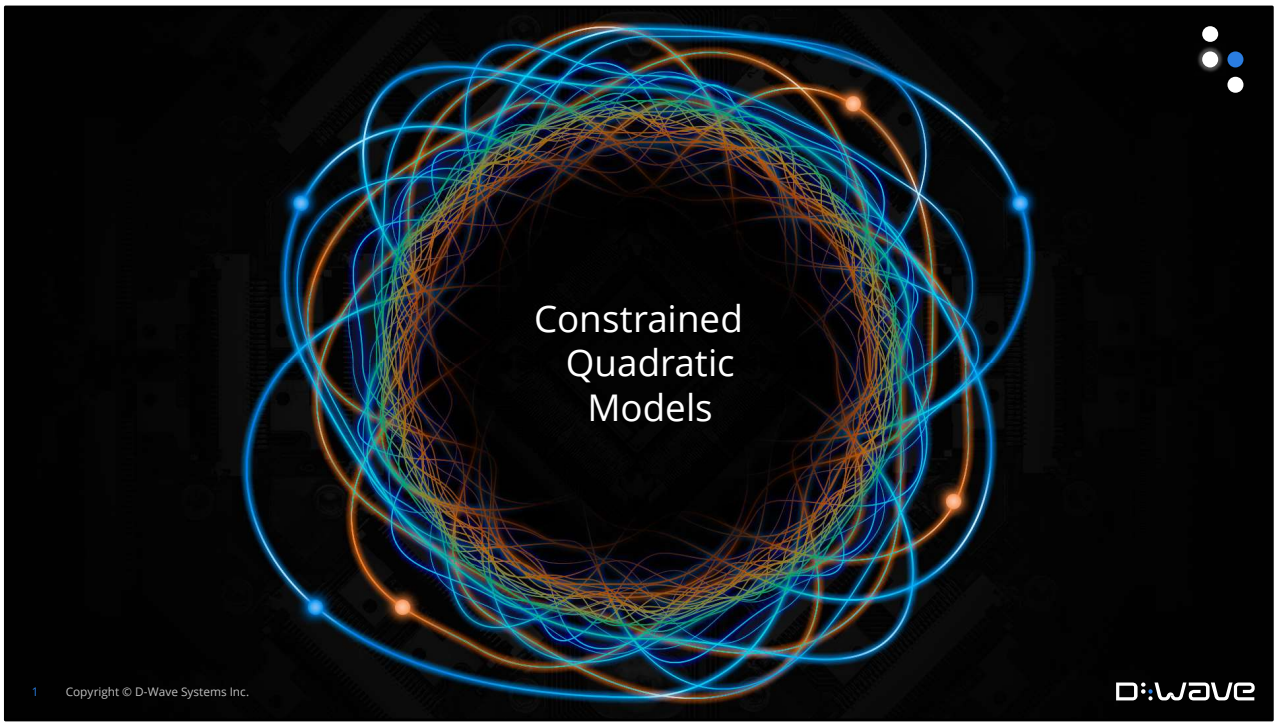

Constrained
Quadratic
Models
1
Copyright © D-Wave Systems Inc.
D:Wave

Constrained Quadratic Model Hybrid Solver

An Expanded Hybrid Solver Portfolio

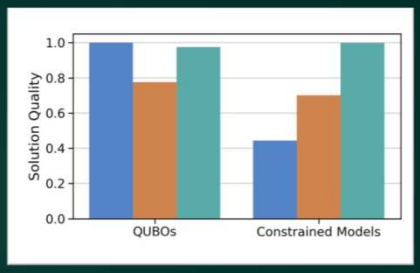
- The best hybrid choice for constrained problems with non-binary variables

More Native Representation of Problem

- Formulate constraints directly instead of using penalties or transformations

Unlocks Larger Application Problems

- Binary and integer variables
- Linear and quadratic terms
- Up to 100,000 constraints
- Inequality & equality constraints



Performance of BQM vs DQM vs CQM on problems native to each:

CQM solver is the best hybrid choice for constrained problems on non-binary variables

BQM solver is the best hybrid choice for unconstrained binary problems

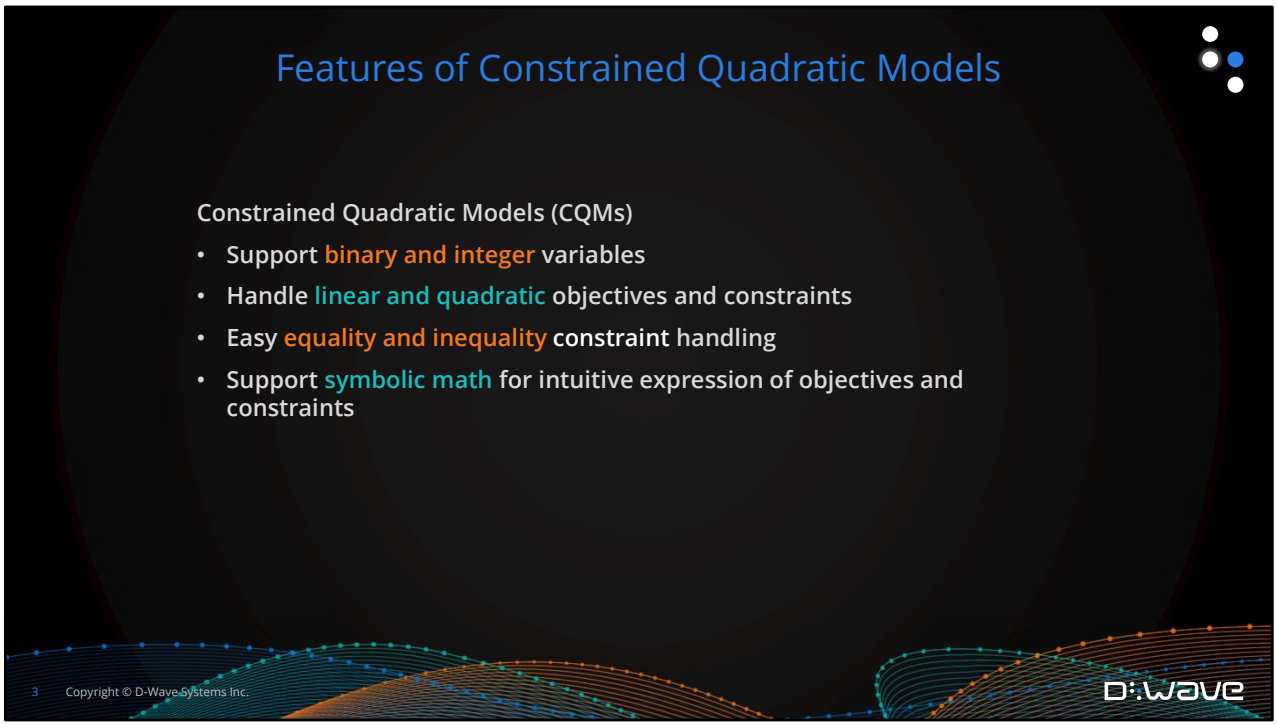
BQM Solver
DQM Solver
CQM Solver

The CQM solver is another hybrid solver available in Leap. As shown in the diagram, it's the best hybrid solver for constrained problems. Interestingly, it also does pretty well on QUBOs.

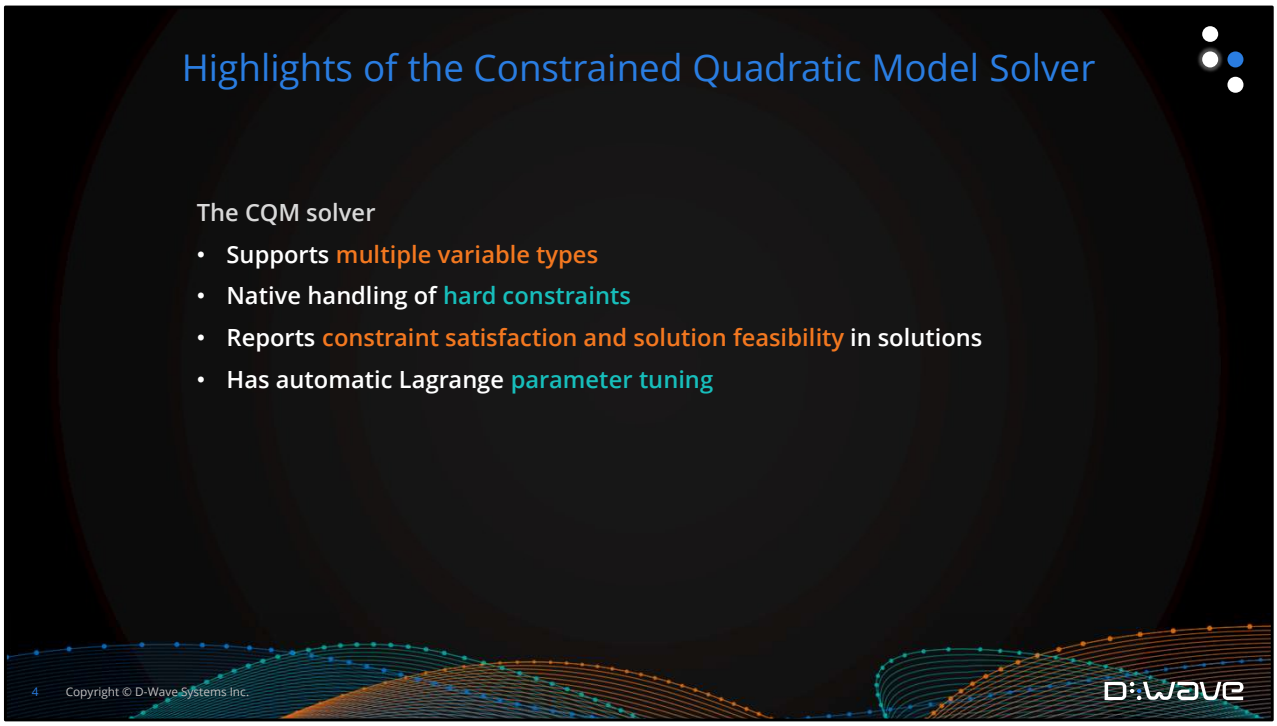
It supports binary and integer-valued problems, as well as models that have a combination of variable types. For example, portfolio optimization uses integer variables and job shop scheduling can use both – binary for selecting machines and integers to determine which time interval the machine and jobs should be matched in.

The CQM object in Ocean allows us to formulate and add constraints to the model more intuitively, especially with the introduction of symbolic math in Ocean 4.

It natively supports up to 100 000 equality and inequality constraints, with both linear and quadratic terms.

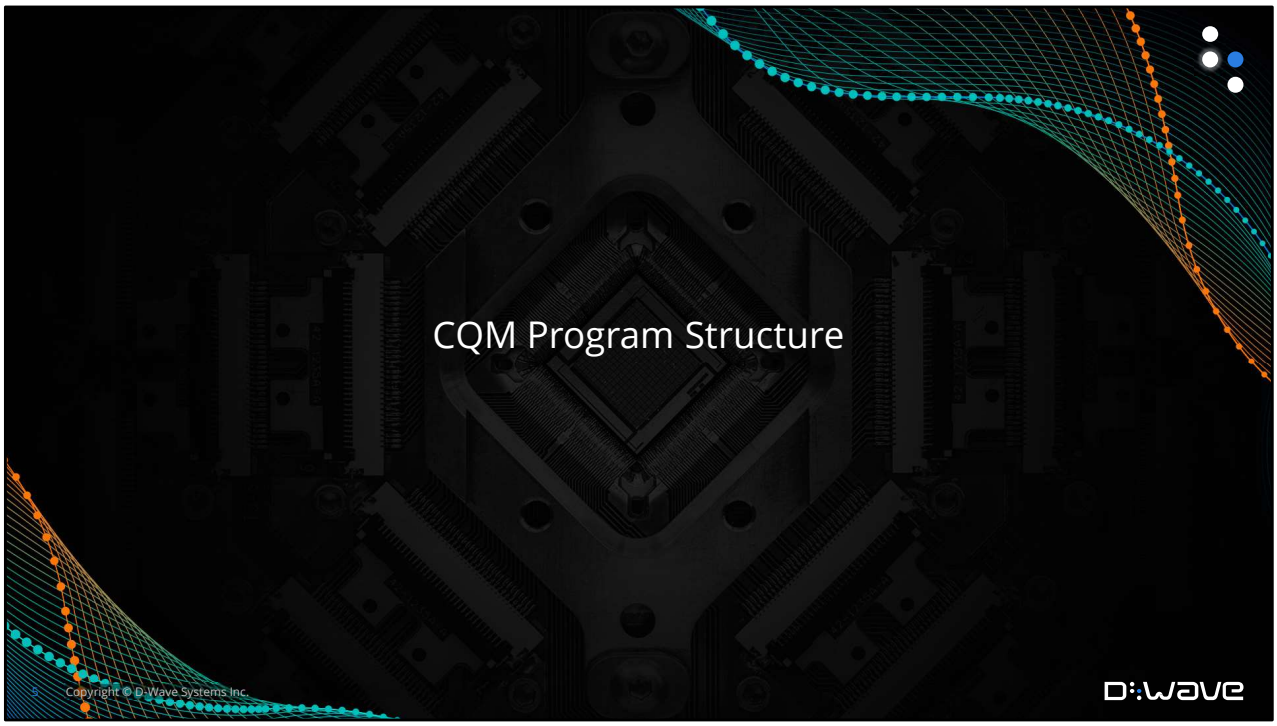


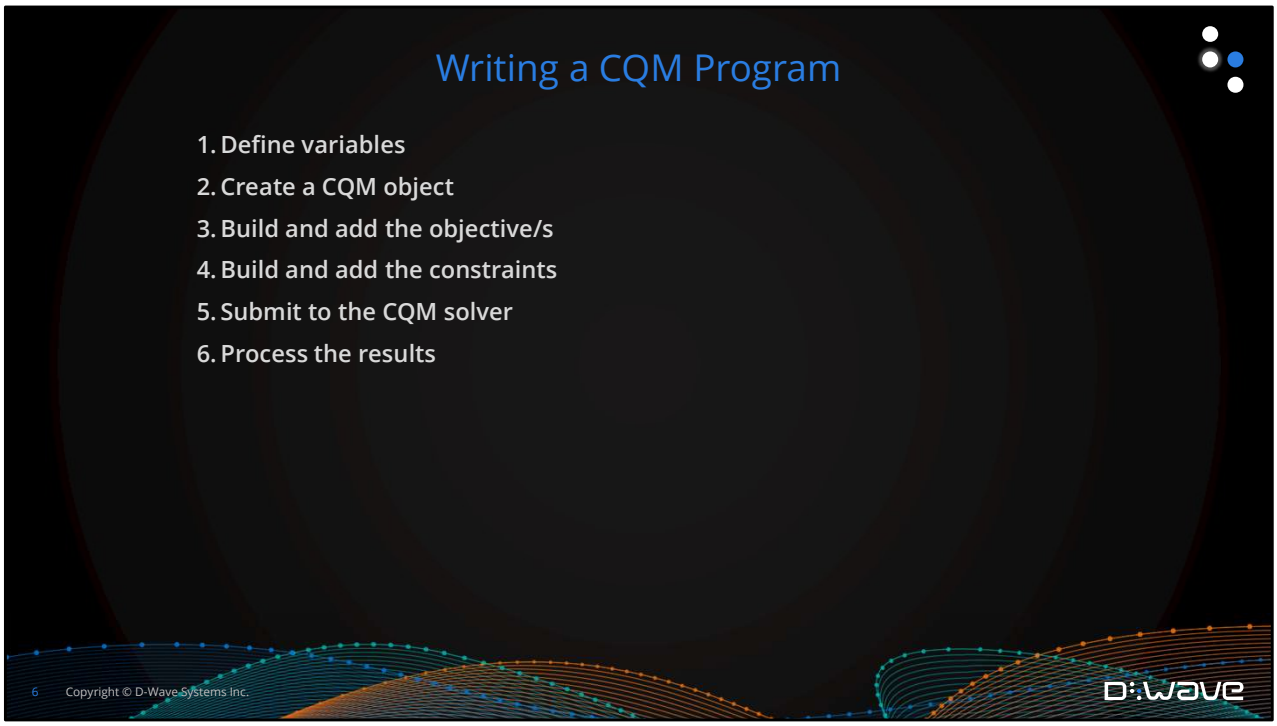
The constrained quadratic model in Ocean has a few key features.



The highlights of the CQM solver are

1. It supports multiple variable types
2. Handles hard constraints natively, which means that all you need to do is define the constraints as equality or inequality expressions, not build the penalty models
3. Reports constraint satisfaction and solution feasibility


CQM Program Structure
5
Copyright © D-Wave Systems Inc.
D:Wave



Writing a CQM program is very similar to writing a program that uses any of our other solvers. We're going to talk about how to instantiate and define variables, a CQM object, and objectives and constraints (though not quite in the order outlined here)

1. Define variables
2. Create a CQM object
3. Build and add the objective/s
4. Build and add the constraints
5. Submit to the CQM solver
6. Process the results


CQM Object and Solver
7
Copyright © D-Wave Systems Inc.
D:Wave

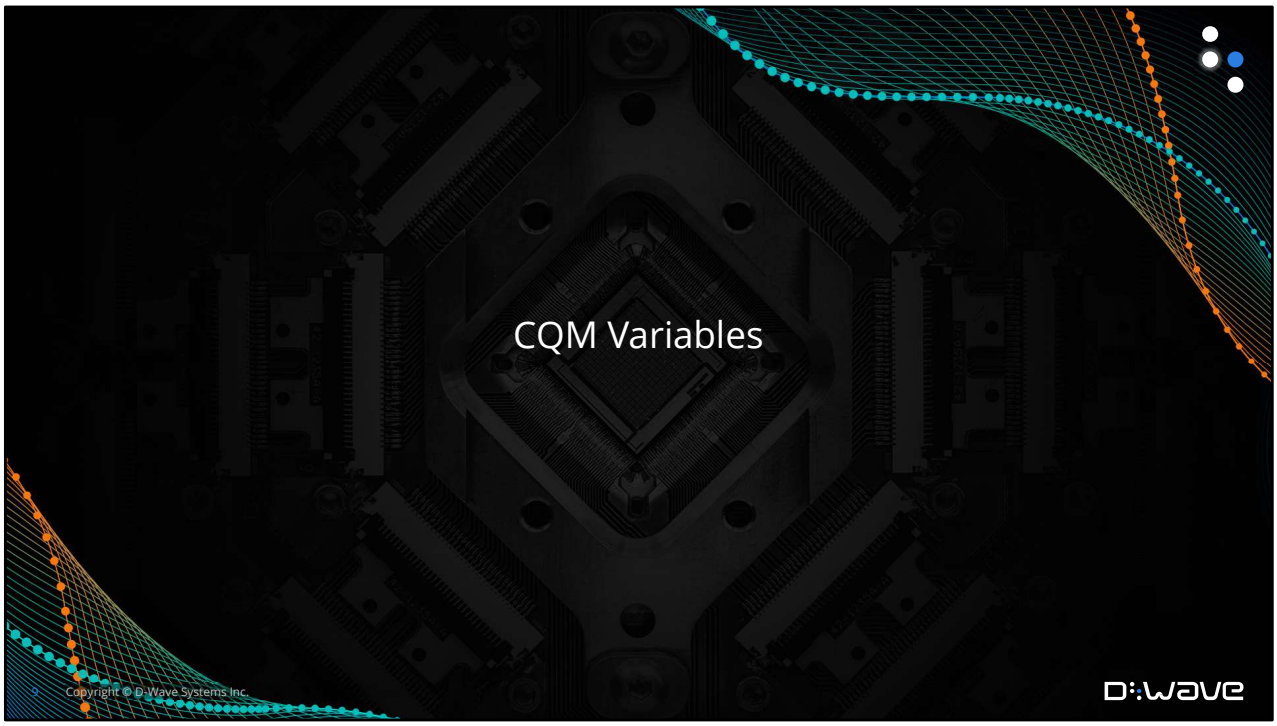
Instantiating a CQM object and sampler

CQM object is in dimod

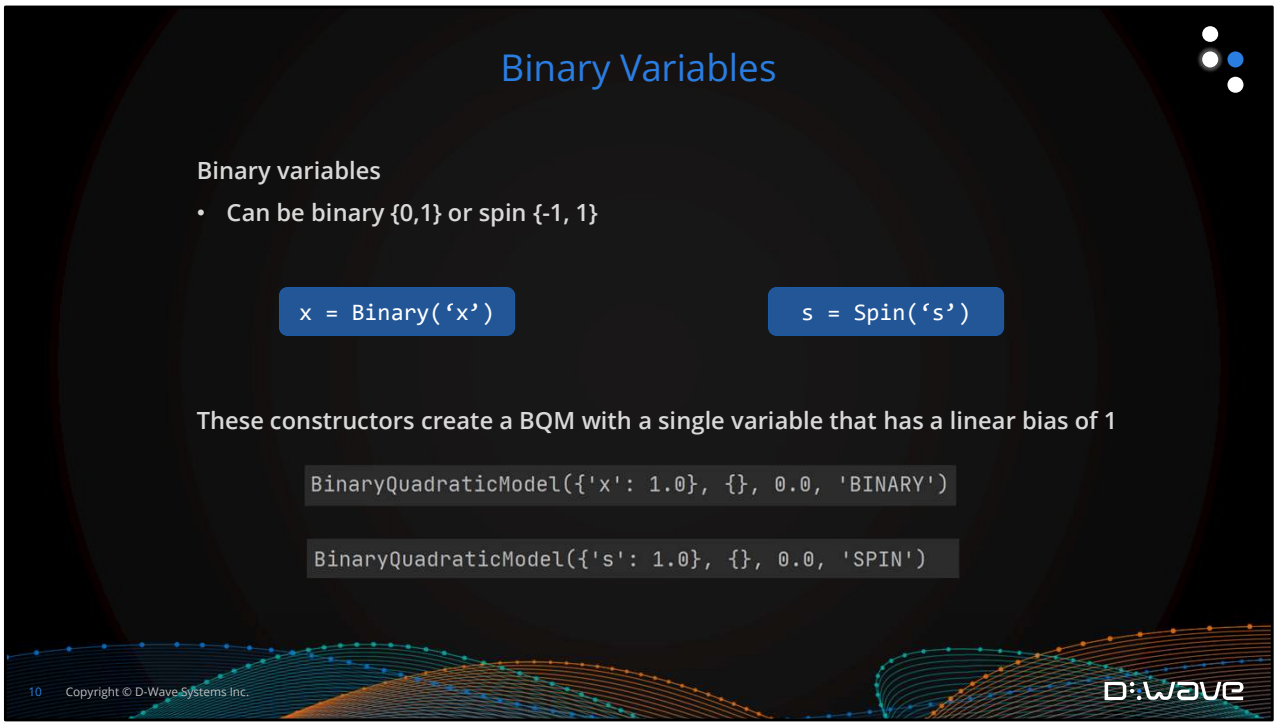
```
from dimod import CQM
cqm = CQM()
```

CQM sampler is in dwave.system

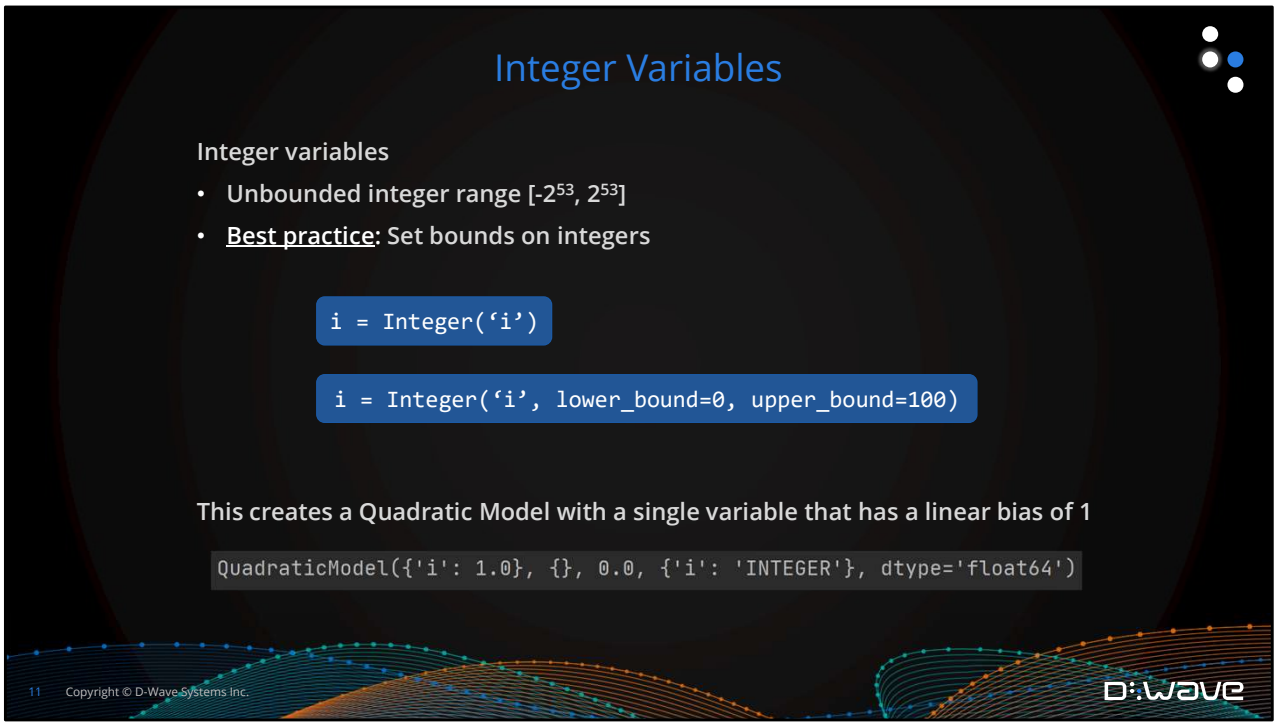
```
from dwave.system import LeapHybridCQMSampler
sampler = LeapHybridCQMSampler()
```



When working with CQM we need to explicitly define our variables.



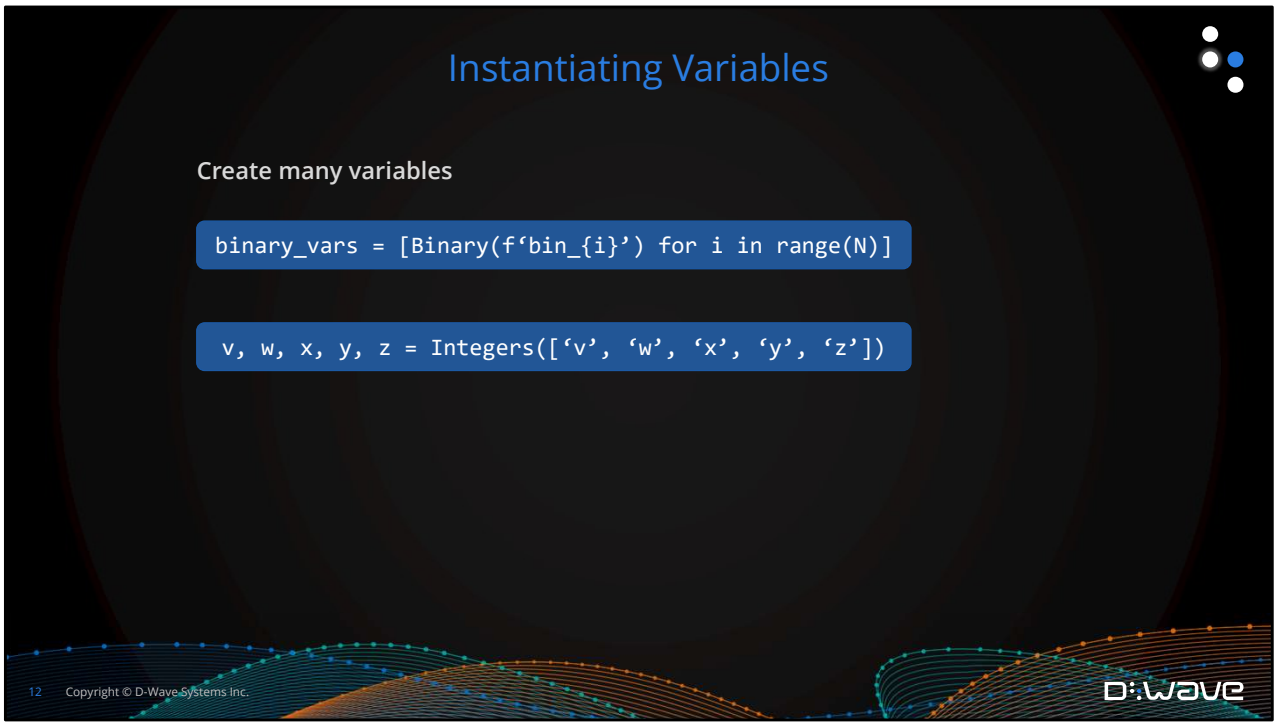
For binary variables, we can work in the binary or spin space. In dimod there are constructors that we can use to create Binary or Spin variables.
These constructors create a BQM with a single variable that has a linear bias of 1
Notice that the string in the parentheses becomes the variable's label in the BQM



Likewise, we can create integer variables using the Integer constructor in dimod. We can also introduce lower and upper bounds on our variables.
If integer variables are unbounded they occupy the range of -2^53 to 2^53 (corresponds to the largest integer that can be stored in a double)

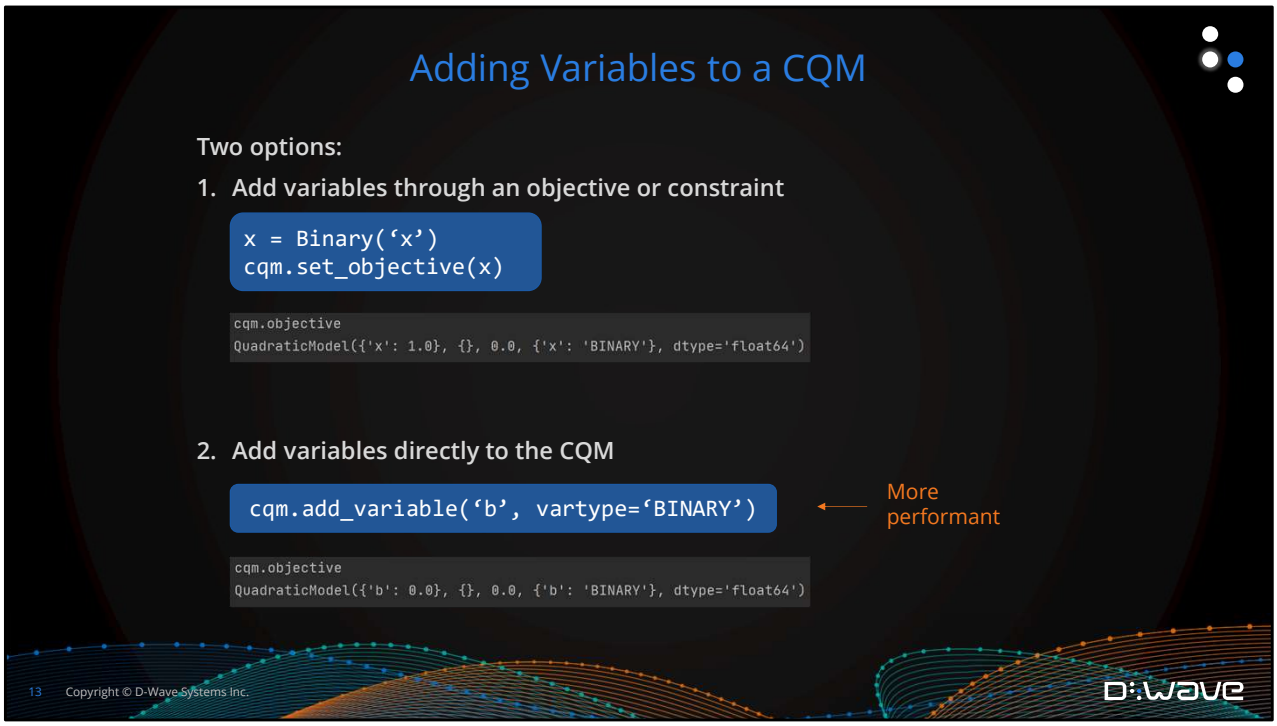
It's best to set bounds on integers, especially if the constraints don't naturally bound them. For example, an equality or inequality constraint on all integer values acts as a natural bound.
Bounds shrink the solution space which is helpful for the solver.

An integer variable is defined as a quadratic model that contains one variable with a linear bias of 1



It's also possible to instantiate a list of variables or multiple variables in a single call.

Creating a list of variables is helpful for using symbolic math as you'll see in later slides.
This is the suggested way of working with variables and constructing CQMs when you're becoming familiar with this model.



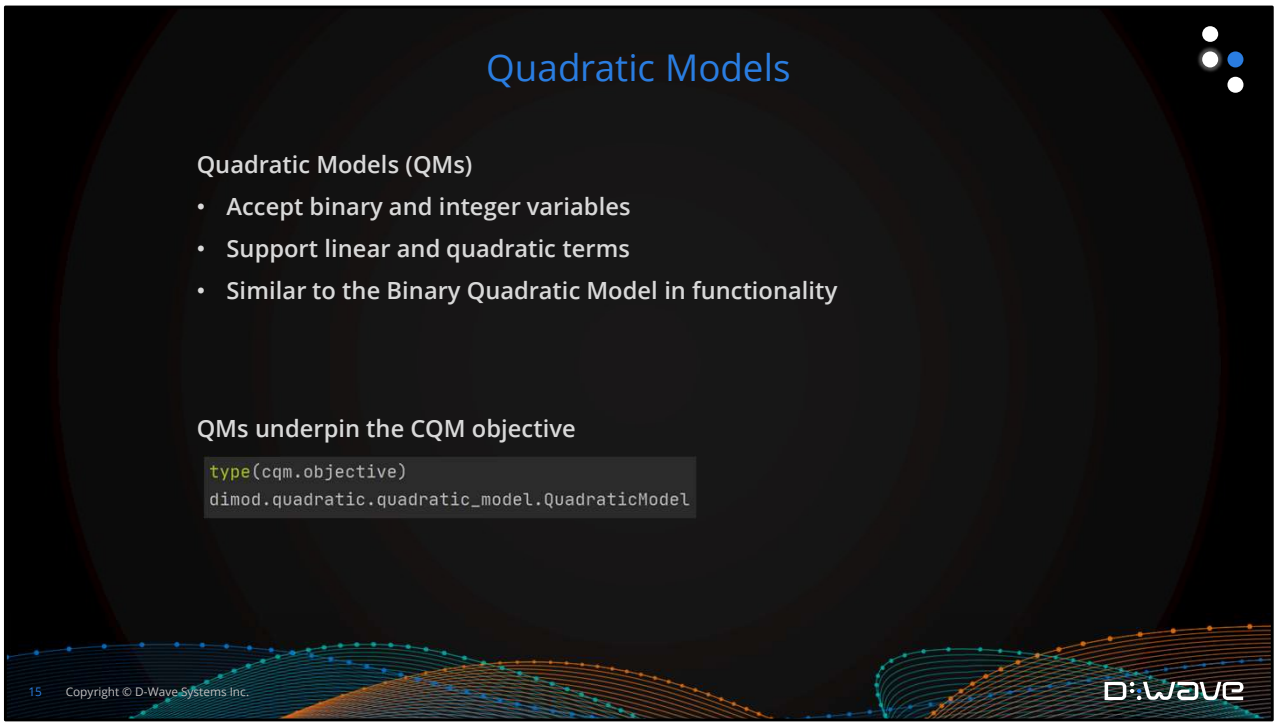
There are two ways to add variables to a CQM.

The first is by defining the variables like we saw on the previous slides and then adding them to the CQM through an objective or a constraint.
In the first example, we define a single binary variable x and add it to the CQM through the objective.
The result of the code snippet is a CQM that has an objective with a single variable x.

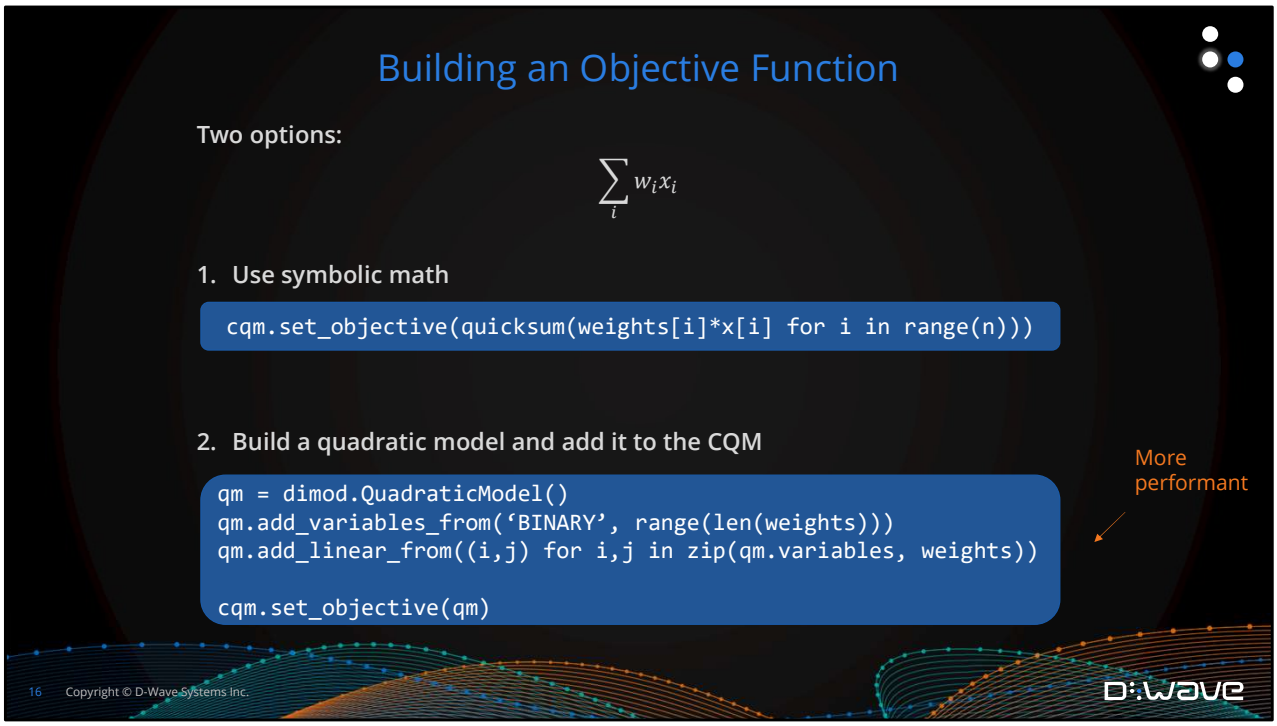
The second way of adding variables to a CQM is through the add_variable function.
This adds a variable that has a linear bias of 0 to the CQM's objective.

The first method tends to be easier to use since it allows us to build objectives and constraints using symbolic math.
The second method is more performant though. So as you scale your CQM problems, this will allow for faster CQM construction.

Let’s look at how to build an objective function now.



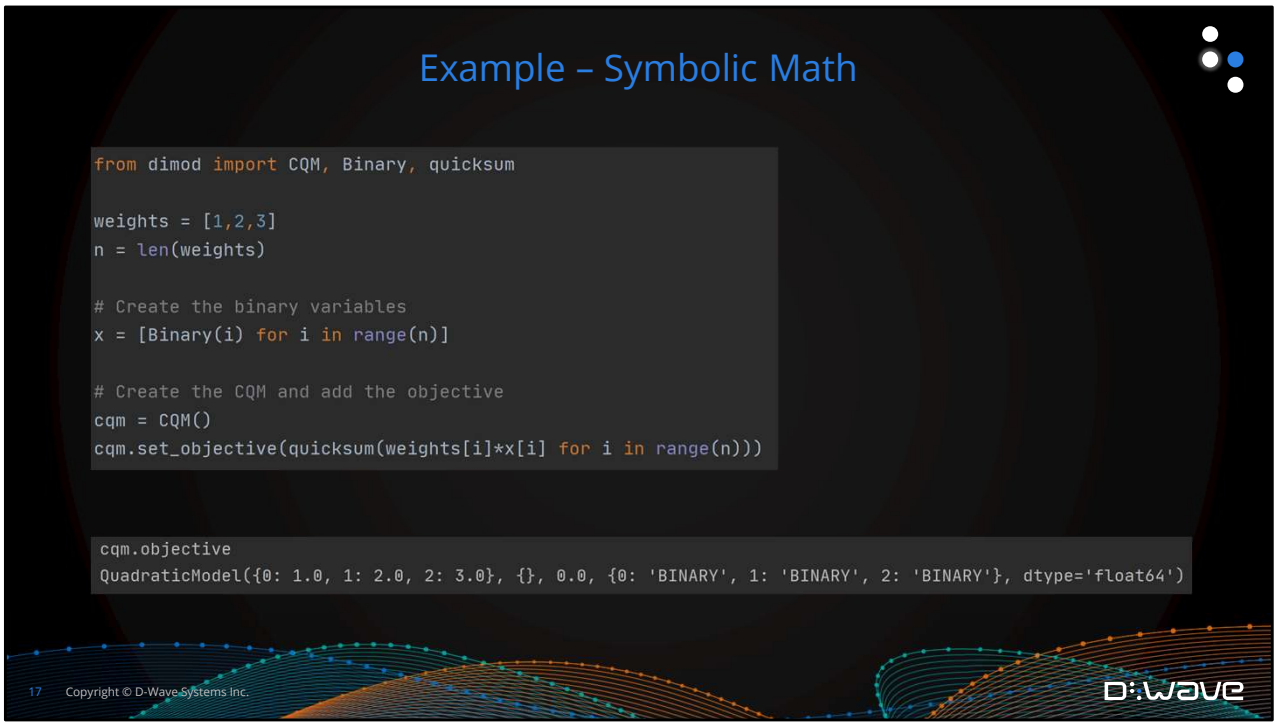
Objective functions in CQMs are stored as quadratic models.
QMs accept binary and integer variables, support linear and quadratic terms, and are similar to the Binary Quadratic Model in functionality



There are two ways we can build and add objective functions to CQMs.
Let's say we have the objective function shown here, where we want to minimize the sum of weighted linear variables.

The first option uses symbolic math.
If x is a list of binary variables, weights is a list of integer weights and n is the number of binary variables we can define and set the objective function in one line.
The quicksum function is an optimized version of python's sum function. We can use it to construct objective functions and constraints
We can use that to build the linear objective shown.

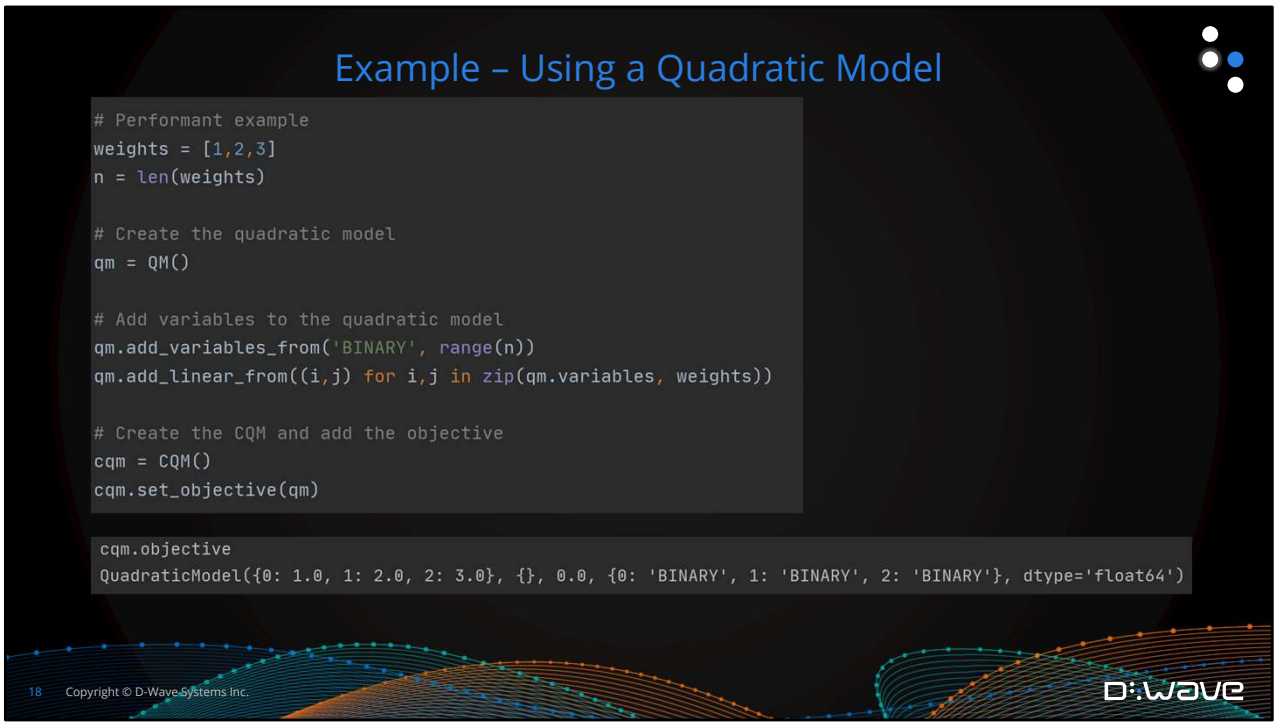
The second option constructs a quadratic model, adds the variables in the objective function, and then adds the linear weights.
Once we have a quadratic model we can add it to the CQM with set_objective()
This option is more performant in the sense that it's faster than using symbolic math.
It's a good choice when you're working with large models, however for this course we encourage the first option since it's or intuitive or closer to the math expression that's being added.



Let's look at a simple code example that adds up 3 binary variables that have the linear weights 1, 2 and 3.

Notice how we create a list of binary variables and then use quicksum to construct and set the CQM's objective.

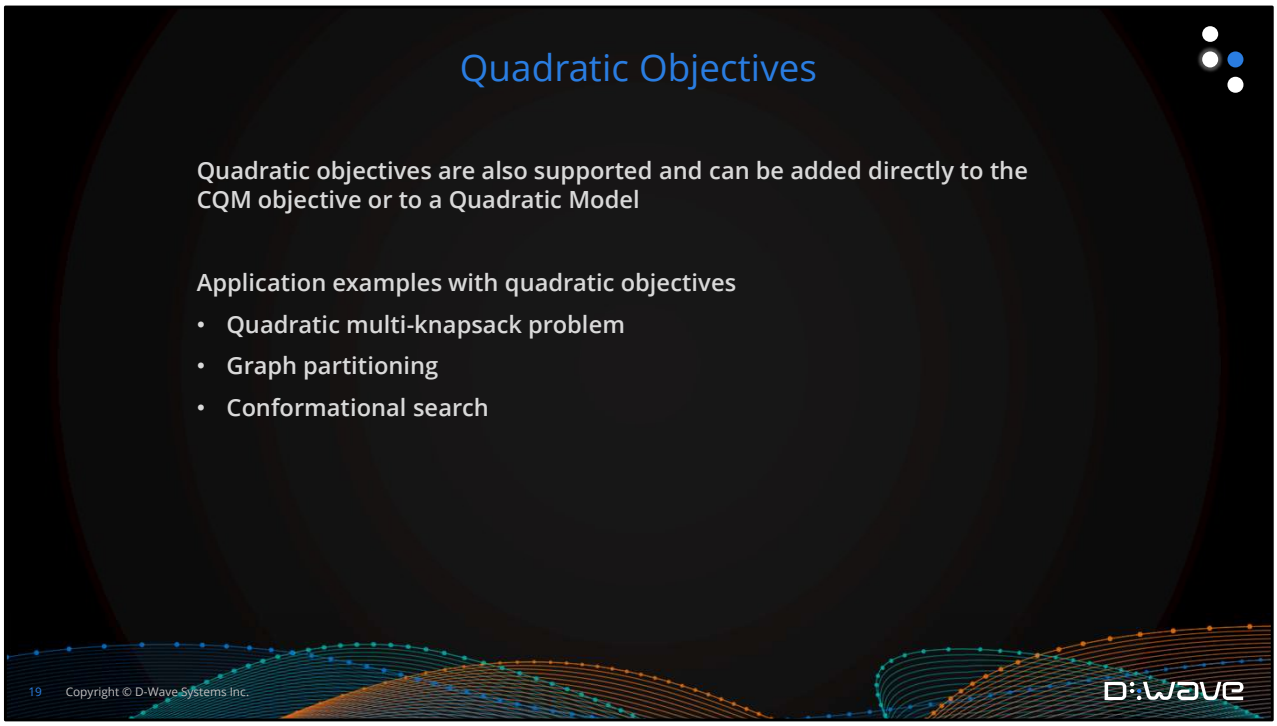
The second code box shows the CQM's objective function once it's been added. We can see that it's a quadratic model, with three binary variables. Each variable has one of the weights we specified in the code above.



Now let's take a look at how we can implement this same problem with the second approach.

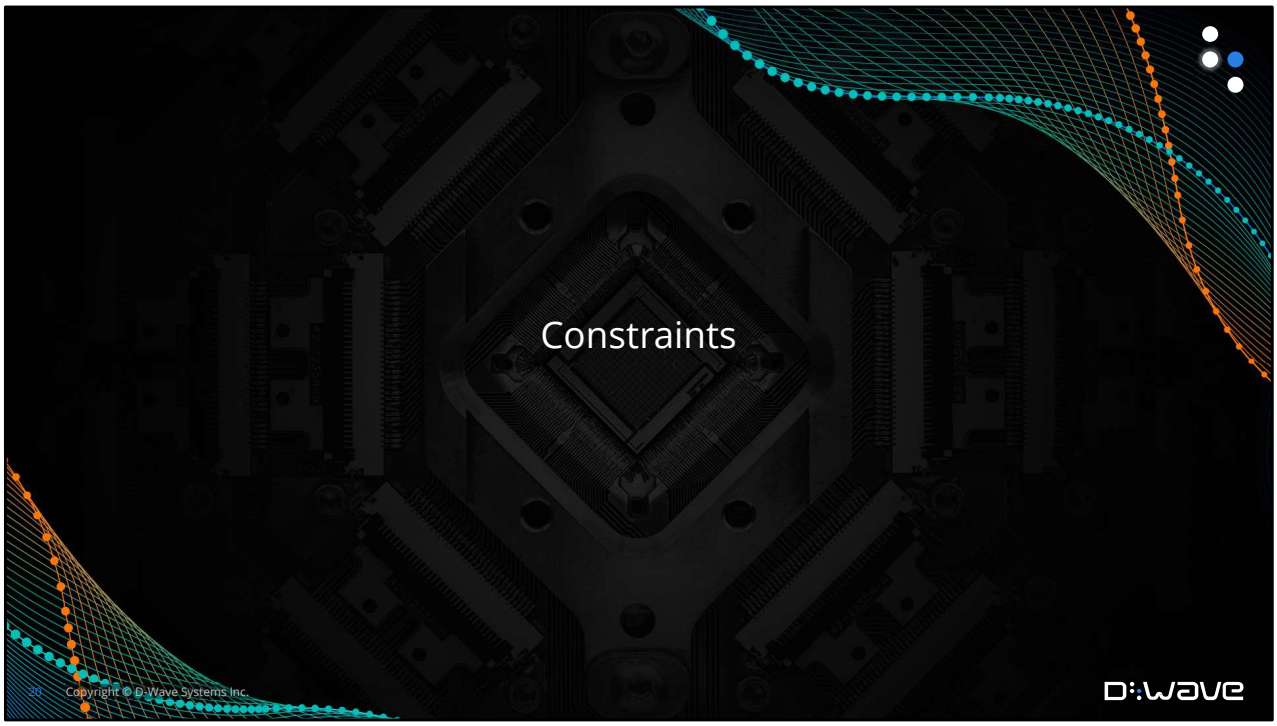
Here we instantiate a quadratic model, add unweighted variables and then add the linear weights on those variables.
We set the CQM's objective by passing the quadratic model to the set_objective function.

Notice that the CQM's objective is the exact same as on the previous slide.
The output of both of these options is the same. They just take a different amount of time to construct the CQM.

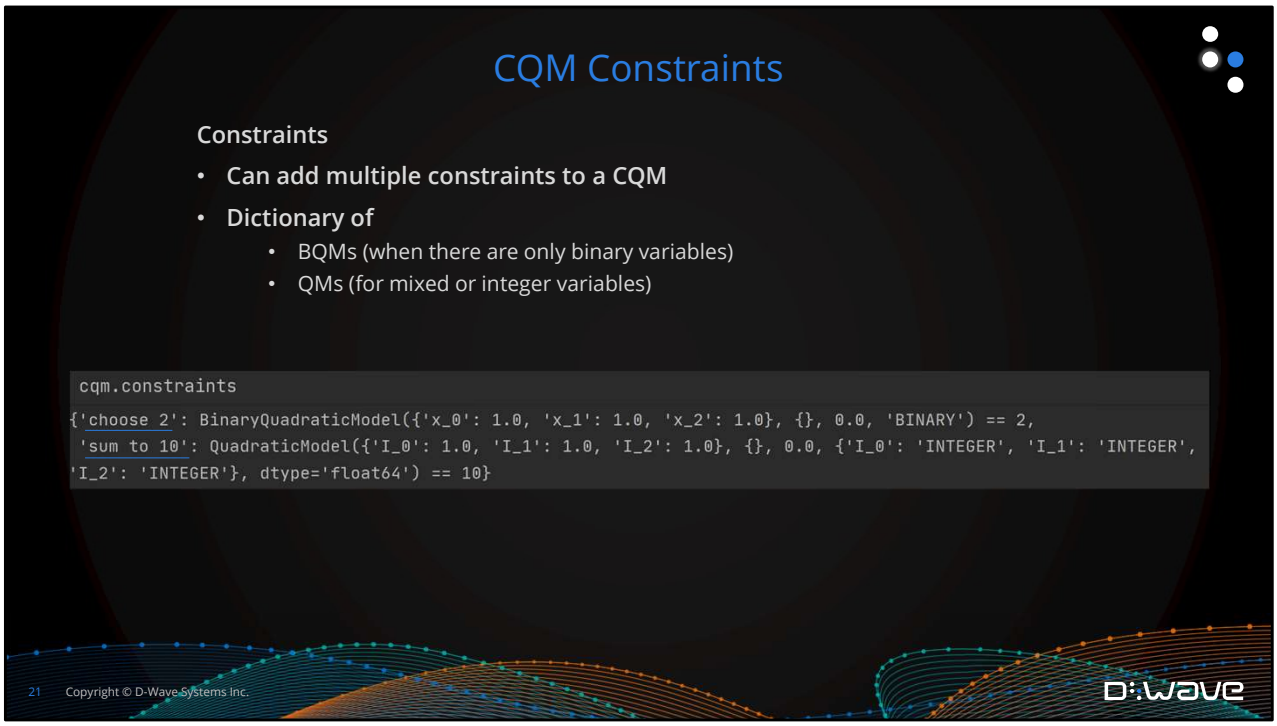


So far the examples we've looked at only show linear objectives.
The CQM solver supports quadratic objectives too.

Some common applications that use quadratic objectives are the quadratic muti-knapsack problem, graph partitioning and conformational search.



The CQM supports equality and inequality constraints natively.

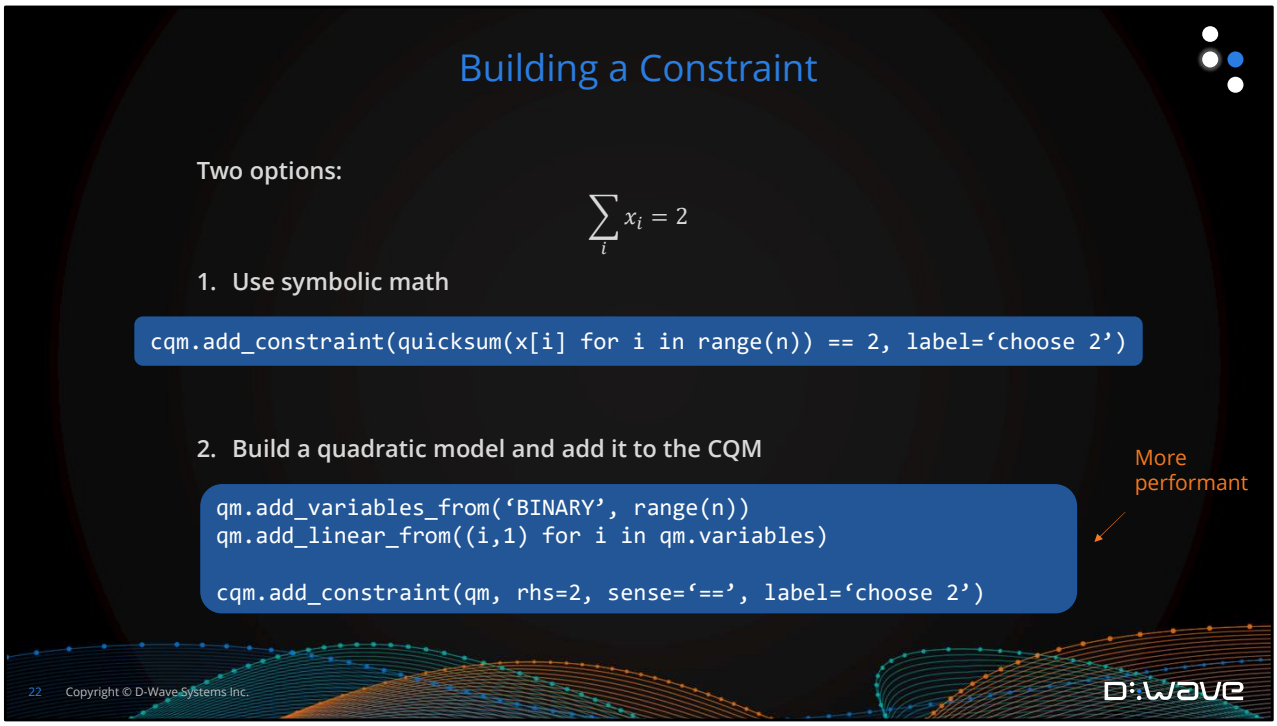


We can add up to 100 000 constraints to a CQM.

The constraints are stored as a dictionary of binary quadratic models and quadratic models depending on the variable type in the constraint. Constraints that only use binary variables are stored as BQMs and constraints with integer or mixed variables are stored as QMs.

Here's an example of a CQM with two constraints.
The first constraint is labelled 'choose 2' and it's a binary quadratic model with 3 binary variables.
The second constraint is labelled 'sum to 10' and it contains 3 integer variables.

Now let's take a look at how we can create and add constraints to CQMs.

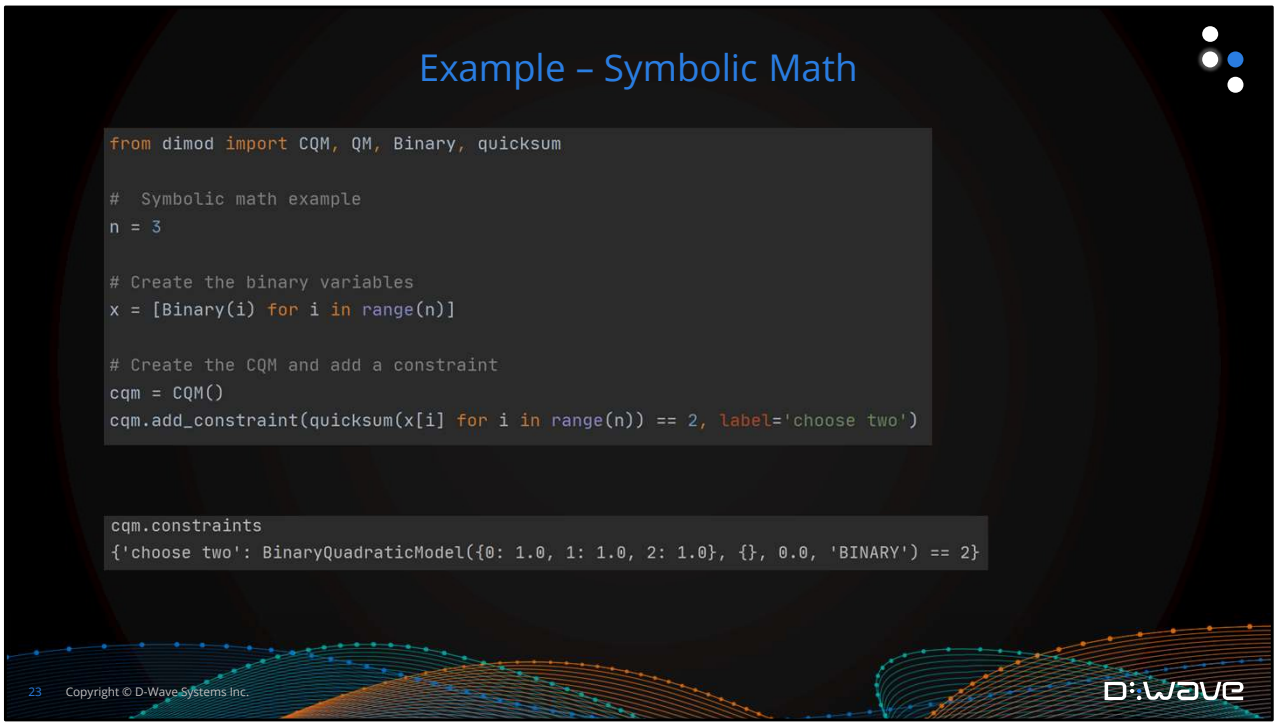


Again, we have two options.
Let's consider the equality constraint where we want to choose 2 binary variables.
Here, x represents a list of binary variables.

Using symbolic math we can employ the same technique as we did with the objective.
We can use the quicksum function to add up the binary variables in x and use the equality operator to define the equality constraint.
It's also helpful to include a label so you can tell the difference between the constraints in the model.

With the second option, we have to create a quadratic model, add the binary variables, and then set the linear weights even when the weights are one.
When adding a QM as a constraint with the add_constraint function, we need to specify additional input parameters.
The quadratic model we built only contains the left hand side of the equation shown.
In add_constraint, we need to specify the right hand side of the equation or the 2 in this case.

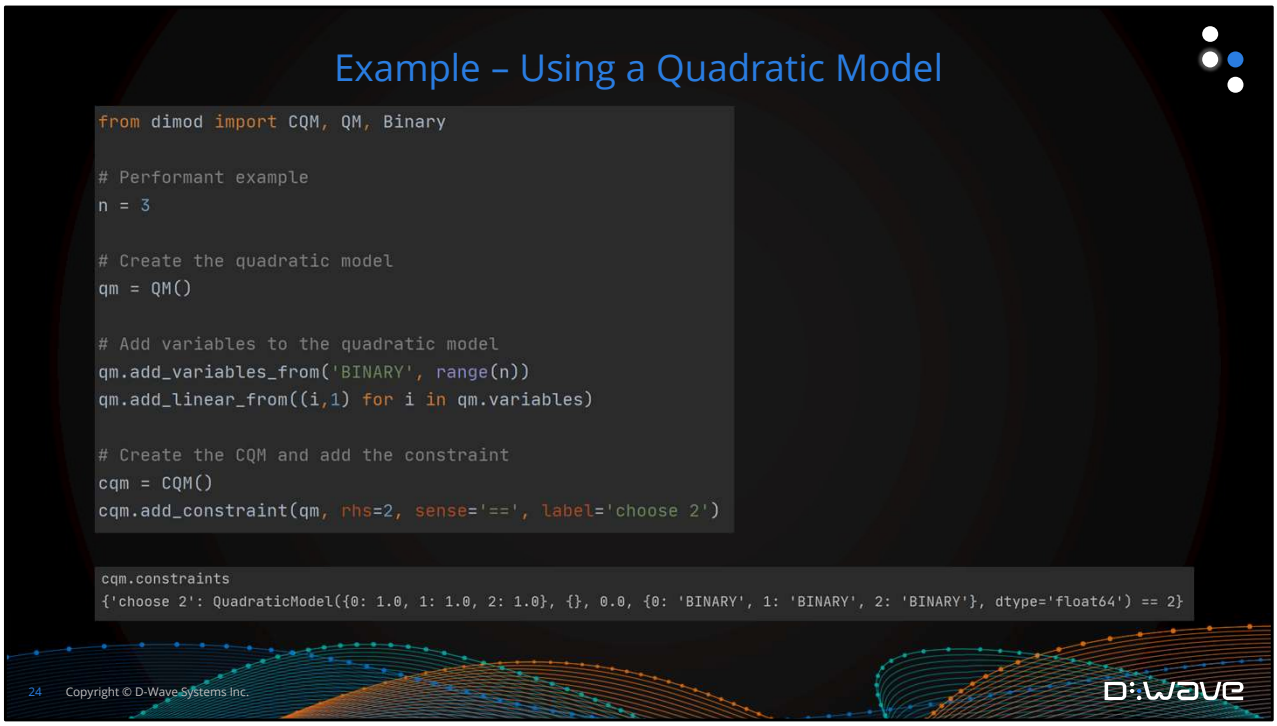
The ‘sense’ parameter is what we use to define the type of constraint. In this case we’re adding an equality constraint as you can see.



Let's take a look at how the full code snippet for how we would define a CQM with this equality constraint.

Using symbolic math our main steps are to create the list of binary variables and then construct the constraint by adding up all of the binary variables and setting the sum as equaly to 2.

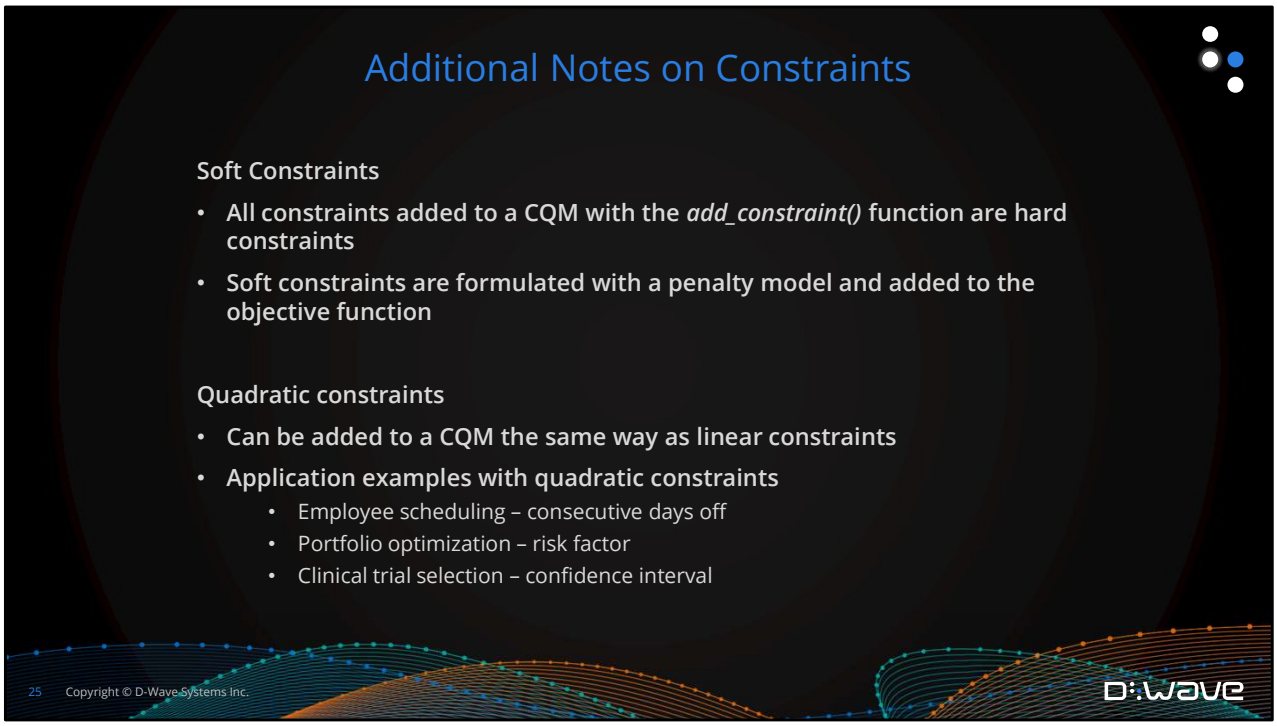
Notice that when we build the constraint this way, it is added to the CQM as a binary quadratic model.



Now let's look at how we would build this equality constraint using a Quadratic model.

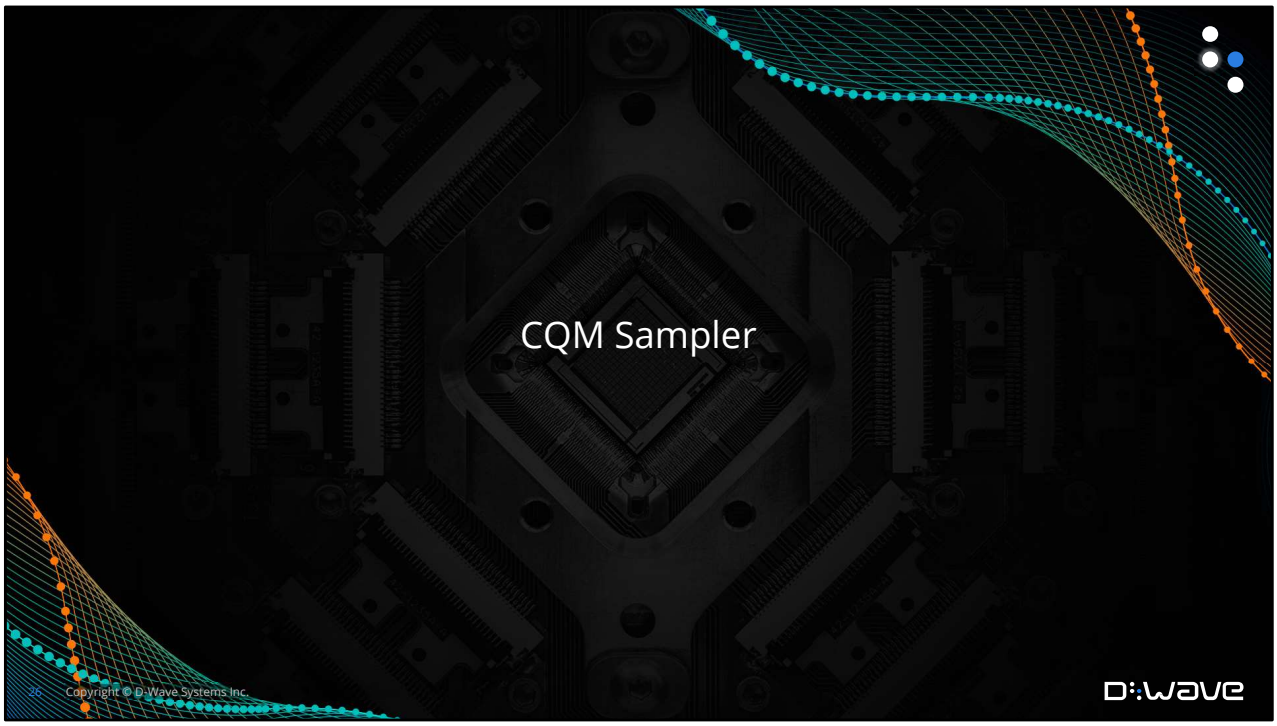
We already saw this code on the other slide, so I'm just going to point out that with this implementation the constraint is added as a quadratic model.
It represents the exact same equation as the other method, it's just expressed differently under the hood.

Again, the point is that both options are valid and produce the same model.
There are just a couple different ways to get there.

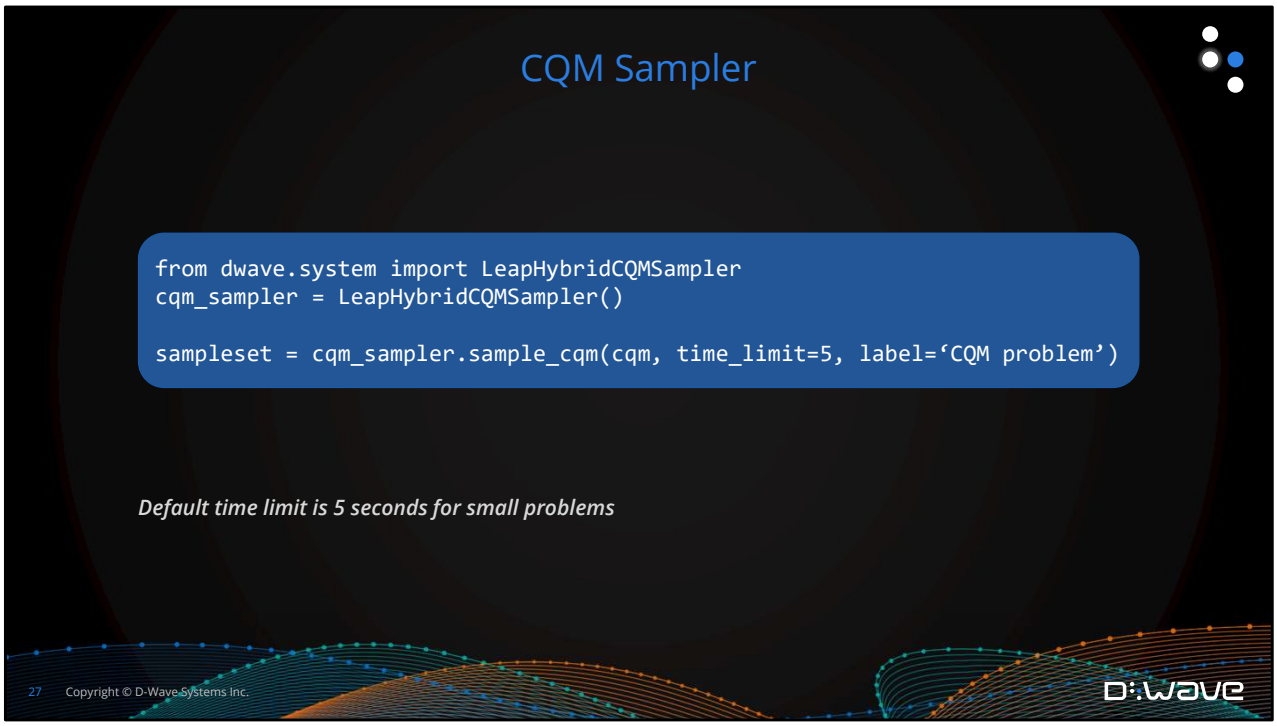


It's worth noting that all constraints that are added to a CQM with the add_constraint function are evaluated as hard constraints, so the CQM tunes the Lagrange parameter so that all constraints are ideally satisfied.
If you have soft constraints that you want to account for in your model, you can construct a penalty model, as you've seen in earlier modules in the course, and add it to the objective.
In that case you may need to manually weight the constraint relative to the objective functions.

The CQM solver also supports quadratic constraints.
The example we looked at only showed a linear constraint, but quadratic constraints can be easily added as well.
Some applications that use quadratic constraints are employee scheduling, portfolio optimization and clinical trial selection.

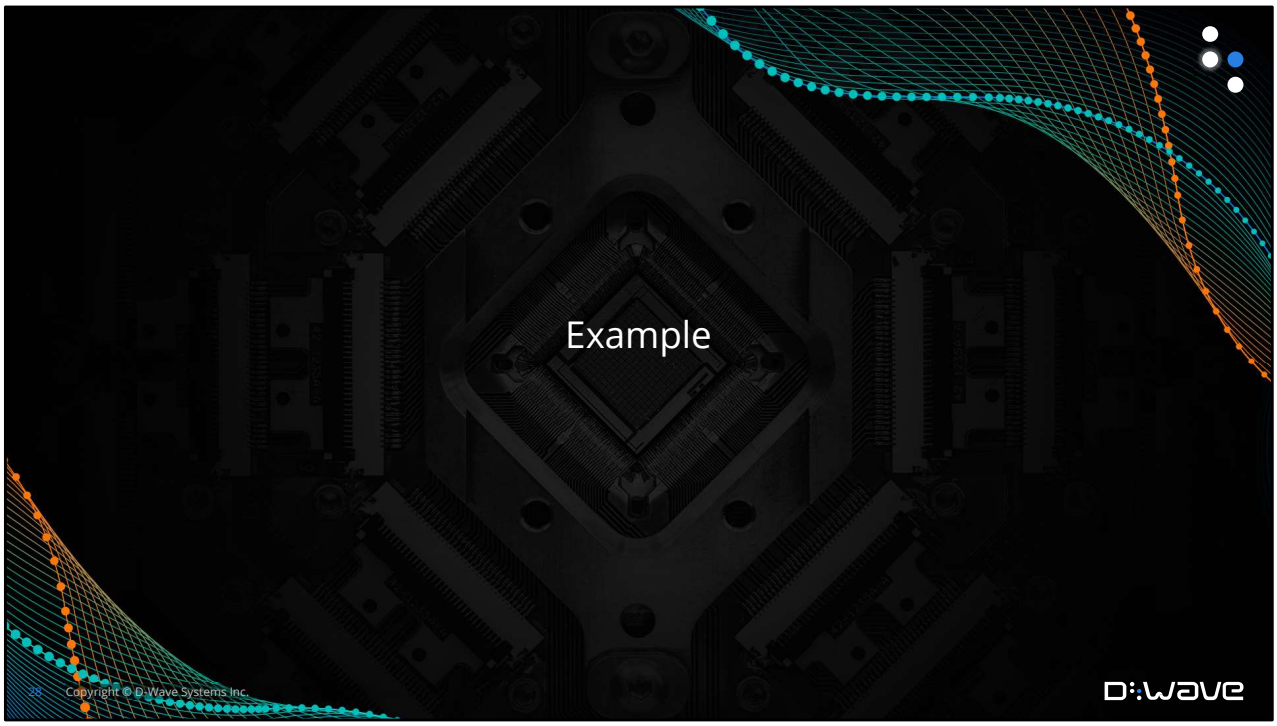


The last step in building a CQM program is to define the sampler and submit the CQM to it.

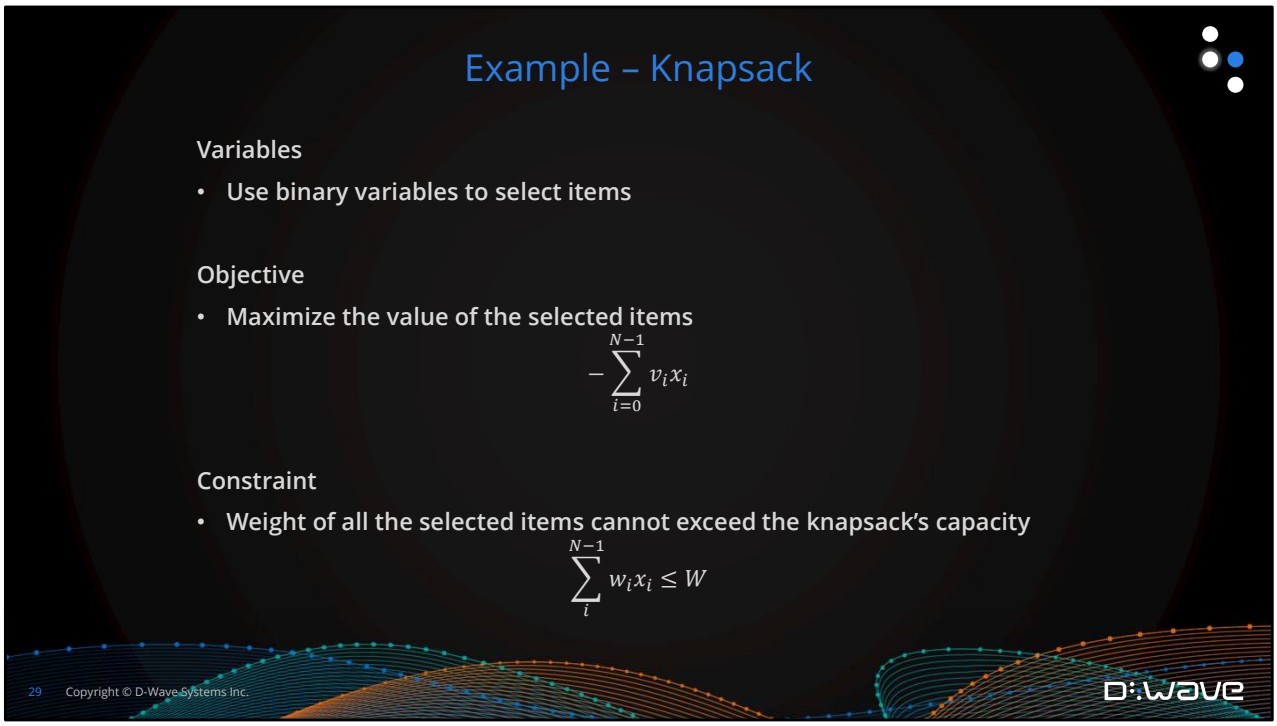


The CQM solver is accessed through the LeapHybridCQMSampler in the dwave.system package and the sample_cqm function is what we use to submit a CQM to the sampler.

Like the other hybrid solvers, the only tunable parameter available is the time limit.
By default the time limit is 5 seconds for small problems. That default scales with the problem size.



Now let’s put everything we’ve learned together by looking at an example.



Let's consider the knapsack problem.

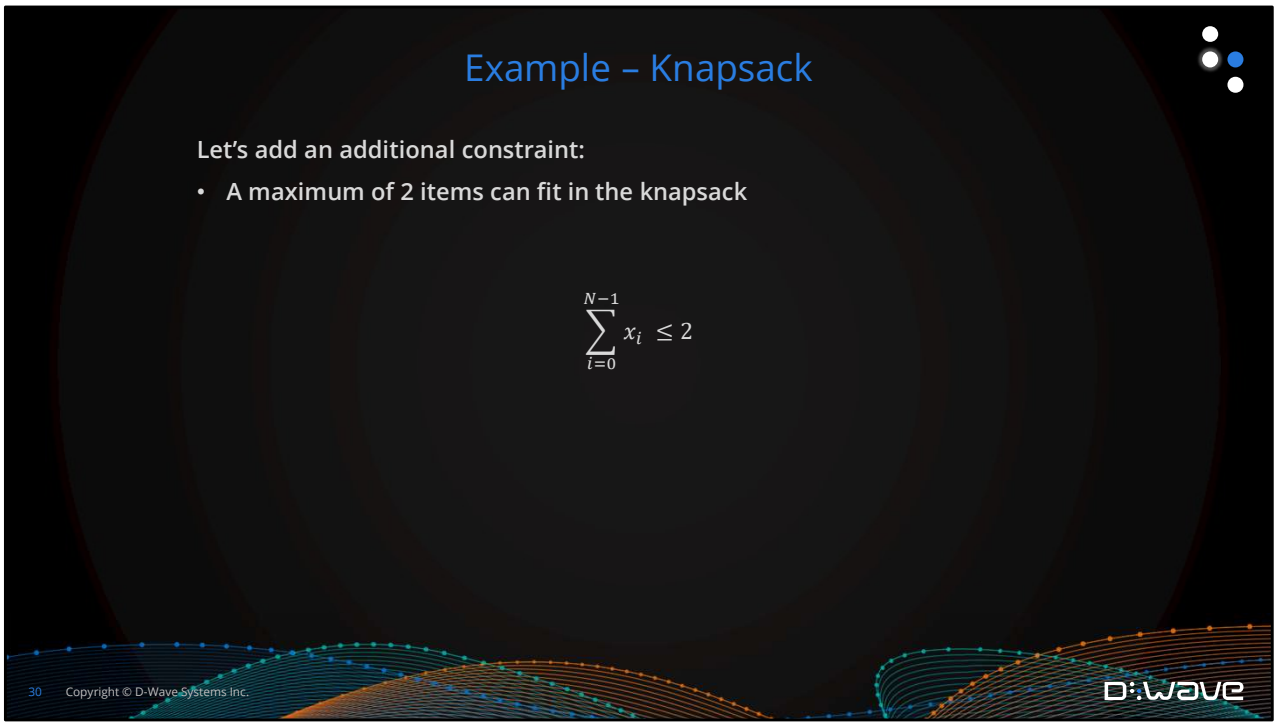
In this problem we want to select items to put into a knapsack, truck or other container so that we maximize the total value of the selected items.
We also need to make sure we don't exceed the weight capacity of the container.

To model this problem we'll use binary variables since we want to choose items. If a binary variable is 1 the item it represents goes into the knapsack and if it's 0 the item is left behind.

The objective is to maximize the total value of the items selected, so we're going to add up the value of each item multiplied by the item's binary variable.
In the objective expression here, v is the value of each item and x is the binary variable for the item.
We multiple this summation by -1 to convert it from a maximization to a minimization since the CQM solver seeks the minimum energy solution.

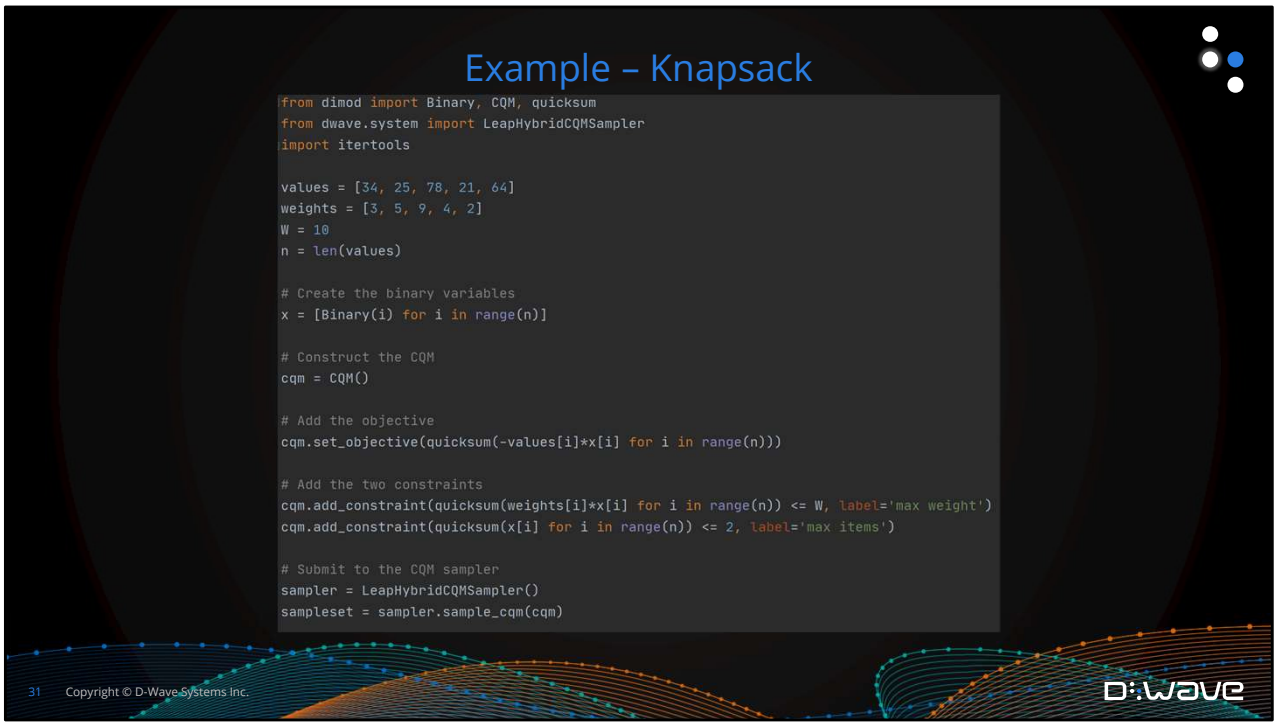
This problem also has an inequality constraint. The total weight of all of the chosen items cannot exceed the maximum weight capacity of the knapsack.

We express this with the inequality expression shown here, where the lowercase w in the summation represents the weight of each item and the upper case W is the knapsack's maximum weight.



For the sake of this example, let's add a second constraint. A maximum of 2 items can fit in the knapsack.

We'll express this with another inequality expression, where we sum up the binary variables and require that summation to be less than or equal to 2.



Here's how we would program this example.

For the sake of this example, we're going to look at a really small problem that only has 5 items.
The value and weight of each item is defined in the values and weights lists.
The maximum weight of the knapsack is set to 10.

We start by creating 5 binary variables, one for each item.

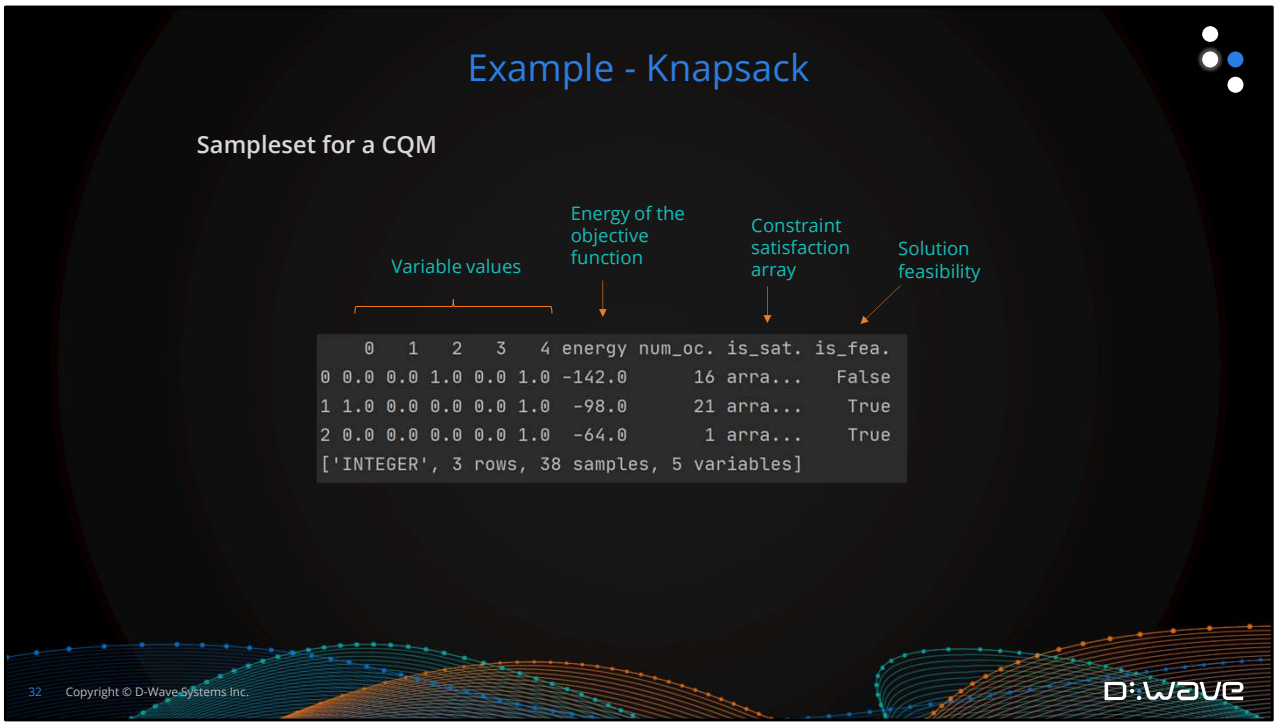
Then we create the CQM and set the objective to maximize the value of the selected items.
Notice the negative sign in front of the multiplication of the values times the binary variables.
Again, that makes the maximization objective compatible with the CQM solver that naturally minimizes.

Next we add the two inequality constraints.
First we sum up the weight of each item times the binary variable associated with the item and set the sum as less than or equal to the maximum weight of the knapsack

We call this constraint ‘max weight’.

Then we add the constraint that limits the number of items chosen to a maximum of 2.
That constraint is labelled ‘max items’

Lastly, we instantiate the CQM sampler and submit the CQM to it.

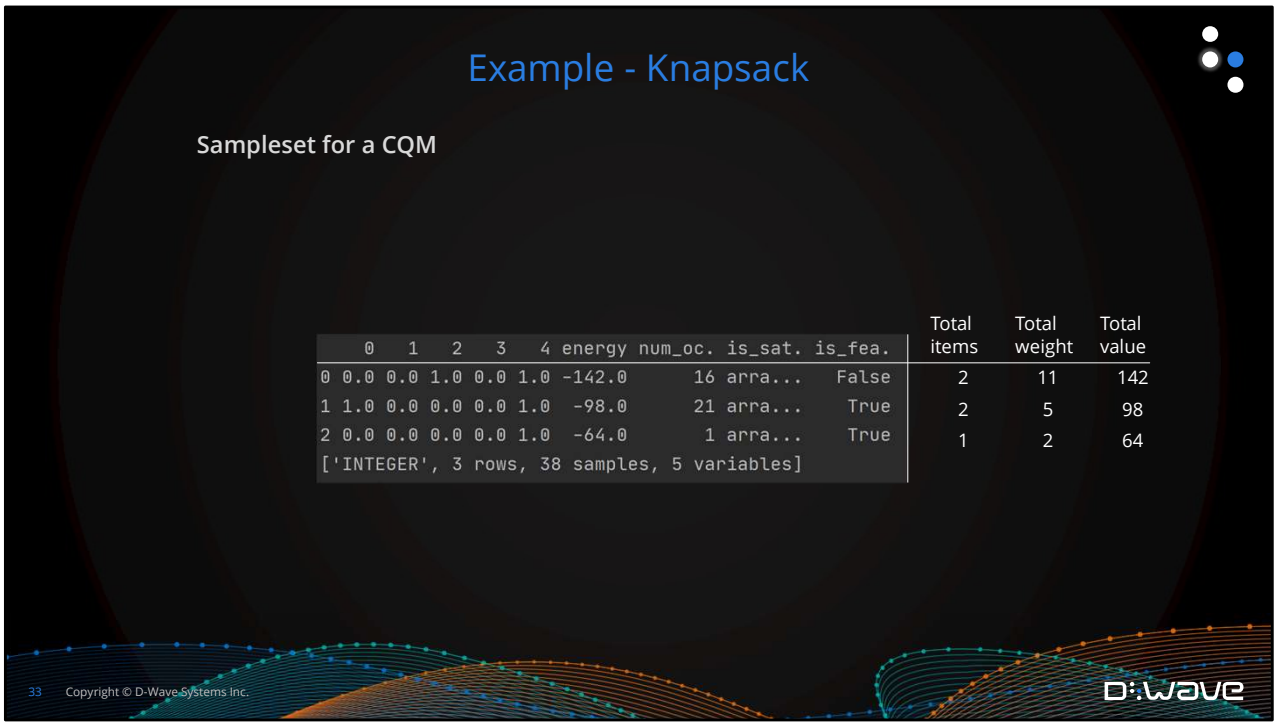


Now we're going to look at the solution.
The CQM sampler returns a sample set.

There are a few differences between this sampleset and those returned by the other solvers.
The first is that the energy is only calculated from the objective function, so the constraints don't impact the energy.
The second difference is that we have two additional columns in the sampleset returned by the CQM solver.
The is_satisfied column contains an array for each sample. The array contains the state of each constraint in the sample. In other words it tells us whether or not each constraint was satisfied.
The is_feasible column tells us if the sample is feasible. A sample is only feasible if all constraints are satisfied.

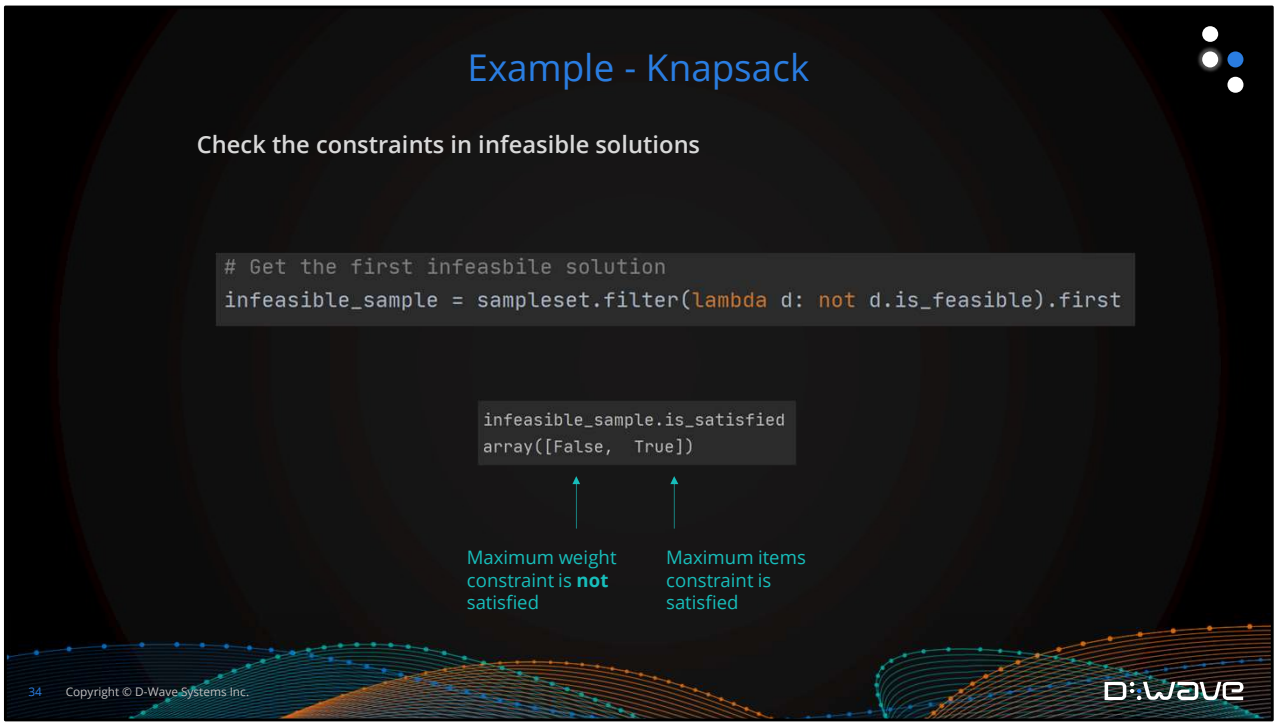
You'll notice that the CQM solver returns feasible solutions, as well as low energy infeasible solutions. We report individual constraint satisfaction so you can examine the low energy infeasible solutions.
Depending on the application, there may be cases where a low energy infeasible solution is worth considering depending on how many and which

constraints are violated.

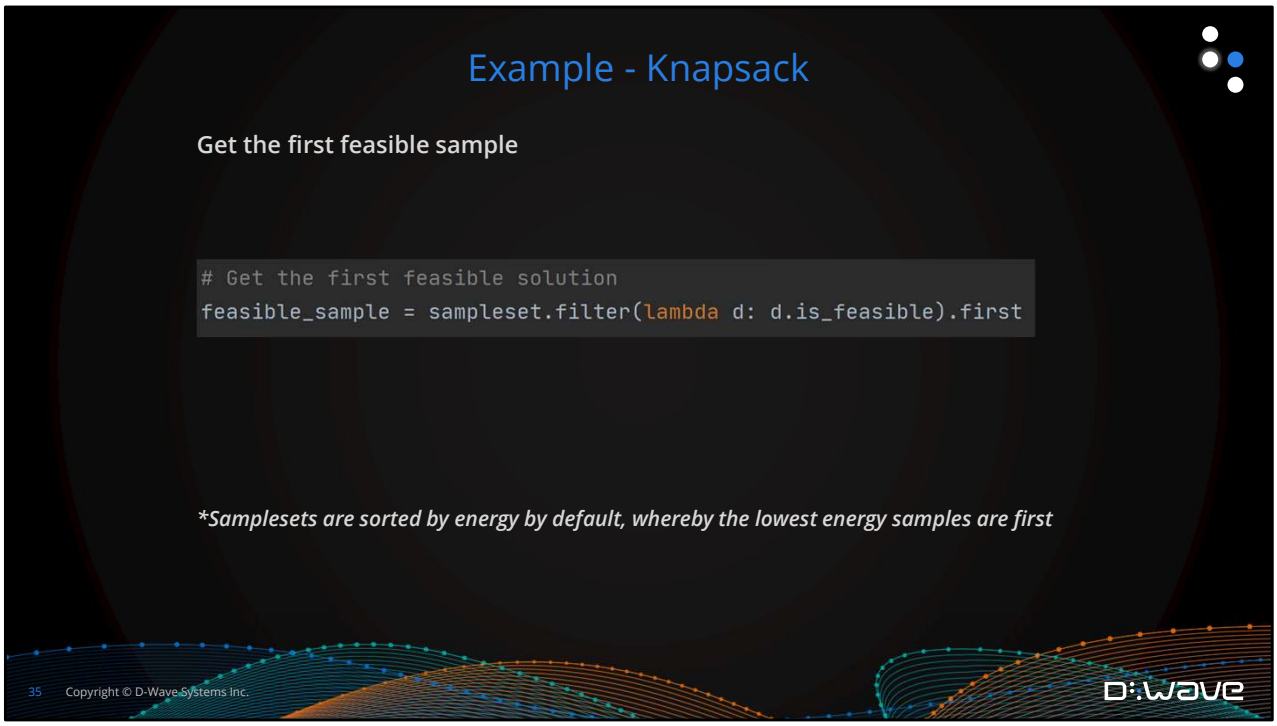


If we look at each sample in the samplset, we can see that all samples returned solutions that select 1 or 2 items, so the ‘max items’ constraint is satisfied in all.

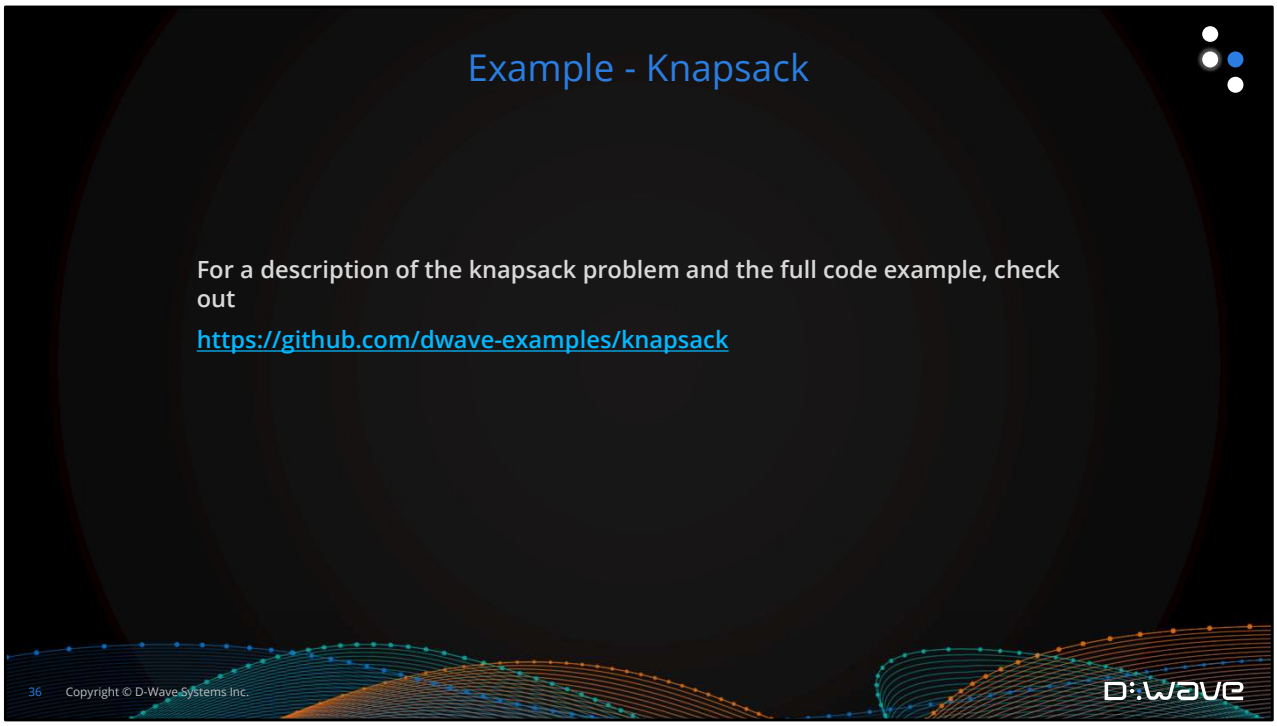
Remember that the maximum weight capacity for the knapsack is 10.
In this example, the best solutions that satisfy both constraints (the last two samples in the sampleset) have a relatively low total value. They values come in at 98 and 64 as is shown on the right.
If we can violate the maximum capacity constraint just a little, we can increase the total value to 142.
In some cases we might have that flexibility and it might be worth violating the constraint a little in order to increase the total value provided by the objective.



If this is the case and you want to examine the infeasible solutions, you can get the lowest energy infeasible solution using the filter function on the sampleset. Here we see what we discerned on the previous slide, that the maximum weight constraint is not satisfied, but the maximum items constraint is.



If we want to retrieve the lowest energy feasible solution, we can also do this with the filter function.
As you saw above, the first sample isn't necessarily the lowest energy feasible solution.
The samples in samplesets are sorted by energy by default.



For a full description of the knapsack problem and a code example with a larger problem, check out the knapsack example in our dwave-examples repo on github or in the collection of examples on Leap