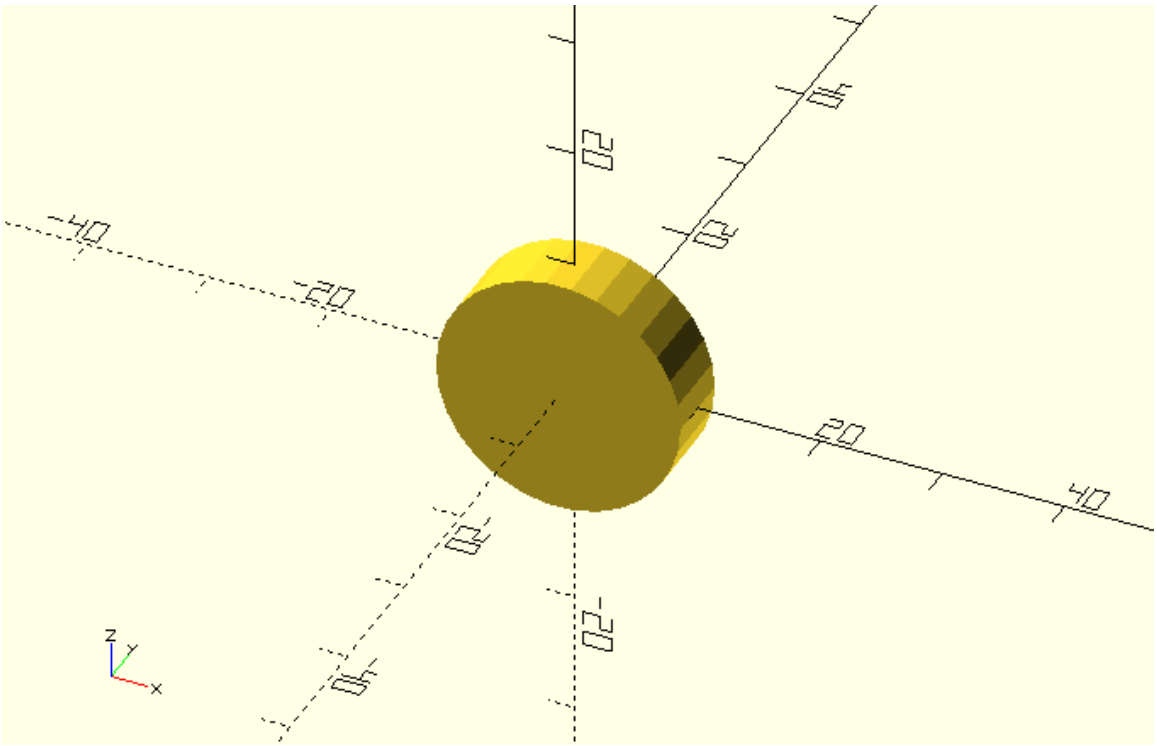
Chapter 5

Creating and utilizing modules as separate scripts

In the previous chapter you learned one of the most powerful features of OpenSCAD, the module, and how it can be used for parametric design. You also had the chance to separate the car into different modules and then recombine them to create a different type of vehicle. Using modules can be also seen as a way to organize your creations and to build your own library of objects. The wheel module could potential be used in a plethora of designs, so it would be great to have it easily available whenever desired without having to redefine it inside the script of your current design. To do so you need to define and save the wheel module as a separate script.

Define the following simple_wheel module in a separate script file. In the same script make a call to the simple_wheel module so that you visually see what object this module creates. Save the script file as a*.scad file named simple_wheel.

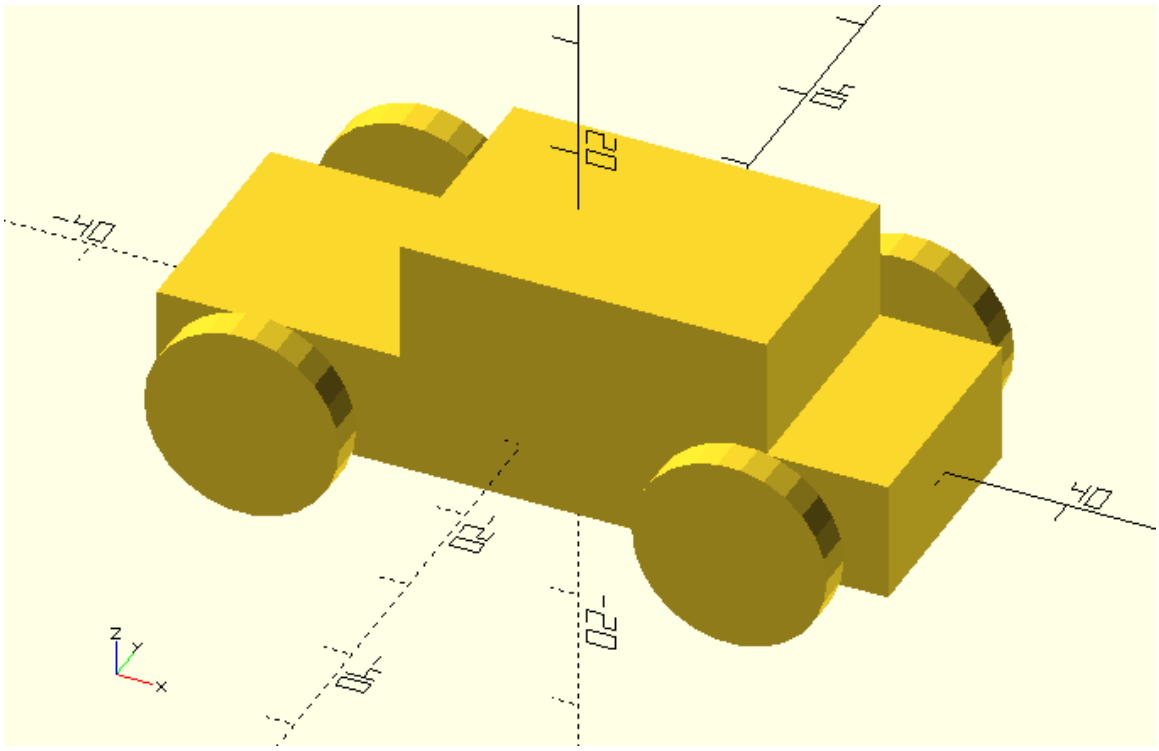
```
module simple_wheel(wheel_radius=10, wheel_width=6) {
   rotate([90,0,0])cylinder(h=wheel_width,r=wheel_radius,center=true);
}
simple_wheel();
```



Now it's time to utilize this saved module in another design. First you need to create a new design.

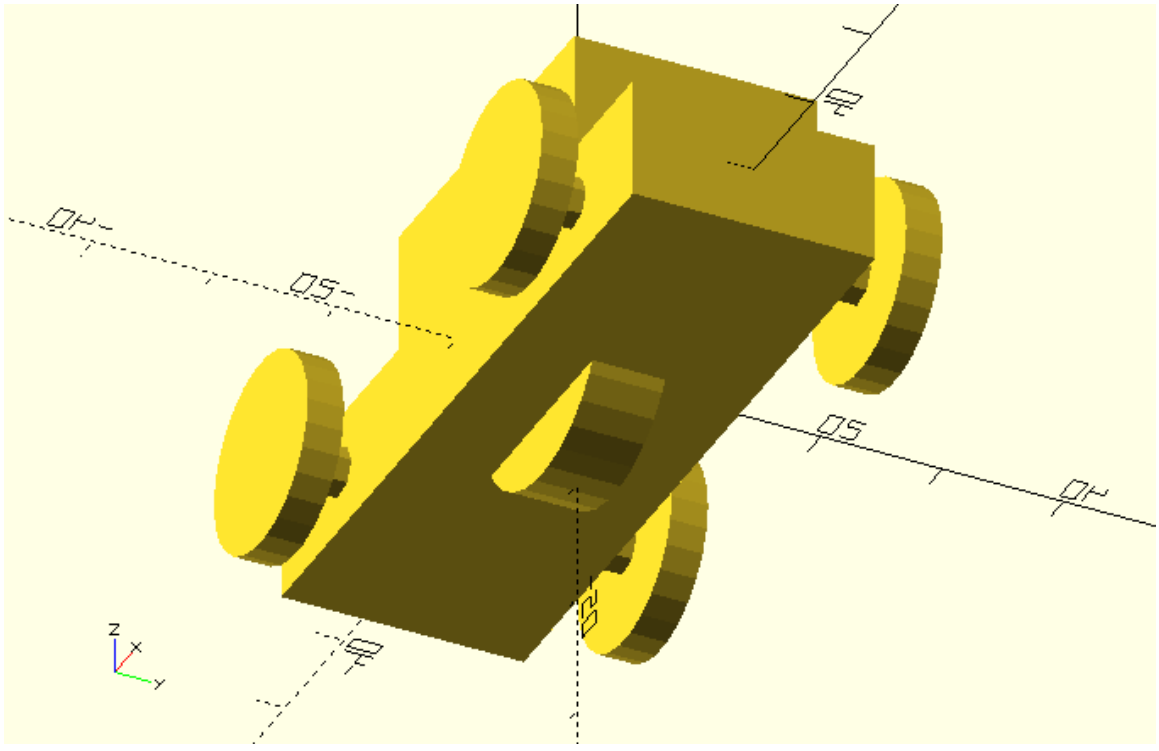
Create a new script with the following car design. Give the script any name you like but save the script in the same working directory as the simple_wheel module.

```
wheel_radius = 8;

base_height = 10;

top_height = 10;

track = 30;

// Car body base

cube([60,20,base_height],center=true);

// Car body top

translate([5,0,base_height/2+top_height/2])cube([30,20,top_height],center=true);

// Front left wheel

translate([-20,-track/2,0])rotate([90,0,0])cylinder(h=3,r=wheel_radius,center=true);

// Front right wheel

translate([-20,track/2,0])rotate([90,0,0])cylinder(h=3,r=wheel_radius,center=true);

// Rear left wheel

translate([20,-track/2,0])rotate([90,0,0])cylinder(h=3,r=wheel_radius,center=true);

// Rear right wheel

translate([20,track/2,0])rotate([90,0,0])cylinder(h=3,r=wheel_radius,center=true);

// Front axle

translate([-20,0,0])rotate([90,0,0])cylinder(h=track,r=2,center=true);

// Rear axle

translate([20,0,0])rotate([90,0,0])cylinder(h=track,r=2,center=true);
```



There are two ways in which the simple_wheel.scad script can be utilized in your car design. It can be either included or used. To include the script, you have to add the following statement at the car's script.

include <simple_wheel.scad>

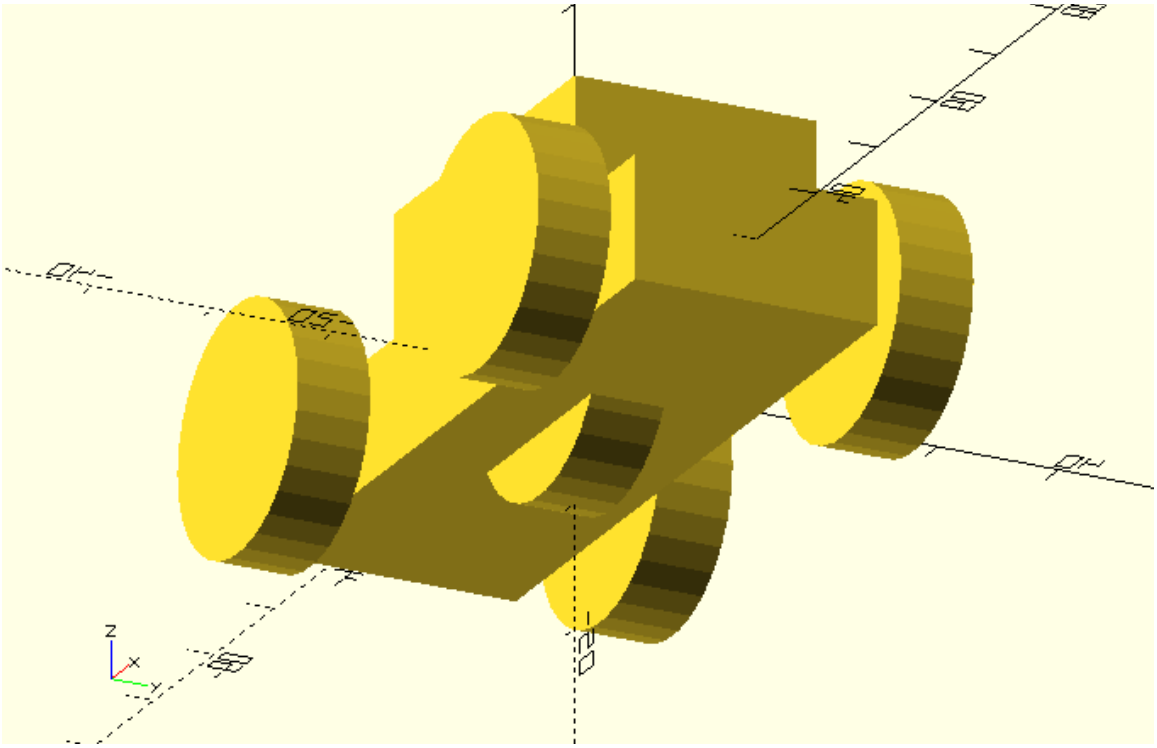


You should notice that something you might didn't expect happened. A wheel has been created at the origin. This is the object of the simple_wheel script. When you use include, OpenSCAD treats the whole external script that you are including as if it was apart off your current script. In the simple_wheel.scad script aside from the simple_wheel module definition there is also a call to the simple_wheel module which creates a wheel object. As a result of using the include command this object is also created in the car's model. This is something that you are going to change by using the use instead of the include command, but don't bother about it for a moment.

The car's wheels are currently created with the cylinder command. Since the simple_wheel.scad script has been included in the car's script, the simple_wheel module should be available. Replace the cylinder commands with calls to the simple_while module. Do any rotate commands become unnecessary? The calls to the simple_wheel module shall not contain any definition of parameters.

```
include <simple_wheel.scad>

wheel_radius = 8;

base_height = 10;

top_height = 10;

track = 30;

// Car body base

cube([60,20,base_height],center=true);

// Car body top

translate([5,0,base_height/2+top_height/2])cube([30,20,top_height],center=true);

// Front left wheel

translate([-20,-track/2,0])simple_wheel();

// Front right wheel

translate([-20,track/2,0])simple_wheel();

// Rear left wheel

translate([20,-track/2,0])simple_wheel();

// Rear right wheel

translate([20,track/2,0])simple_wheel();

// Front axle

translate([-20,0,0])rotate([90,0,0])cylinder(h=track,r=2,center=true);
```

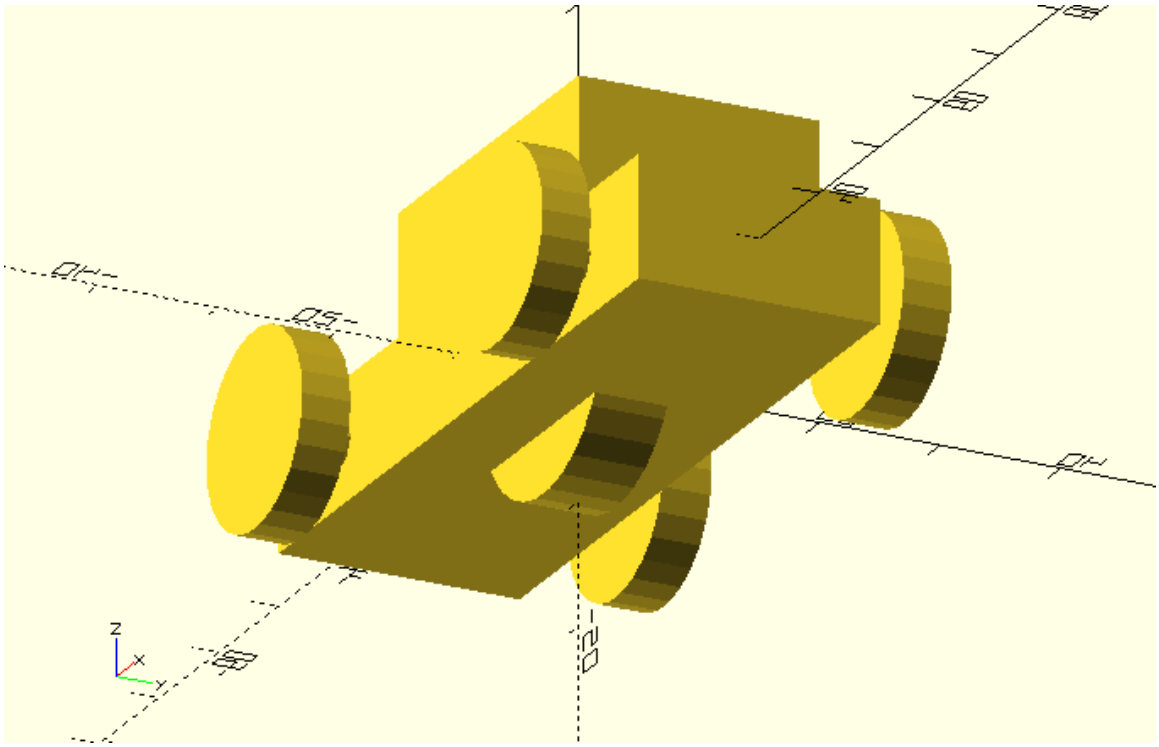
```
// Rear axle

translate([20,0,0])rotate([90,0,0])cylinder(h=track,r=2,center=true);
```



Define the wheel_radius and wheel_width parameters in the calls to the simple_wheel module. To do so use the existing wheel_radius variable as well as a wheel_width variable that you are going to define. Set the variables equal to values that you like.

```
include <simple_wheel.scad>

wheel_radius = 8;

wheel_width = 4;

base_height = 10;

top_height = 10;

track = 30;

// Car body base

cube([60,20,base_height],center=true);

// Car body top

translate([5,0,base_height/2+top_height/2])cube([30,20,top_height],center=true);

// Front left wheel
```

```
translate([-20,-track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

// Front right wheel

translate([-20,track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

// Rear left wheel

translate([20,-track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

// Rear right wheel

translate([20,track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

// Front axle

translate([-20,0,0])rotate([90,0,0])cylinder(h=track,r=2,center=true);

// Rear axle

translate([20,0,0])rotate([90,0,0])cylinder(h=track,r=2,center=true);
```

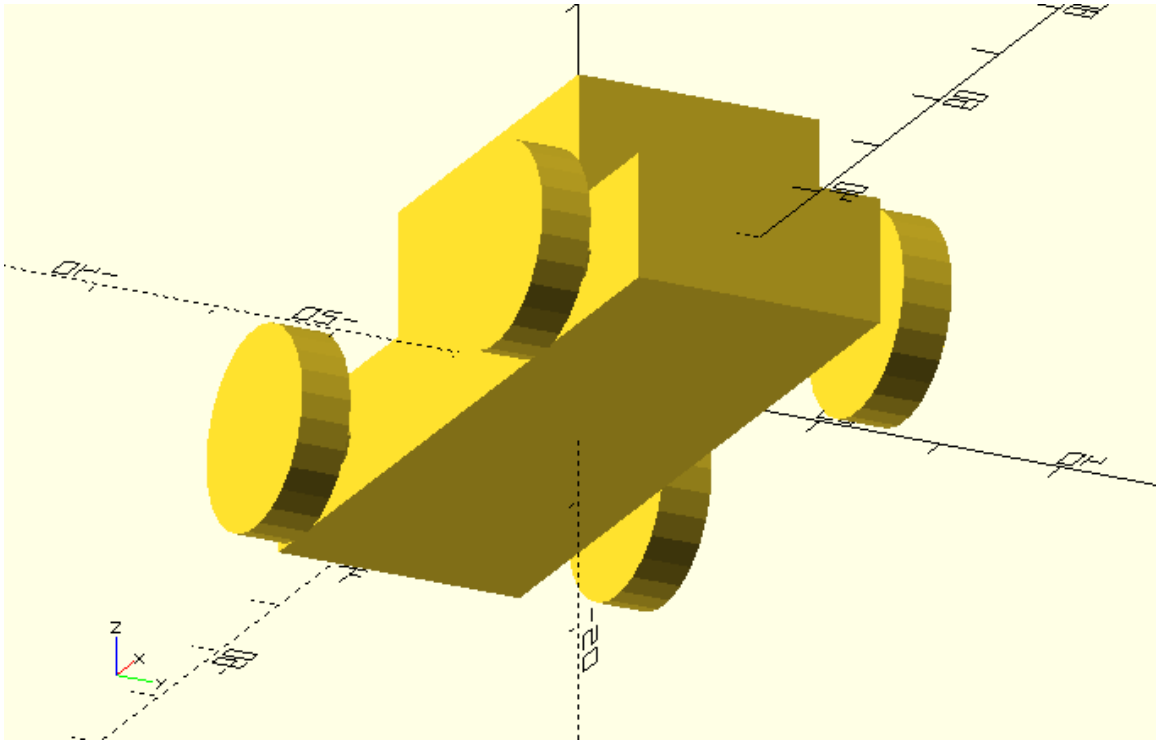


From the above examples you should keep in mind that when you include an external script in your current script, the modules of the external script become available in your current script, but additionally any objects that were created in the external script are also created in the

current one. Since the wheel at the origin is not desired in this case, it's time to use the use command instead of the include.

Replace the include command of the last example with a use command.

use <wheels.scad>



You should notice that a wheel is no longer created at the origin. You should keep in mind that the use command works like the include command with the only difference being that the use command doesn't create any objects, but rather just makes the modules of the external script available in the current script.

Using a script with multiple modules

In the previous example, the simple_wheel.scad script had only one module. The simple_wheel module. This doesn't have to always be the case.

Add the following module in the simple_wheel.scad script. Rename the simple_wheel.scad script to wheels.scad.

module complex_wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2) {

cylinder_height=2*wheel_radius;

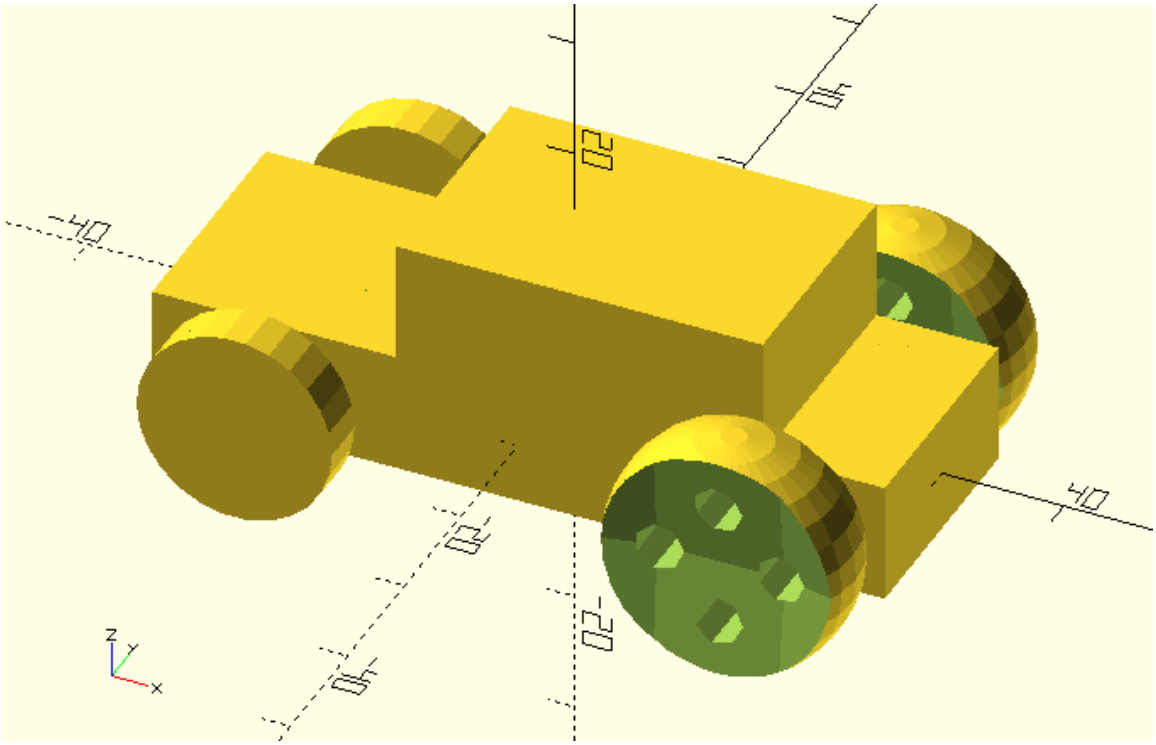
difference(){

```
// Wheel sphere

sphere(r=wheel_radius);

// Side sphere 1

translate([0,side_spheres_radius + hub_thickness/2,0])sphere(r=side_spheres_radius);

// Side sphere 2

translate([0,- (side_spheres_radius + hub_thickness/2),0])sphere(r=side_spheres_radius);

// Cylinder 1

translate([wheel_radius/2,0,0])rotate([90,0,0])cylinder(h=cylinder_height,r=cylinder_radius,cent
er=true);

// Cylinder 2

translate([0,0,wheel_radius/2])rotate([90,0,0])cylinder(h=cylinder_height,r=cylinder_radius,cent
er=true);

// Cylinder 3

translate([-
wheel_radius/2,0,0])rotate([90,0,0])cylinder(h=cylinder_height,r=cylinder_radius,center=true);

// Cylinder 4

translate([0,0,-
wheel_radius/2])rotate([90,0,0])cylinder(h=cylinder_height,r=cylinder_radius,center=true);

};

}
```

Use the wheels.scad script in your car script. Use the simple_wheel module to create the front wheels and the complex_wheel module to create the rear wheels.

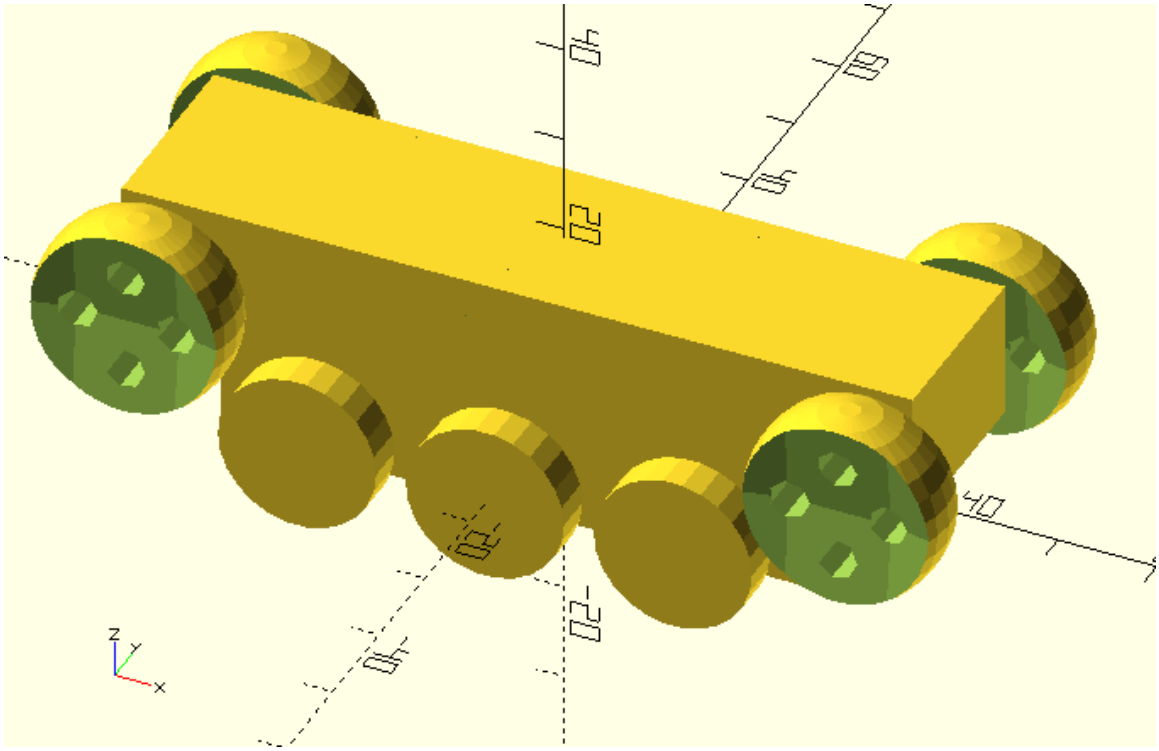
```
use <wheels.scad>

wheel_radius = 8;

wheel_width = 4;

base_height = 10;

top_height = 10;

track = 30;

// Car body base

cube([60,20,base_height],center=true);
```

```
// Car body top

translate([5,0,base_height/2+top_height/2])cube([30,20,top_height],center=true);

// Front left wheel

translate([-20,-track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

// Front right wheel

translate([-20,track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

// Rear left wheel

translate([20,-track/2,0])complex_wheel();

// Rear right wheel

translate([20,track/2,0])complex_wheel();

// Front axle

translate([-20,0,0])rotate([90,0,0])cylinder(h=track,r=2,center=true);

// Rear axle

translate([20,0,0])rotate([90,0,0])cylinder(h=track,r=2,center=true);
```



With this example it should be clear that the name of the script doesn't have to be the same as the name of the module as well as that a script can contain multiple modules. There is no right or wrong way on how you should go about organizing your library of modules. In the last example a wheel.scad script which defines the different wheel modules was used. Alternatively you could have saved each module as a separate *.scad script.

Create a vehicle_parts.scad script. Inside this script define a simple_wheel, complex_wheel, body and axle module. Use this script in another script named vehicle_concept to make the corresponding modules available. Use the modules to create a vehicle concept that looks similar to the following.




```
use <vehicle_parts.scad>

wheel_radius = 8;

wheel_width = 4;

base_length = 60;

top_length = 80;

track = 30;

wheelbase_1 = 38;

wheelbase_2 = 72;

z_offset = 10;
```

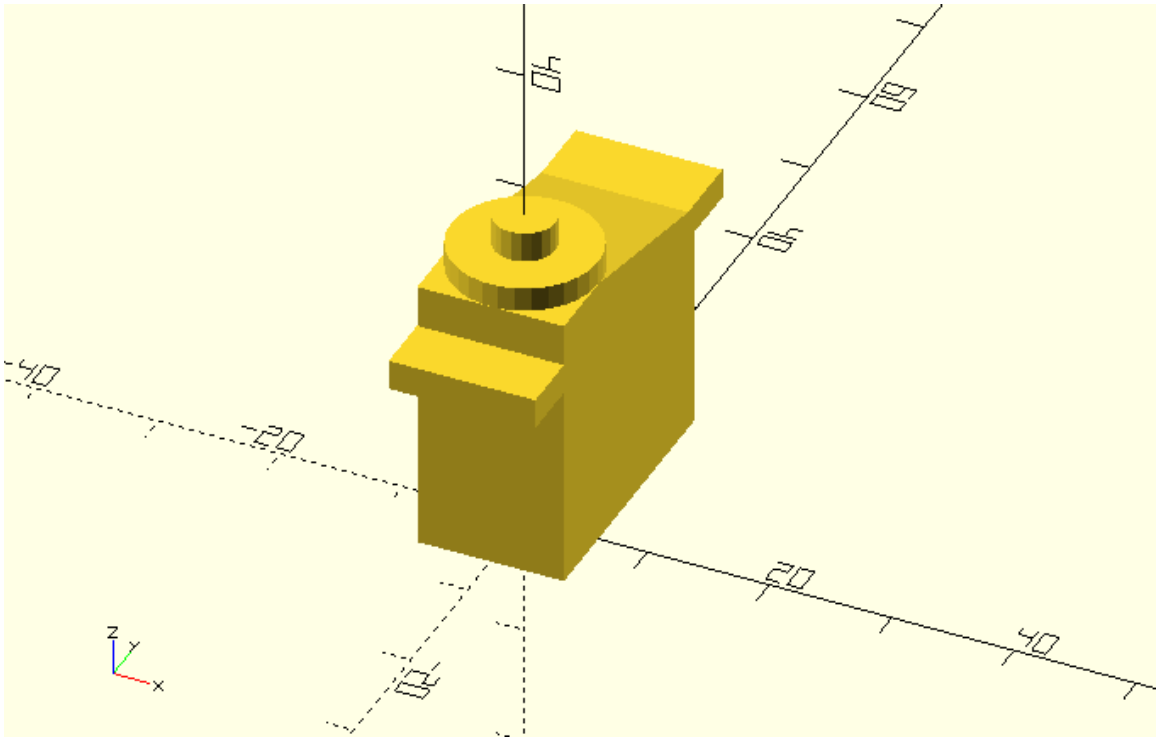
```
body(base_length=base_length, top_length=top_length, top_offset=0);

// Front left wheel

translate([-wheelbase_2/2,-track/2,z_offset])complex_wheel();

// Front right wheel

translate([-wheelbase_2/2,track/2,z_offset])complex_wheel();

// Rear left wheel

translate([wheelbase_2/2,-track/2,z_offset])complex_wheel();

// Rear right wheel

translate([wheelbase_2/2,track/2,z_offset])complex_wheel();

// Front axle

translate([-wheelbase_2/2,0,z_offset])axle(track=track);

// Rear axle

translate([wheelbase_2/2,0,z_offset])axle(track=track);


// Middle front left wheel

translate([-wheelbase_1/2,-track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

// Middle front right wheel

translate([-wheelbase_1/2,track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

// Middle left wheel

translate([0,-track/2,0])simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);

// Middle right wheel

translate([0,track/2,0])simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);

// Middle rear left wheel

translate([wheelbase_1/2,-track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

// Middle rear right wheel

translate([wheelbase_1/2,track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);
```

```
// Middle front axle

translate([-wheelbase_1/2,0,0])axle(track=track);

// Middle axle

translate([0,0,0])axle(track=track);

// Middle rear axle

translate([wheelbase_1/2,0,0])axle(track=track);
```

Using the MCAD library

The MCAD library (https://github.com/openscad/MCAD) is a library of components commonly used in mechanical designs that comes with OpenSCAD. You can utilize objects of the MCAD library by using the corresponding *.scad script and calling the desired modules. For example, there is a servos.scad script which contains the models of the Align DS420 and Futaba S3003 servo motors. The servos.scad script contains two modules, one for the Align DS420 and one for the Futaba S3003. You can open this script to check what the parameters of each module are and then use them to add the servos in your design. You can create an Align DS420 servo using the following script.

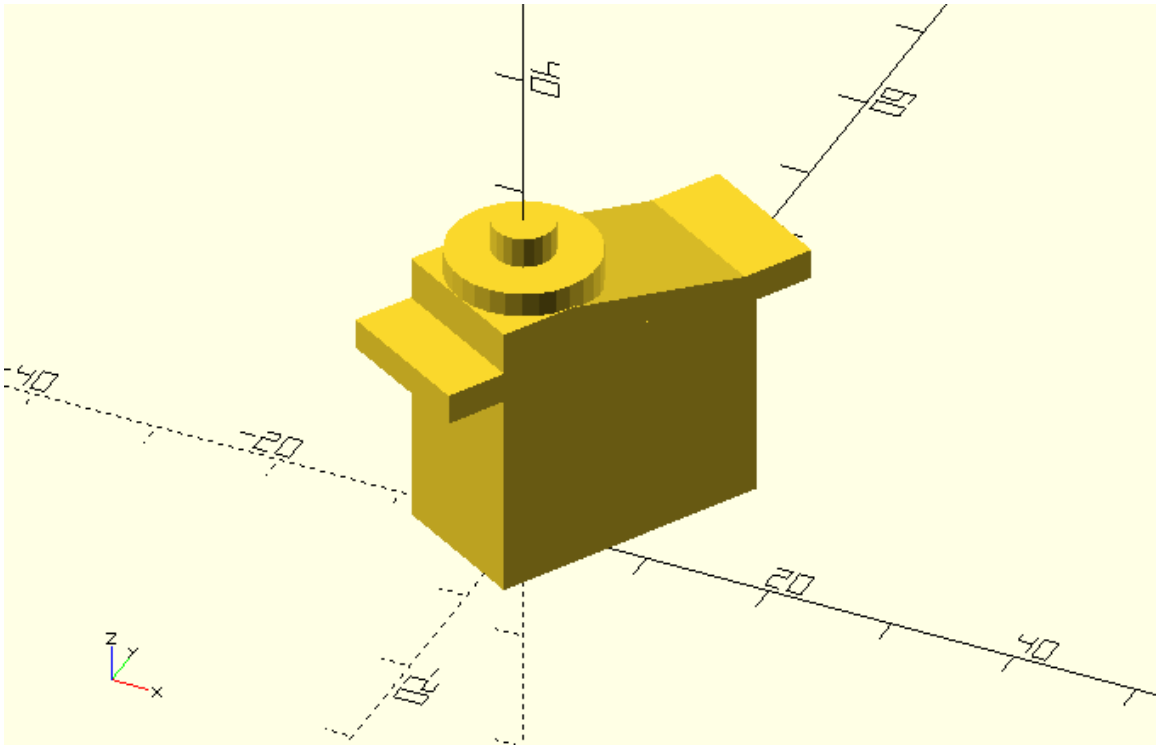
```
use <MCAD/servos.scad>

alignds420(position=[0,0,0], rotation=0);
```



The module's parameters can be used to position the servo motor.

```
use <MCAD/servos.scad>
```

alignds420(position=[0,0,0], rotation=[0,0,-30]);

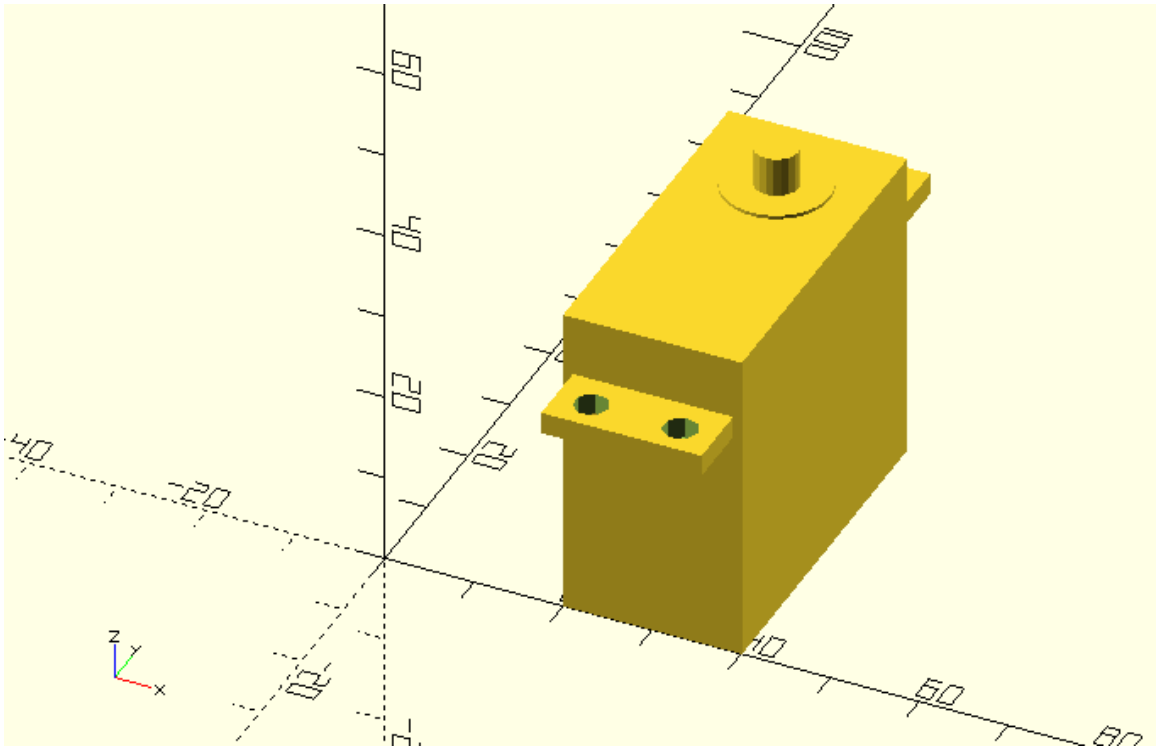


The servos.scad script is located in the MCAD directory which is under the libraries directory. The later can be found in OpenSCAD's installation folder. Should you wish to have any of your own libraries accessible from any directory, you can add it in the libraries directory. You can also browse other available OpenSCAD libraries at https://www.openscad.org/libraries.html.

Use the servos.scad script of the MCAD library to create a Futaba S3003 servo motor that is translated 20 units along the positive direction of the X axis.

use <MCAD/servos.scad>

futabas3003(position=[20,0,0], rotation=[0,0,0]);



Creating even more parameterizable modules

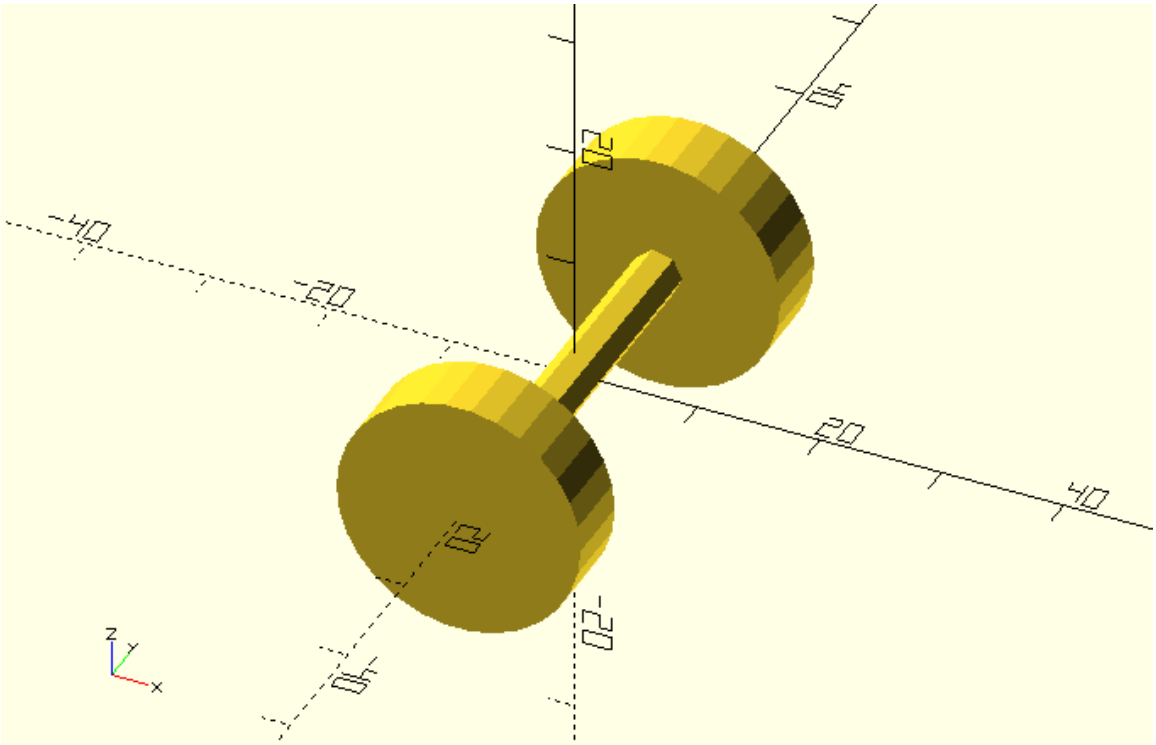
So far, the only input to the modules that have been created was through the module's input parameters that were defined for each case. The complex_wheel module for example was able to create a plethora of parameterized wheels according to the chosen input parameters such as wheel_radius, hub_thickness etc.

In your vehicle designs you have been using body, wheel and axle modules which when combined can produce various types of vehicles. In all the vehicle designs, two wheels along with an axle have been used together to form a set of wheels. You may have thought the need of an axle_wheelset module which simultaneously defines the above three objects with a single statement. And you would have been right! But there is a reason this module hasn't been created yet and you are now going to find out why now.

Throughout the previous chapters you have created two different wheel designs (simple_wheel and complex_wheel) and a single axle design. You can use your existing knowledge to combine the simple_wheel and axle modules in the following way.

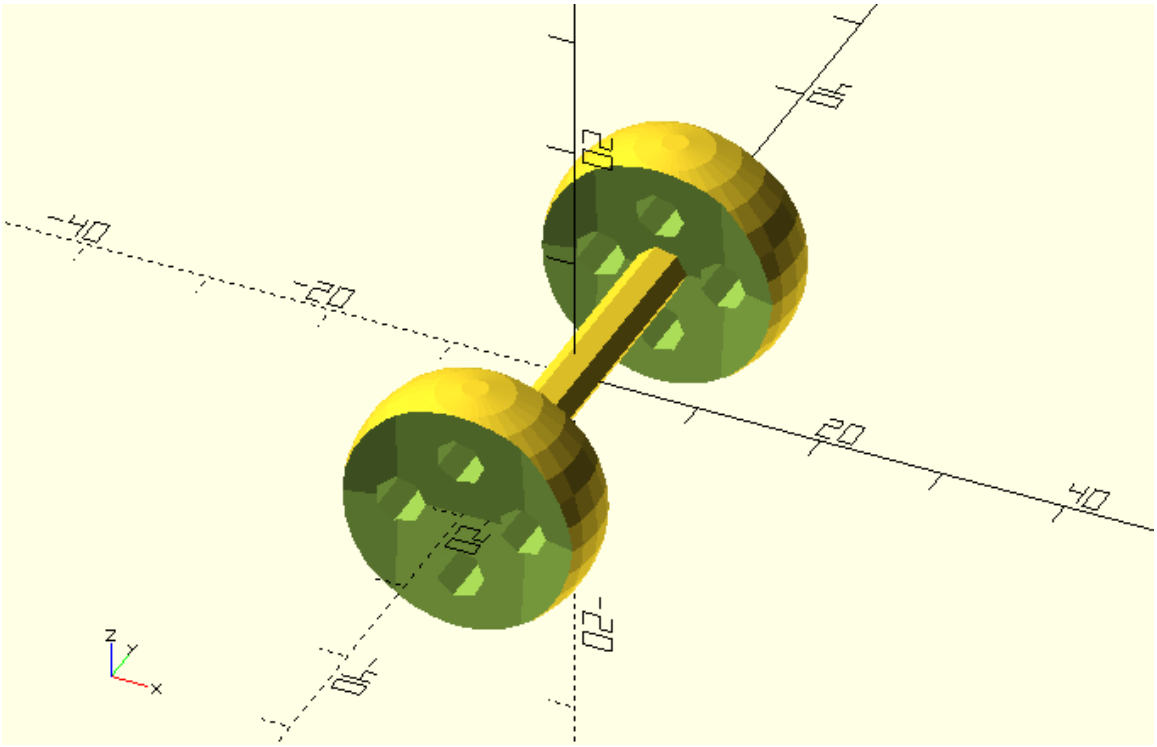
```
module axle_wheelset(wheel_radius=10, wheel_width=6, track=35, radius=2) {

   translate([0,track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

   axle(track=track, radius=radius);
```

```
    translate([0,-track/2,0])simple_wheel(wheel_radius=wheel_radius,
wheel_width=wheel_width);

}

axle_wheelset();
```



If you simply wanted to only use the above set of simple_wheels, that approach would be just fine. The problem though is that this axle_wheelset module is not really flexible and parameterizable to the desired degree. On one hand you can customize all input parameters, but on the other hand, could you swap the simple_wheel design with the complex one? The fact is that with the above approach in order to do so you would have to define a completely new module.

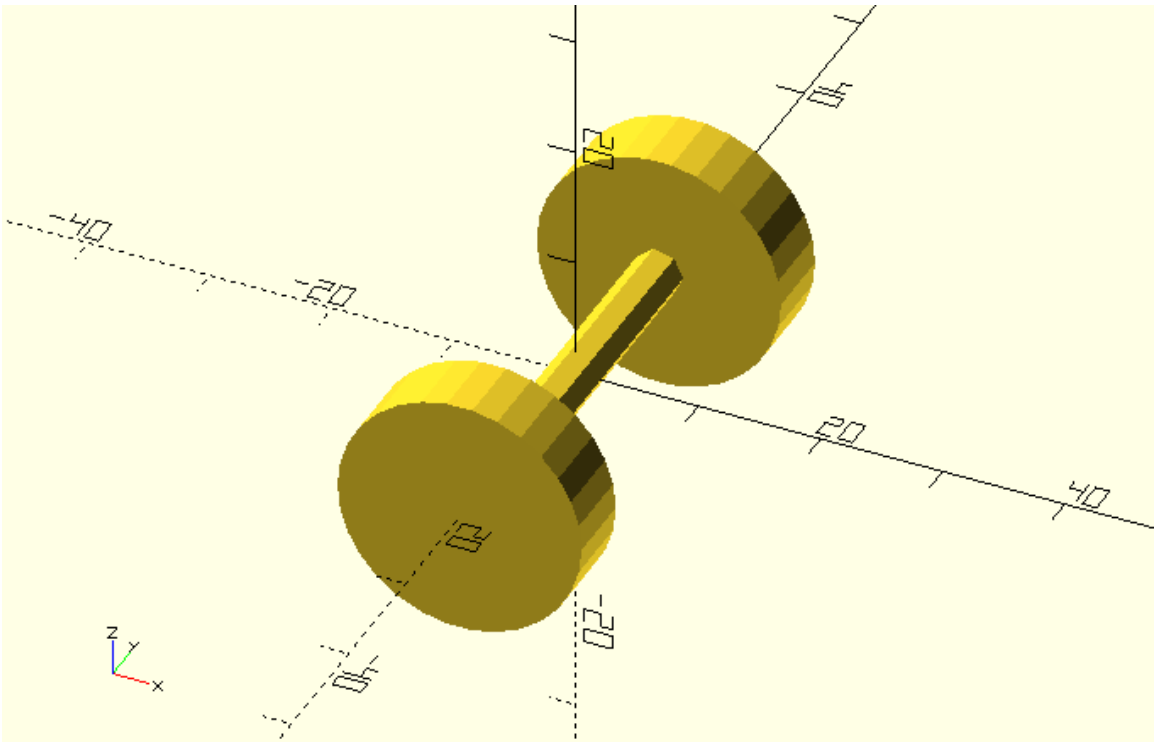
```
module axle_wheelset_complex(wheel_radius=10, side_spheres_radius=50, hub_thickness=4,
cylinder_radius=2, track=35, radius=2) {

    translate([0,track/2,0])complex_wheel(wheel_radius=10, side_spheres_radius=50,
hub_thickness=4, cylinder_radius=2);

    axle(track=track, radius=radius);

    translate([0,-track/2,0])complex_wheel(wheel_radius=10, side_spheres_radius=50,
hub_thickness=4, cylinder_radius=2);

}

axle_wheelset_complex();
```



If you can't see yet how this is a problem, image the case where you had six different wheel designs and two different axle designs in your library. If you wanted to implement the axle_wheelset module you would need to define 12 different modules to cover all combinations of wheel and axle designs. Furthermore, if you were to add a new wheel or axle design in your collection, you would need to define a number of additional axle_wheelset modules which would make maintaining your library very hard.

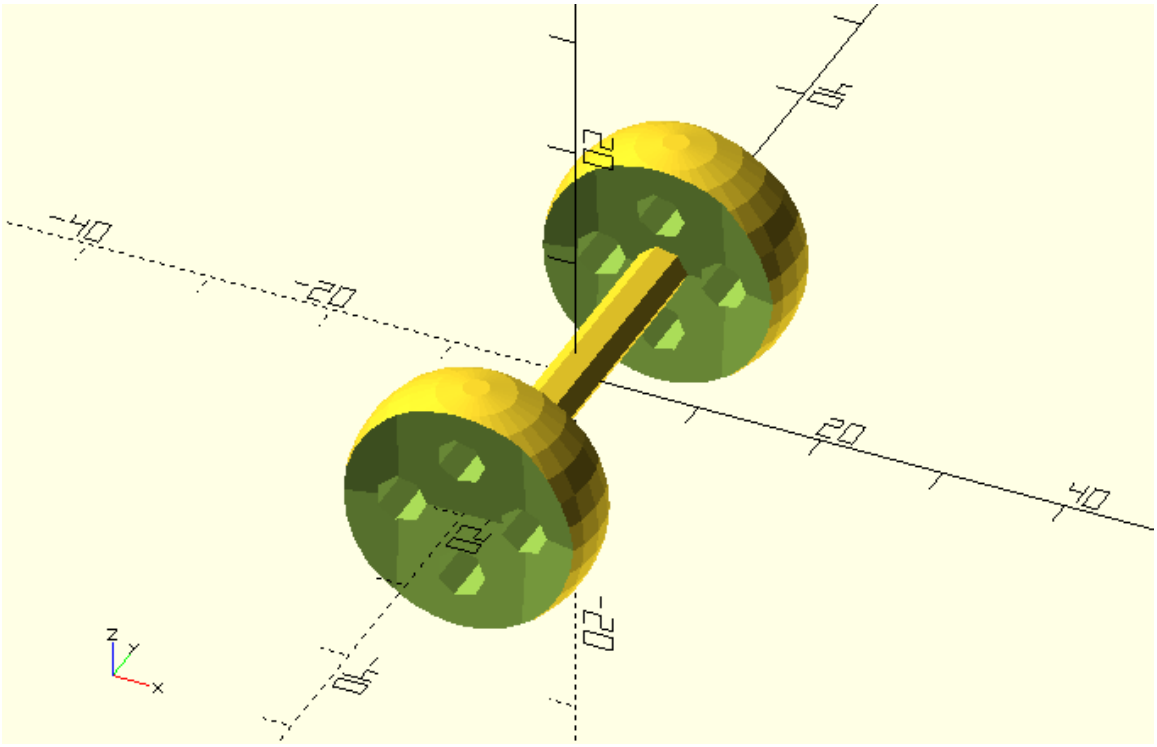
The good thing is that the two modules above look very similar. If you could keep the structure of the module the same but have the specific choice of wheel design parameterized, then the problem could be solved. Fortunately, OpenSCAD supports this functionality and parameterizing the specific choice of wheel design can be achieved in the following way.

```
module axle_wheelset(track=35, radius=2) {
   translate([0,track/2,0])children(0);
   axle(track=track, radius=radius);
   translate([0,-track/2,0])children(0);
}
axle_wheelset(){
   simple_wheel();
}
```

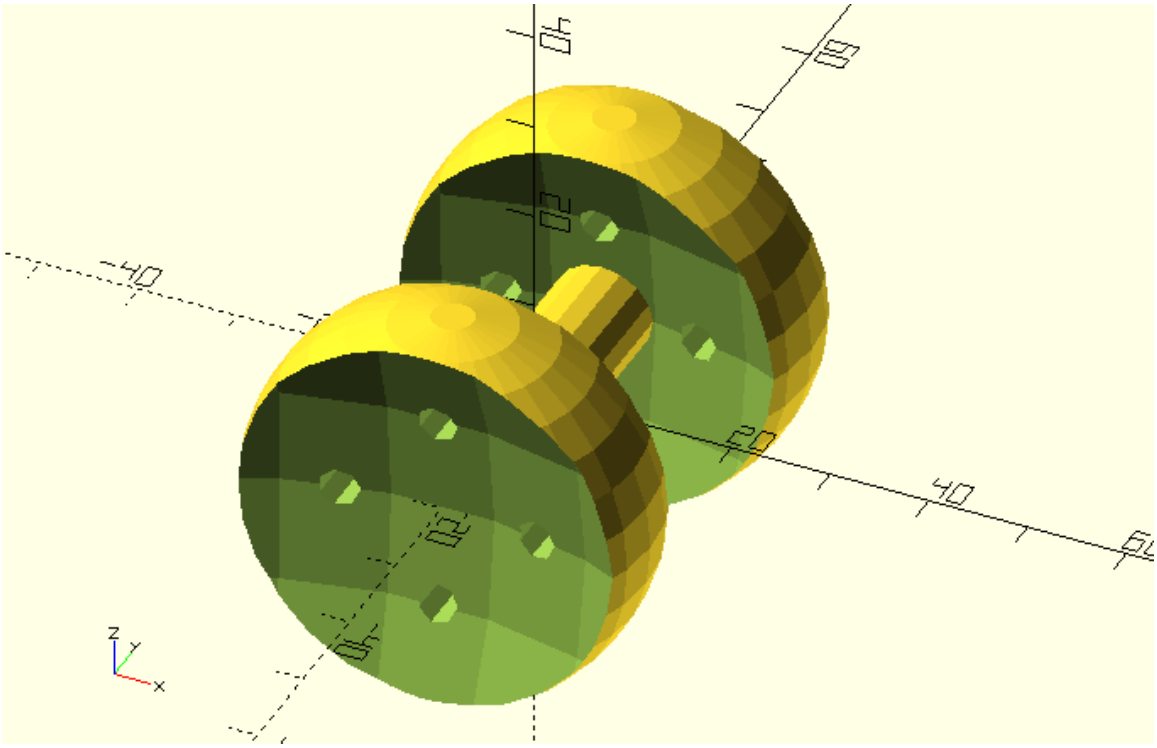


The wheel design can now be effortlessly changed, making this module a truly parametric one.

```
axle_wheelset(){
   complex_wheel();
}
```



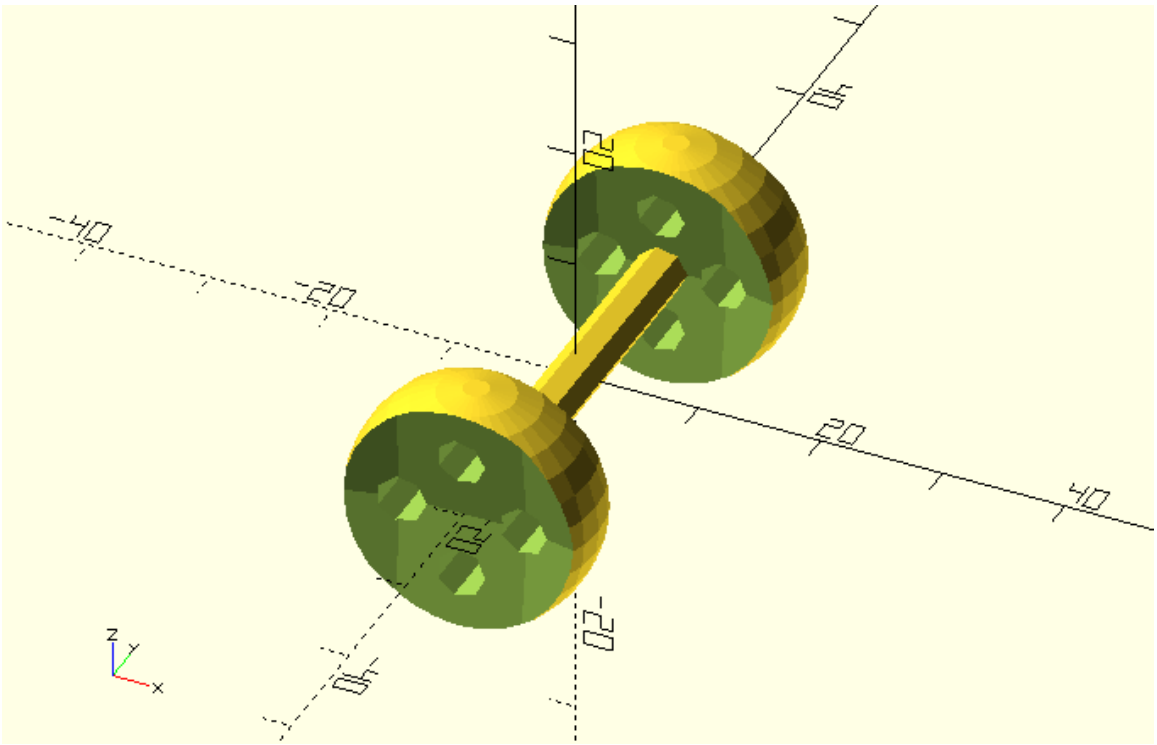

```
axle_wheelset(radius=5){
    complex_wheel(wheel_radius=20);
}
```



There is a very important concept that you should grasp here. The first thing you should notice is the definition of this new module. This new module is similar to the previous ones with the difference that the command children(0) is used in place of a call to a specific wheel module. The second thing you should notice is the call to the axle_wheelset module. The call to the axle_wheelset module contains a pair of curly brackets inside of which the specific wheel design to be used by module is defined each time. OpenSCAD keeps an ordered list of the objects that have been defined inside the curly brackets and numbers them starting from zero. These objects can then be referenced by the children command. The number that is passed inside the children command corresponds to the first, second, third etc. object that was defined inside the curly brackets, counting from zero. In the above example, only one object is defined inside the curly brackets. That is either a simple_wheel or a complex wheel object. This object is created every time the children(0) command is used. The children command is in essence a way to pass objects as input to a module.

The next examples can help make this concept more concrete. In the previous example there is no way to use the axle_wheelset module and end up creating an axle that has different wheels on each side. This would not happen even if you passed/defined two different objects inside the curly brackets when calling the axle_wheelset module, because only the first one, children(0), is referenced for both sides of the axle.

```
axle_wheelset(){

   complex_wheel();

   simple_wheel();

}
```

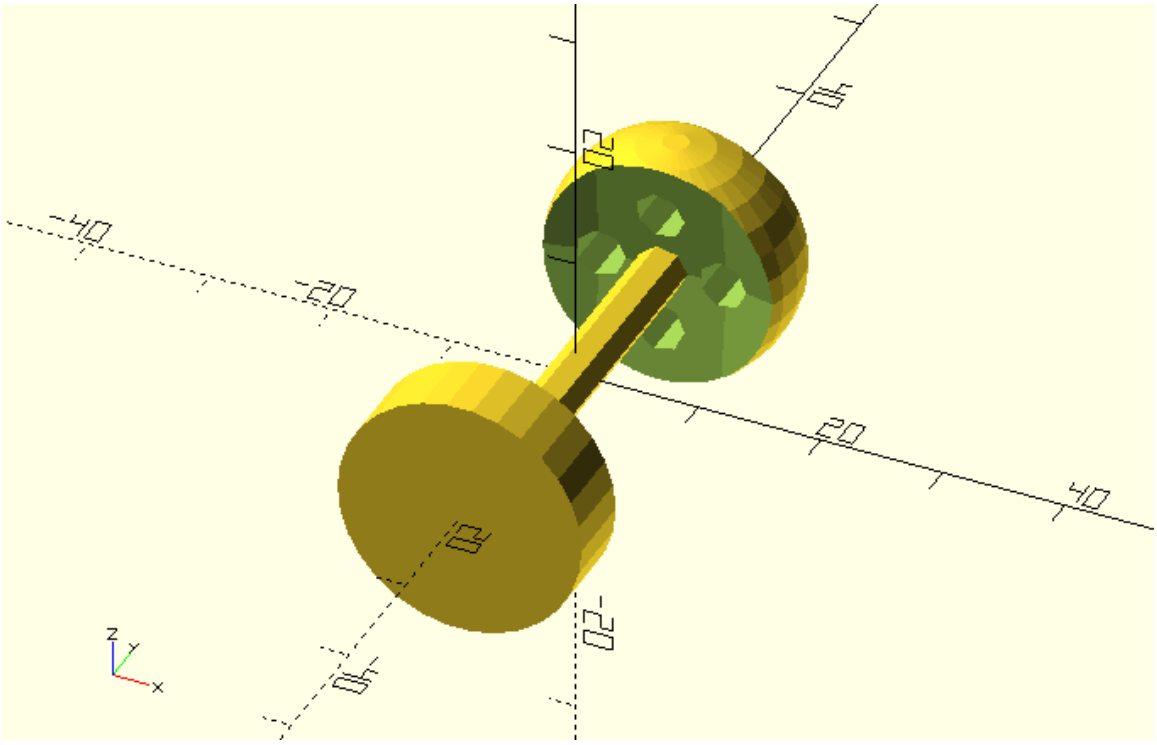


In order to create an axle with different wheels on each side, the definition of the of the axle_wheelset module would need to be modified. Instead of referencing the first object, children(0), for both sides, the axle_wheelset module would need to reference the first object, children(0), for one side and the second object, children(1), for the second side.

```
module axle_wheelset(track=35, radius=2) {
   translate([0,track/2,0])children(0);
   axle(track=track, radius=radius);
   translate([0,-track/2,0])children(1);
}
```

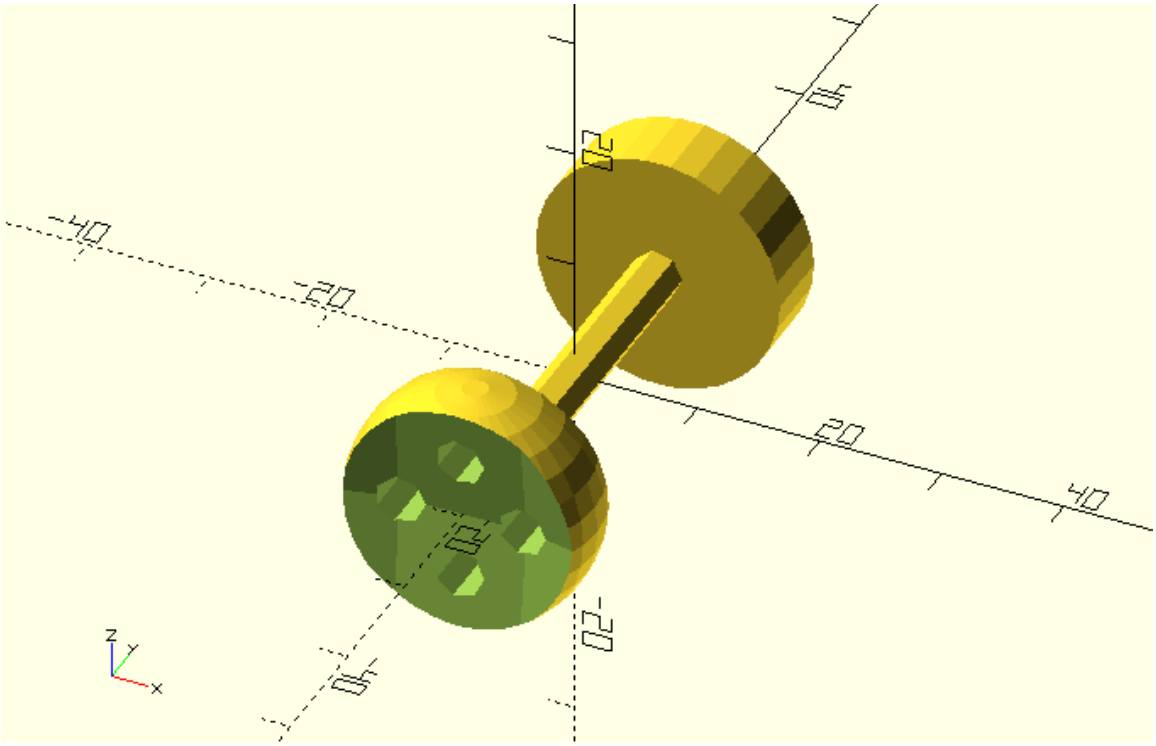
By defining two different wheel objects inside the curly brackets, the following model would be created.

```
axle_wheelset(){
   complex_wheel();
   simple_wheel();
}
```



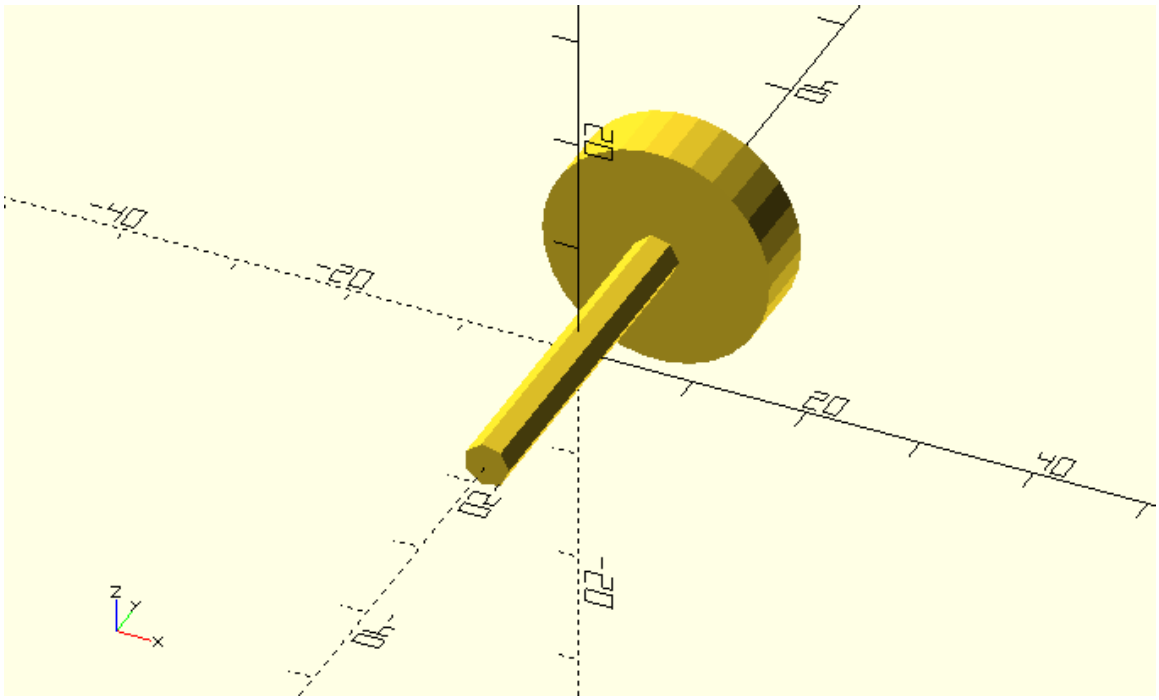
Try swapping the order in which the wheels are defined inside the curly brackets when calling the axle_wheelset module. What happens?

```
axle_wheelset(){
   simple_wheel();
   complex_wheel();
}
```



Try defining only one wheel inside the curly brackets? Do you get an error message?

```
axle_wheelset(){
   simple_wheel();
}
```



Add an axle_wheel module on the vehicle_parts.scad script. Make use of the children command to parameterize the specific choice of wheel design. Use the vehicle_parts.scad script on another script to create any vehicle deign that you like.

Challenge

The material you have been learning in the last two chapters gives you a powerful set of tools to start creating your own library of objects that can be flexibly combined and customized to create new designs.

Think of any model that you would like to create. Break it down into different parts. Come up with alternative designs for each part and define modules that create them. What should the input parameters of each module be? Define one or more modules using the children functionality in order to flexibly combine the various parts that you have created.