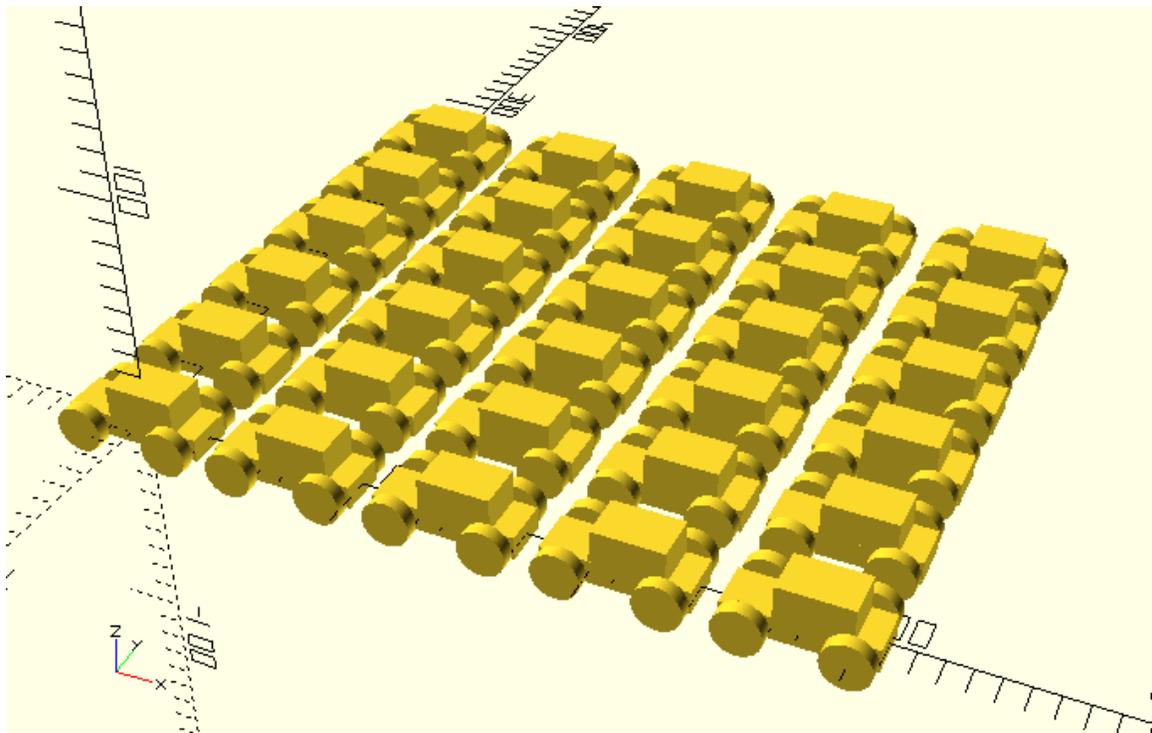
...
n_cars = 6;
y_spacing = 50;

```

```

n_rows = 5;
x_spacing = 70;
for (dx=[0:x_spacing:n_rows*x_spacing-1]) {
    for (dy=[0:y_spacing:n_cars*y_spacing-1]) {
        translate([dx,dy,0])car();
    }
}
...

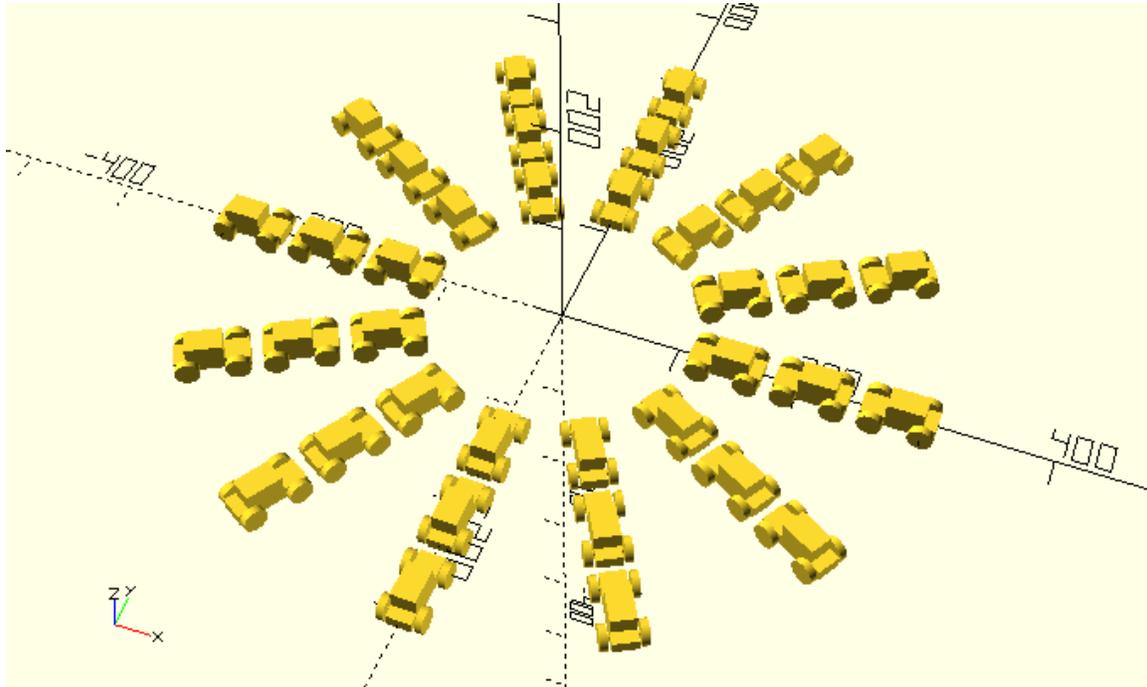
```



You should notice the following concept. During the first repetition of the outer for loop all repetitions of the inner for loop are executed, thus creating the first row of cars. During the second repetition of the outer for loop all repetitions of the inner for loop are executed, thus creating the second row of cars. And so forth. What makes the rows of cars to be properly positioned is the parameterized translation along the X axis, which is controlled with the dx variable. During one repetition of the outer for loop the value of dx remains constant for all repetition of the inner for loop. On the following repetition of the outer for loop the dx variable takes its next value which also remains constant for all repetitions of the inner for loop, thus translating all car of the corresponding row by the appropriate amount along the X axis. And so forth.

Use nested for loops to create three circular patterns of cars similar to the image below. The for loop variable of the outer loop should be used to parameterize the radius of each pattern.

The radius of the circular patterns should be 70, 140 and 280 units respectively. Each pattern should be consisted of 12 cars.



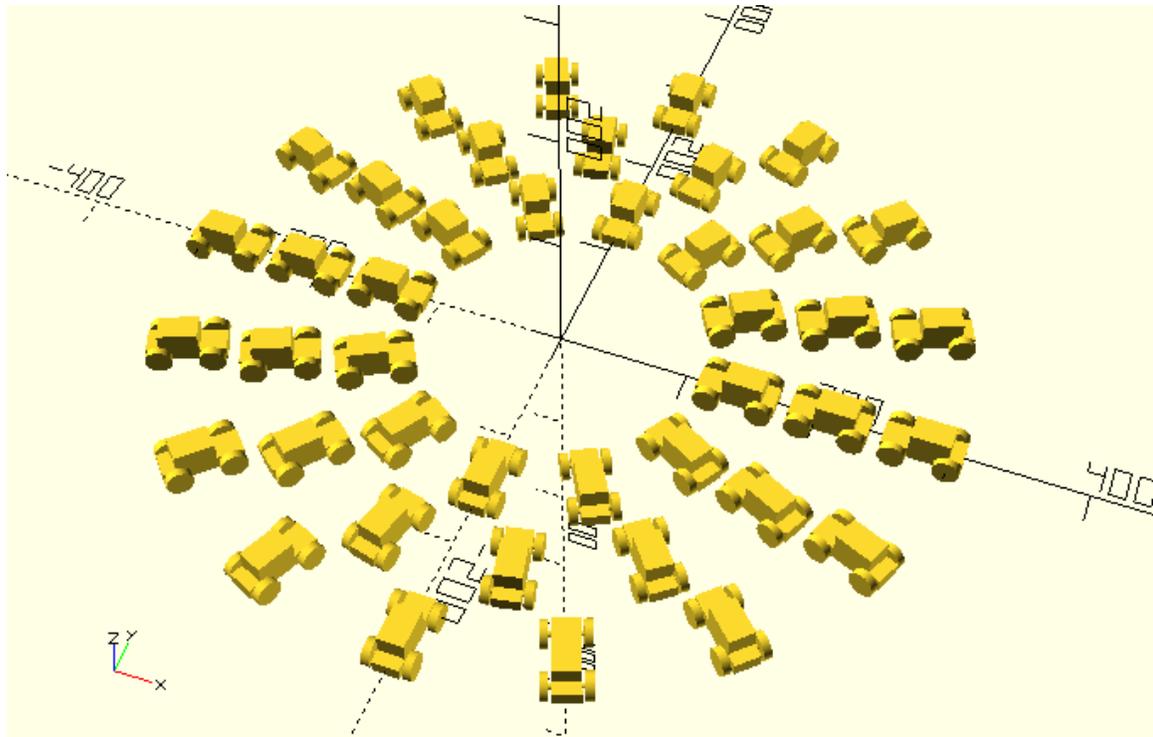
...

```
n = 12; // number of cars
step = 360/n;
for (r=[140:70:280]) {
    for (angle=[0:step:359]) {
        dx = r*cos(angle);
        dy = r*sin(angle);
        translate([dx,dy,0])rotate(angle)car();
    }
}
```

...

Modify the script of the previous exercise so that not only the radius but also the number of cars is different for each pattern. To do so use an index variable  $i$  as the variable of the outer loop instead of the variable  $r$  that corresponds to the radius. The variable  $r$  should be calculated at each repetition of the outer for loop according to the formula  $r = 70 + i*70$ . Additionally, on each repetition of the outer for loop the  $n$  variable should take different

values according to the formula  $n = 12 + i*2$ . The step variable also needs to be updated on each repetition of the outer for loop. The  $i$  variable should take the values 0, 1 and 2.



```
for (j=[0:1:2]) {  
    r = 140 + j*70;  
    n = 12 + 2*j;  
    step = 360/n;  
    for (angle=[0:step:359]) {  
        dx = r*cos(angle);  
        dy = r*sin(angle);  
        translate([dx,dy,0])rotate(angle)car();  
    }  
}
```