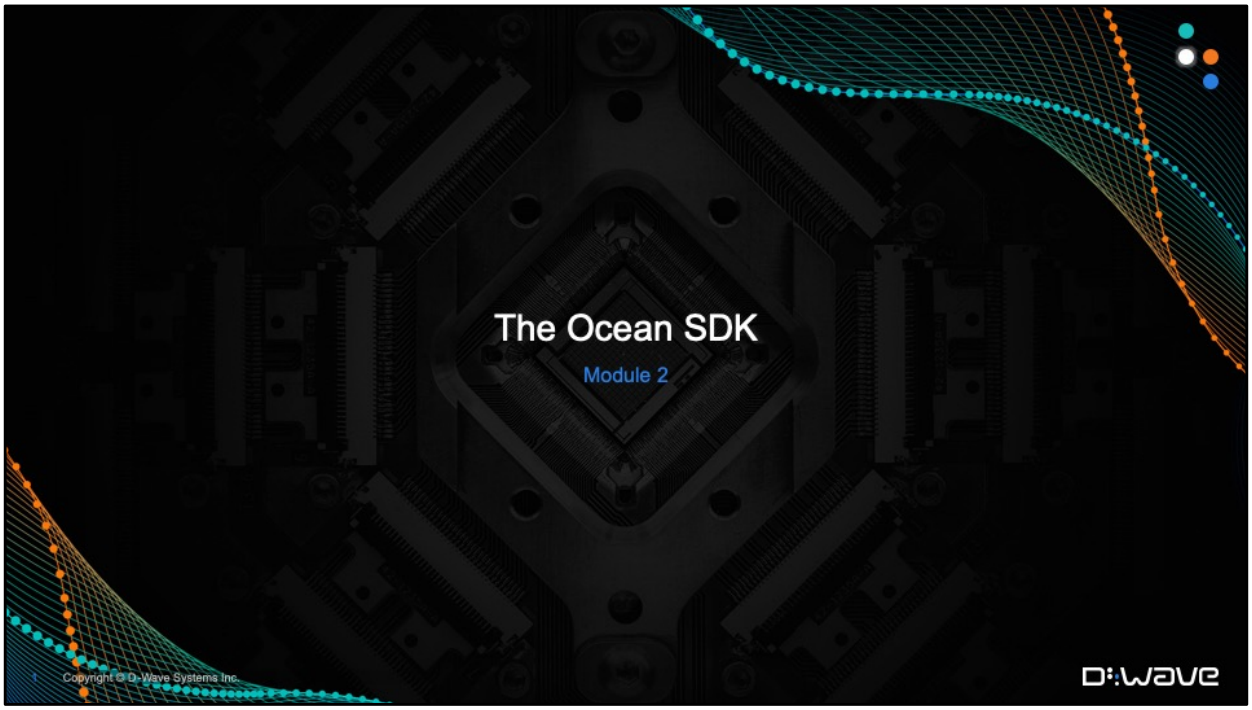
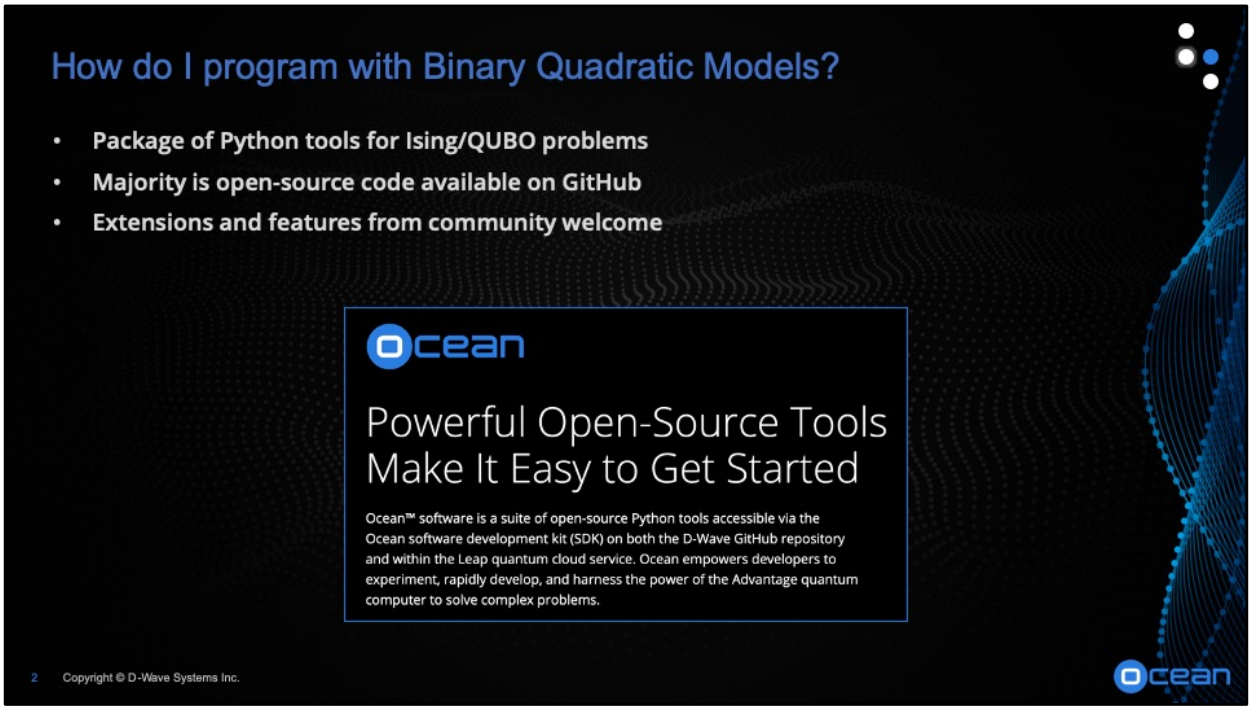
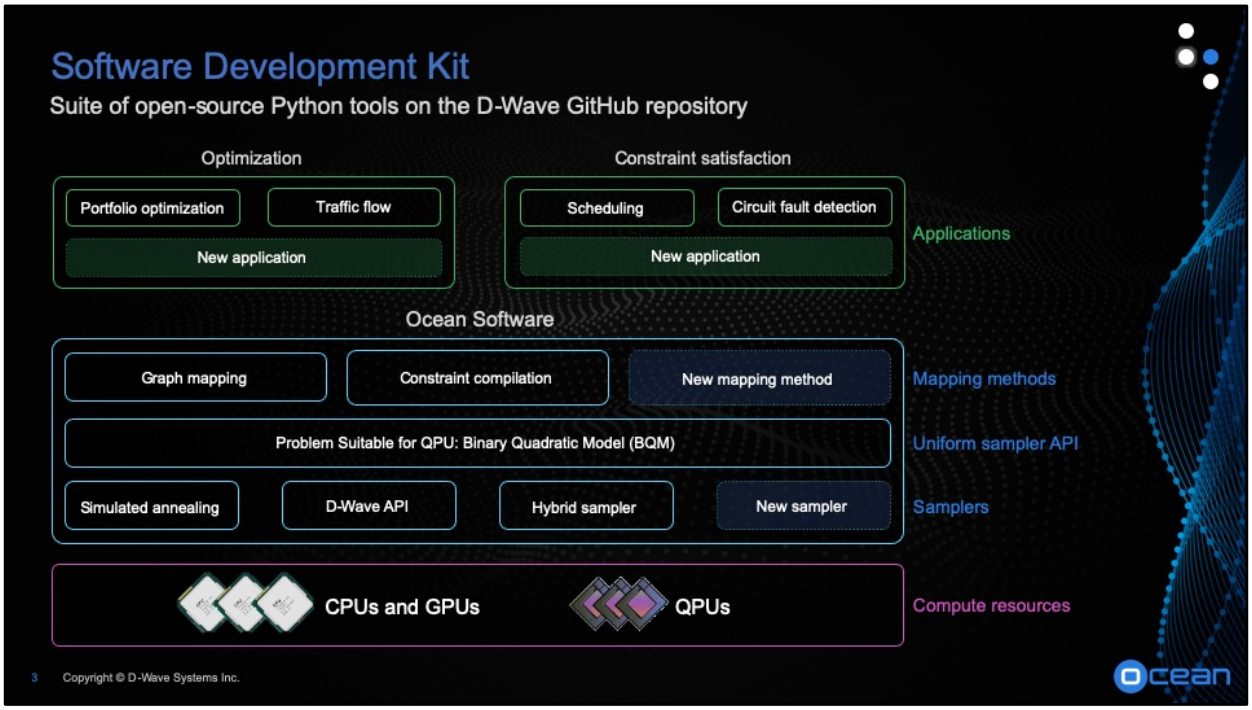


In this module we'll look at D-Wave's Ocean Software Development Kit and explore some of the built-in Ocean algorithm tools.



Now that you're familiar with QUBOs and binary quadratic models, we want to understand how to submit them to the quantum computer. To send problems over to the quantum processing unit or QPU we use our Ocean SDK. This is a package of Python tools, where almost everything is open-source. Our software team welcomes feature requests and code extensions from our community of users.



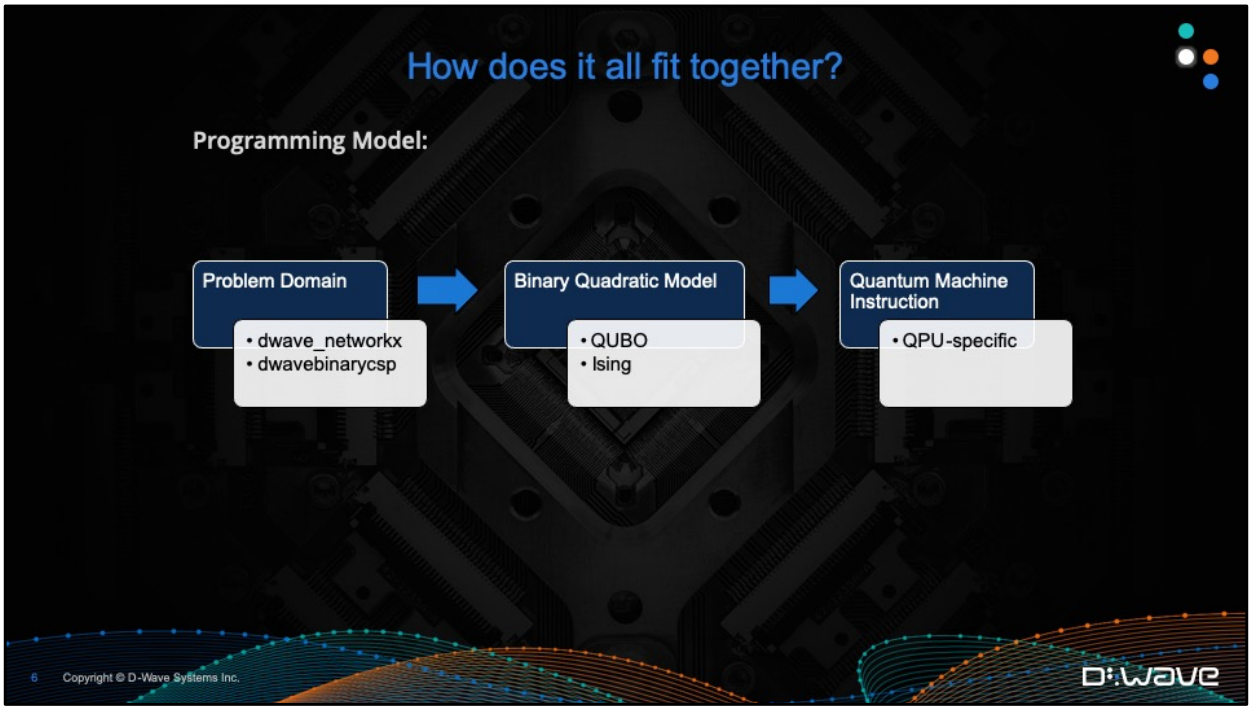
Here is an overview of the Ocean software stack. At the bottom are the compute resources that you can interact with. We provide access to our QPU, and you may also have additional resources like CPUs, GPUs, and even the laptop that you might be using to watch this lecture.

Above the compute resources we have our samplers and solvers. These are the different algorithms and tools available to solve your problem. Some of our samplers interact with quantum resources, like how the D-Wave API interacts with the QPU, some are classical samplers like simulated annealing running on classical hardware, and others are hybrid, like our Hybrid Solver Service. Each of these samplers is set up to run your problem in a binary quadratic model form, like QUBO.

For some special problem domains like graph theory or constraint satisfaction we have specialized packages available. The green layer of packages at the top are individual customer packages written for their specific applications. As you start working on applications you'll likely build your own packages and pipelines that sit on top of our Ocean Software Development Kit, or SDK.

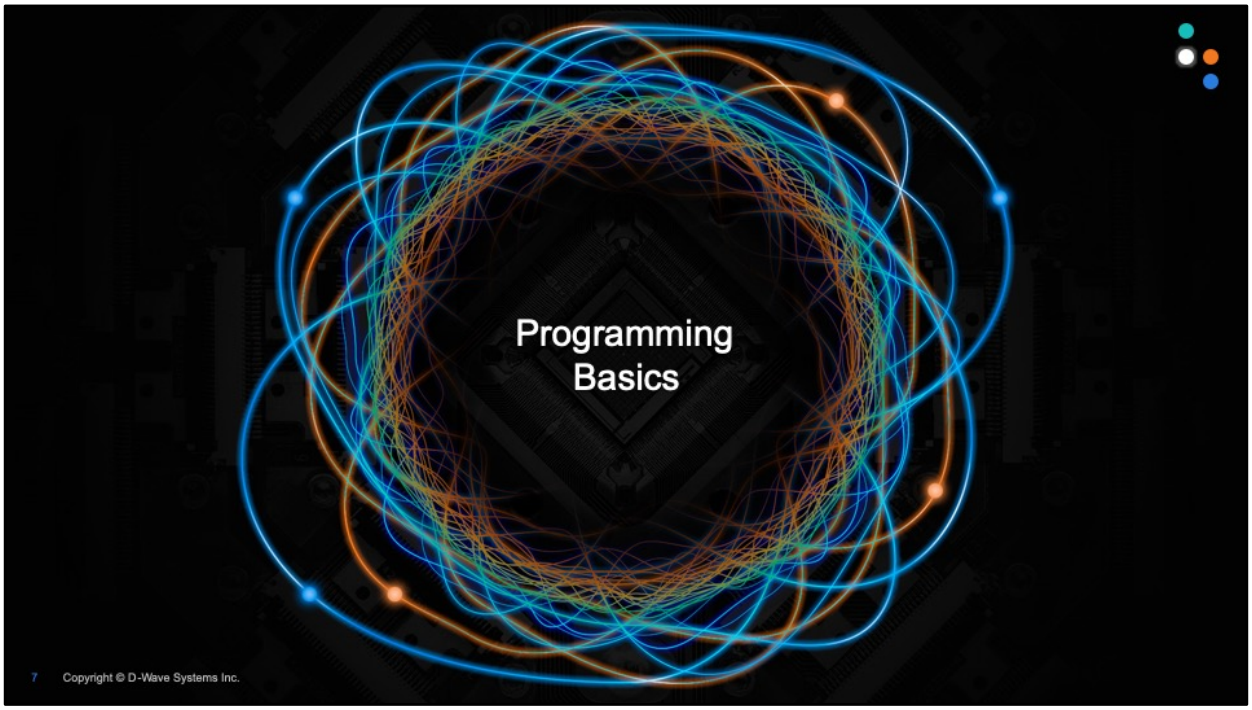
Within the Ocean SDK there are many different packages available. This is a snapshot of some of the available packages, each of which has a different function. For example, dwave-cloud-client is called every time you send a problem to our system in the cloud, and minorminer is used to figure out where and how to map your problem onto the physical chip. Each package has a different function, and they all work together to run your problem.

Our software team is actively building and improving the Ocean SDK, so you may see new packages appearing from time to time.

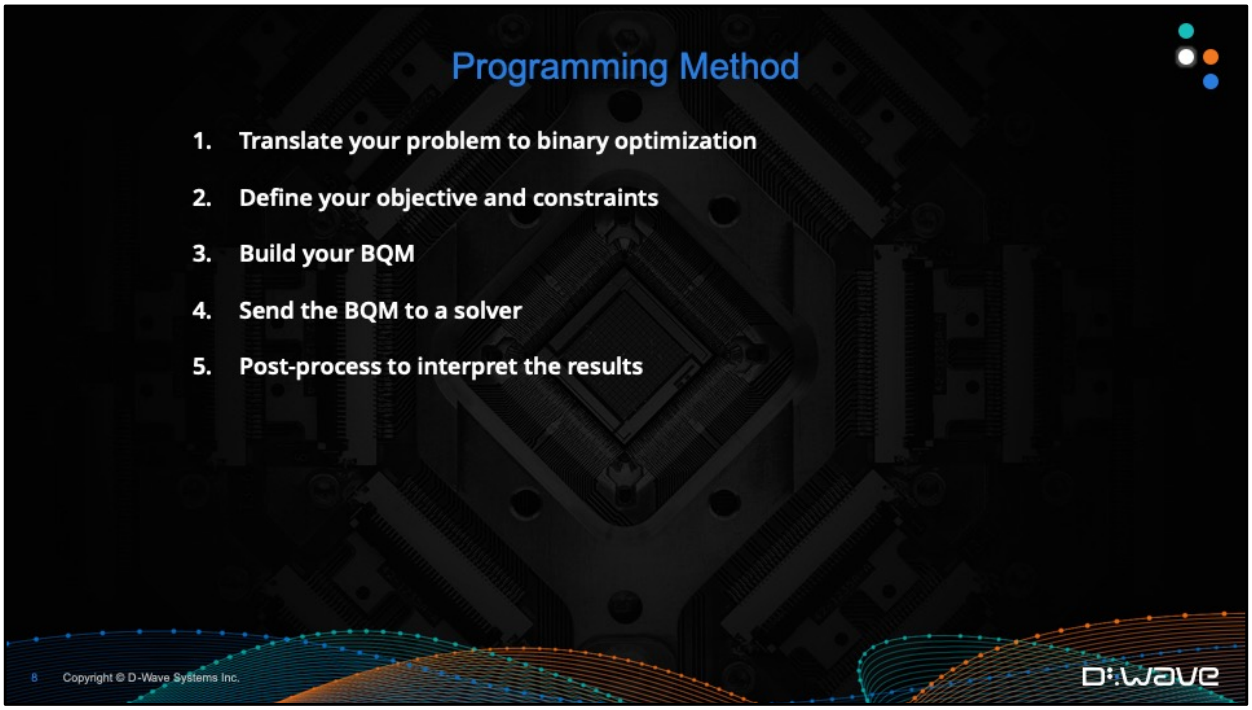


How do all of these pieces fit together to run your problem?

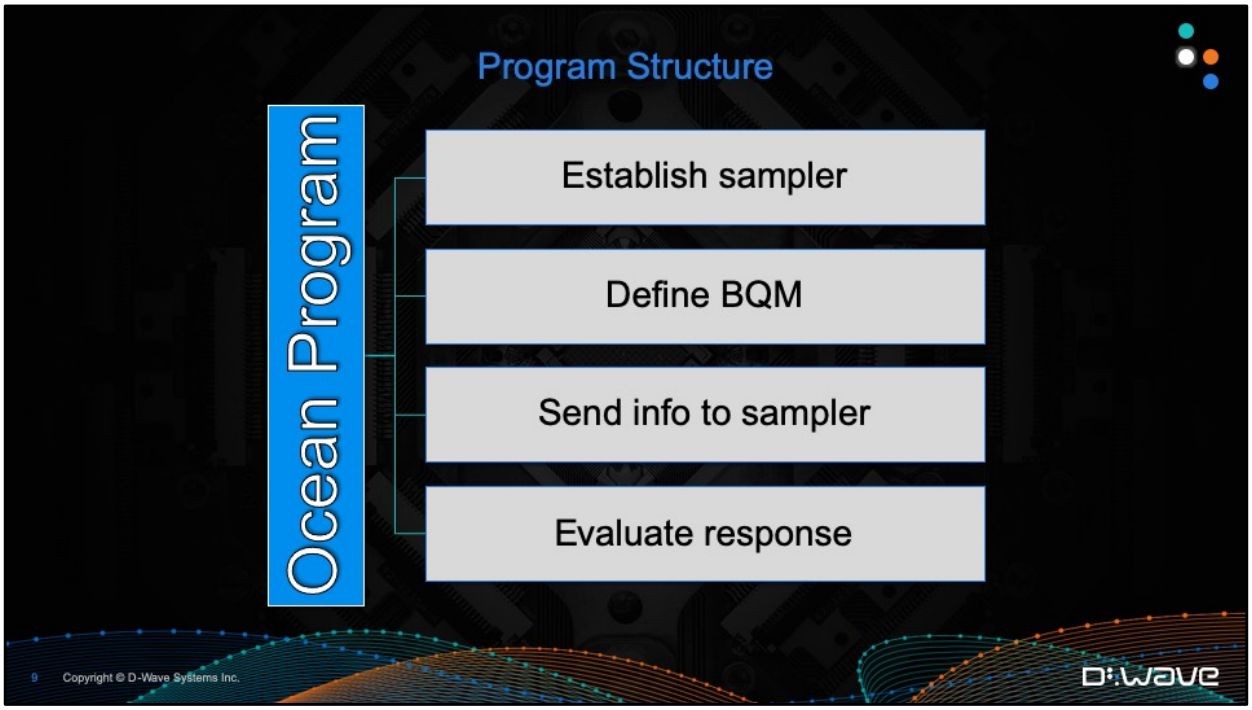
We start with a problem in the real world, in your problem domain. As I mentioned before, we have some specialized packages for certain domains, like dwave_networkx for graph theory problems. Your first step is to map your problem to a binary quadratic model like QUBO, as you saw in the last module. Once you have your QUBO, our software tools will construct a quantum machine instruction, or QMI. A QMI is the instruction set used to map your problem to the chip at the hardware level, and the Ocean SDK takes care of building this instruction set from your QUBO for you.



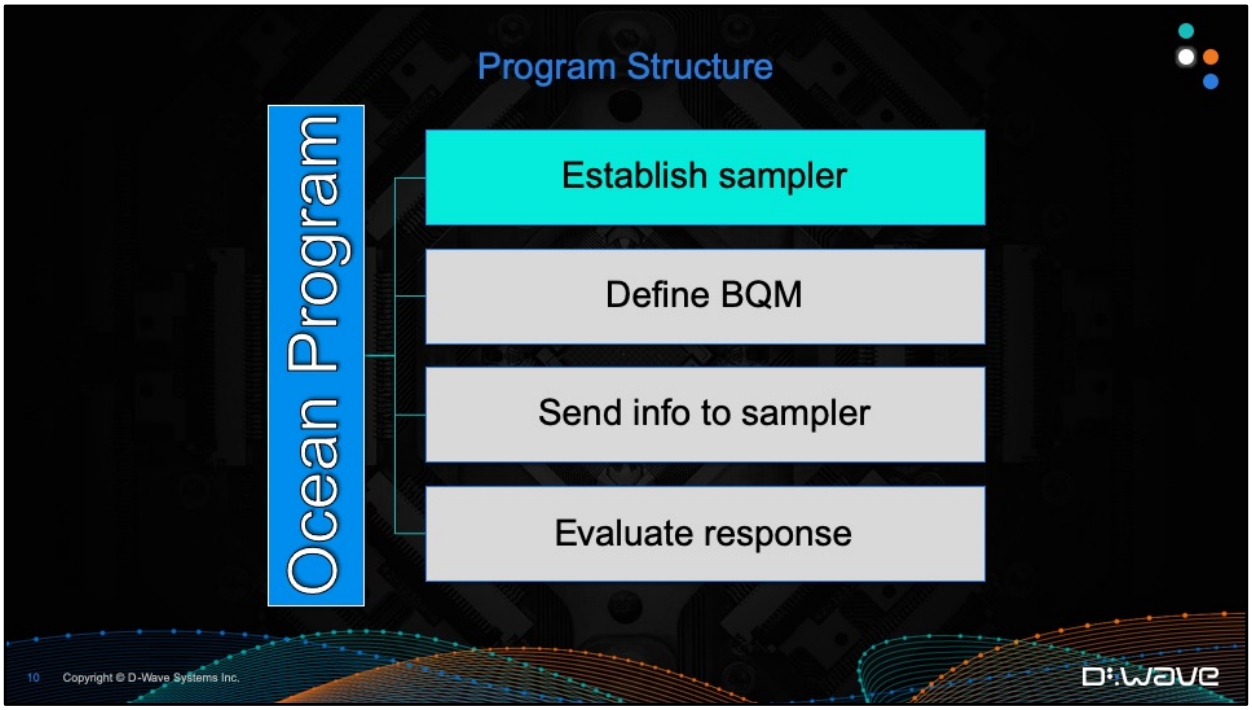
Let’s look at some programming basics for Ocean.



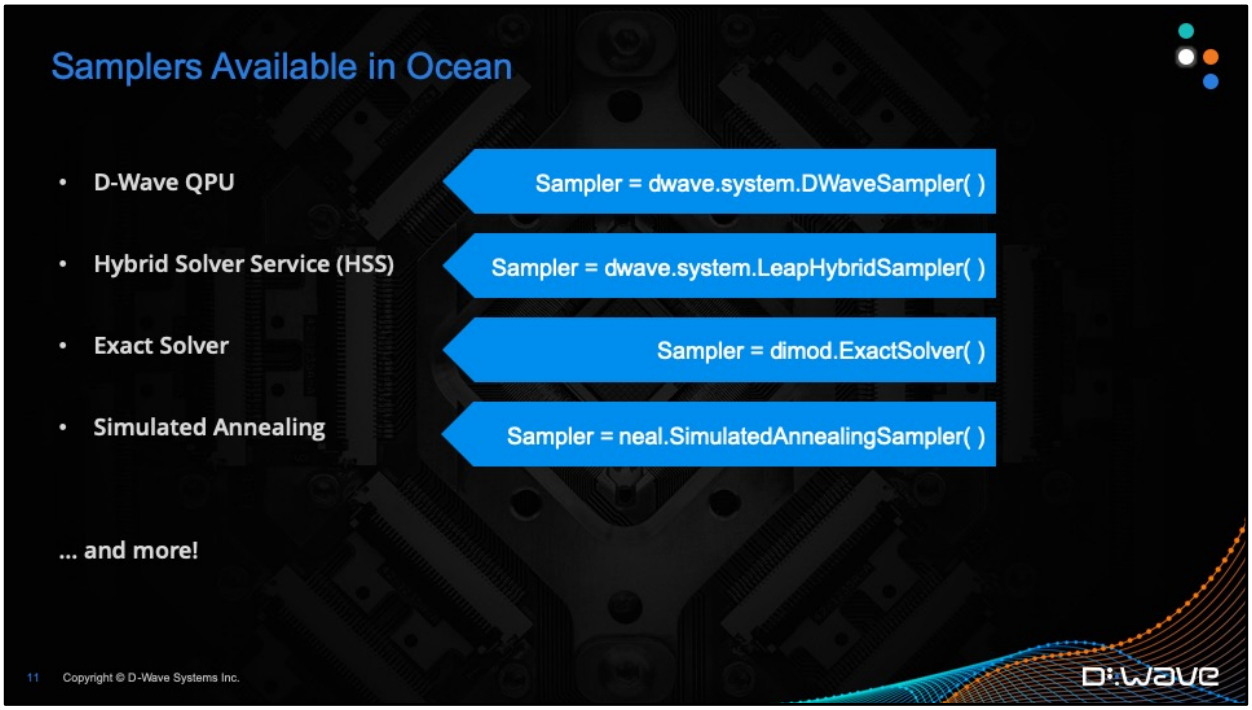
To run your problem using the Ocean SDK, you'll first need to translate the problem into a binary optimization problem and convert it to a binary quadratic model with an objective and/or constraints. Once you've done that, we build the BQM with Ocean. This BQM is the input for our software tools and is what we send to the assigned solver or sampler. Results are returned and you'll most likely want to do some post-processing to interpret the results and translate them back to your problem domain to examine them.



A typical Ocean program has four main components, and we'll go through each of these one by one.

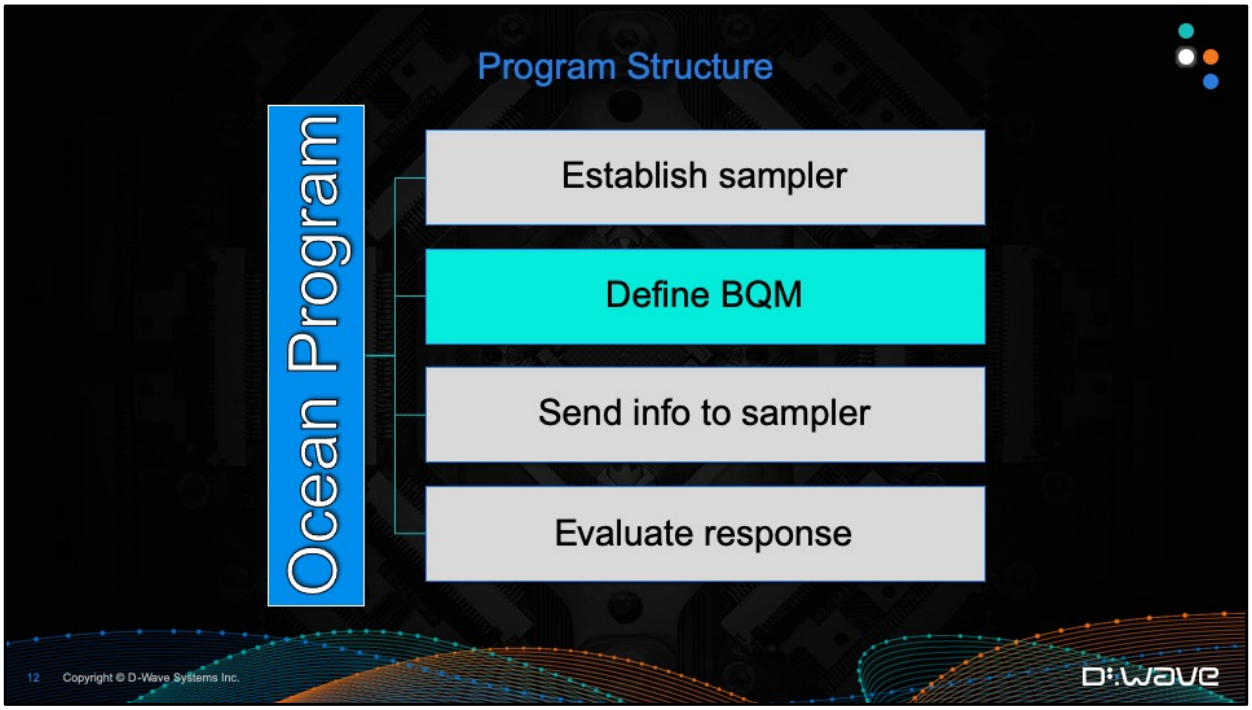


First, you need to define your sampler. Which tool do you want to use to solve your problem? The QPU? A hybrid solver? Our first step is to define that in our program.

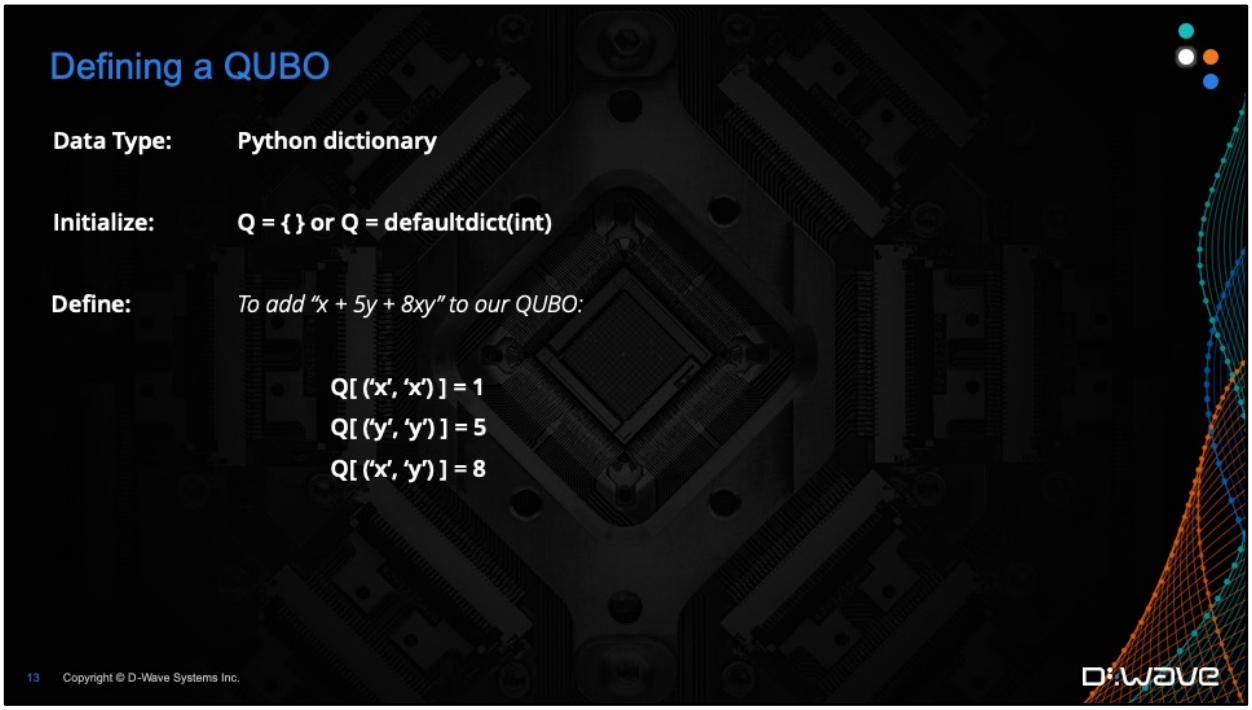


The Ocean SDK has a variety of samplers available for you to use, and here I've shown four popular ones. On the left you see the name of the sampler, and on the right is how you might define it in your python program. Note that you'll need to import the package where that sampler lives, so you might want to refer back to our Python prep module to recall how to do that.

For example, to use the D-Wave QPU we need to import the dwave system package, and then our sampler definition starts with dwave.system to tell our program which package to look in. We can follow that with the name of the sampler, DWaveSampler.



Next we need to define our problem as a binary quadratic model, in either QUBO or Ising form. We haven't talked too much about Ising form, but they are a type of binary quadratic model that is equivalent to QUBO – we'll just focus on QUBO for now.



To define a QUBO as a matrix we define a python dictionary in our program, which we usually call Q.

In this example, we are using the row and column names or indices from our QUBO matrix as the keys in our dictionary. Keys are the items in parenthesis, like (x, x) or (x, y), which are indicating which coefficient we are assigning in our equation. Keys are usually strings or integers. Here we see linear coefficients being assigned to variables x and y, as well as a quadratic coefficient on the term x times y. The value in our dictionary, like the numbers 1, 5, and 8, are the coefficients in our QUBO equation.

Defining a BQM

Initialize: **bqm = BinaryQuadraticModel('BINARY')**

To add "x + 5y + 8xy" to our QUBO:

Matrix:

```
bqm.add_variables_from( [ ('x', 1), ('y', 5) ] )
bqm.add_interactions_from( [ ('x', 'y', 8) ] )
```

Symbolic:

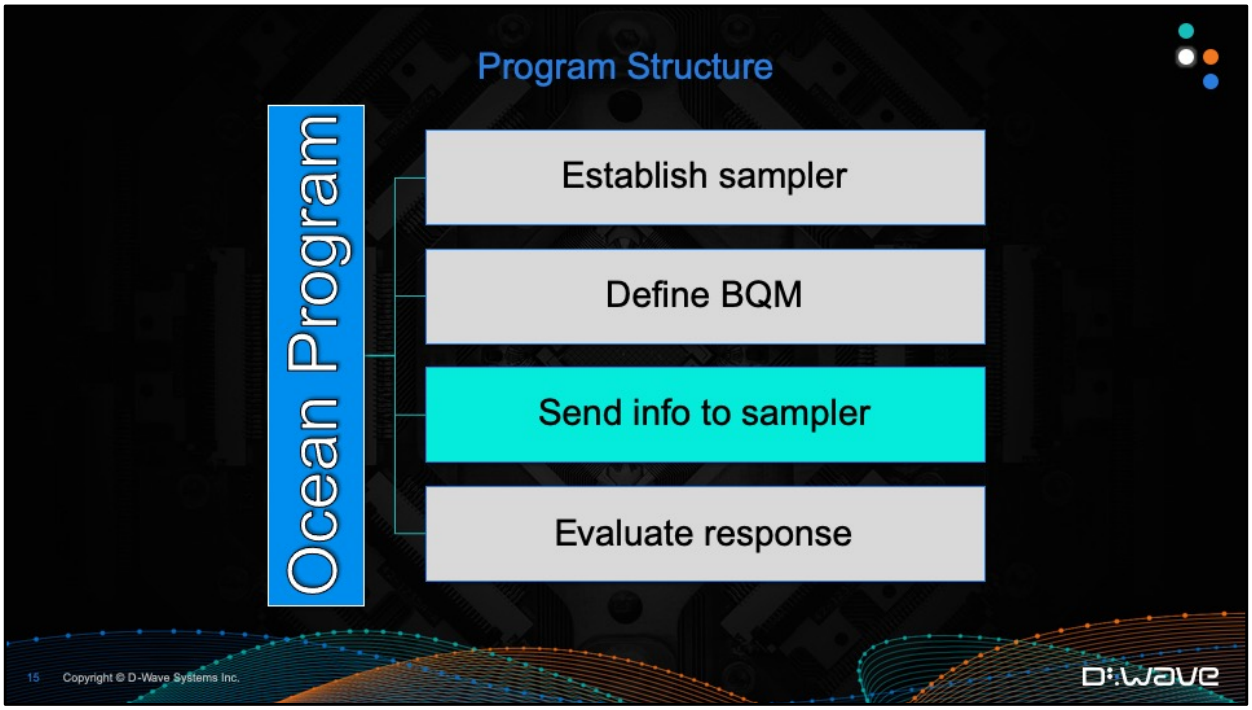
```
x = dimod.Binary('x')
y = dimod.Binary('y')
bqm = x + 5*y + 8*x*y
```

14 Copyright © D-Wave Systems Inc.

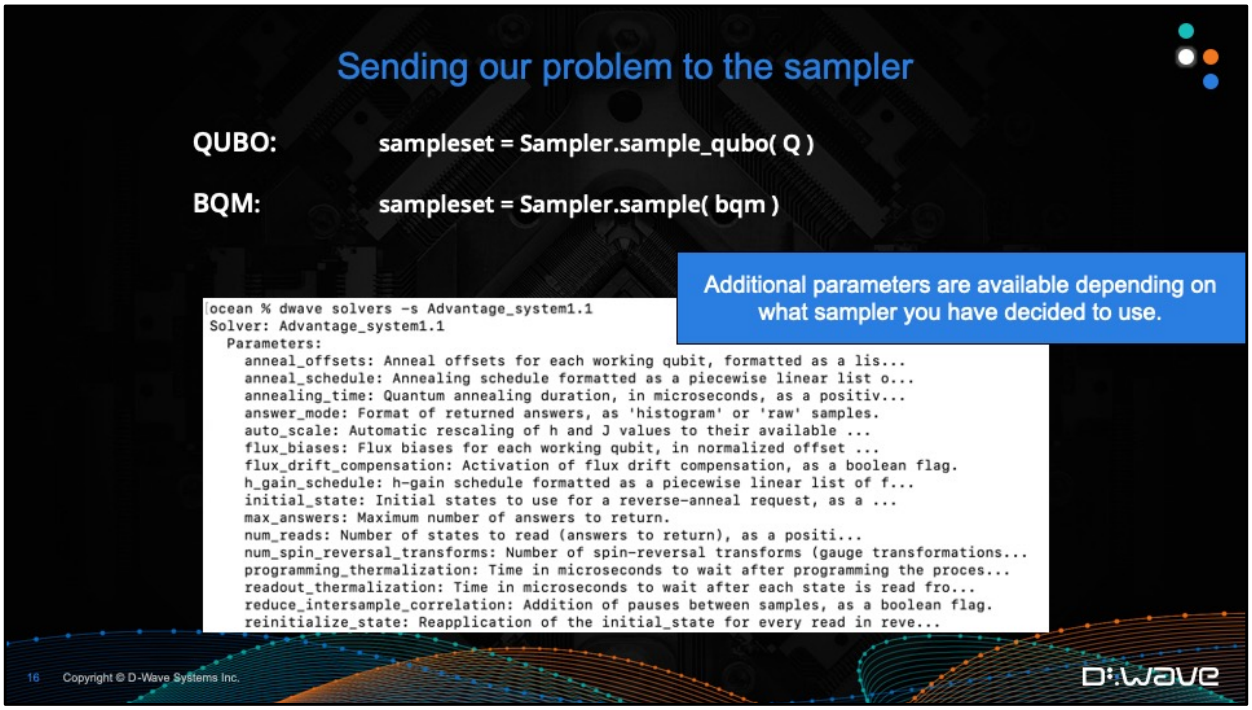
To define our QUBO as a binary quadratic model with Ocean, we have a few different options.

We can build up our equation in a similar way as we saw on the last slide, by assigning linear coefficients and quadratic coefficients directly using the functions add_variables_from and add_interactions_from, respectively.

Another option is to give the equation directly to the binary quadratic model. This will build the same data structure as if we had directly assigned the linear and quadratic coefficients individually. All of these methods produce the same BQM for the equation x + 5y + 8xy.

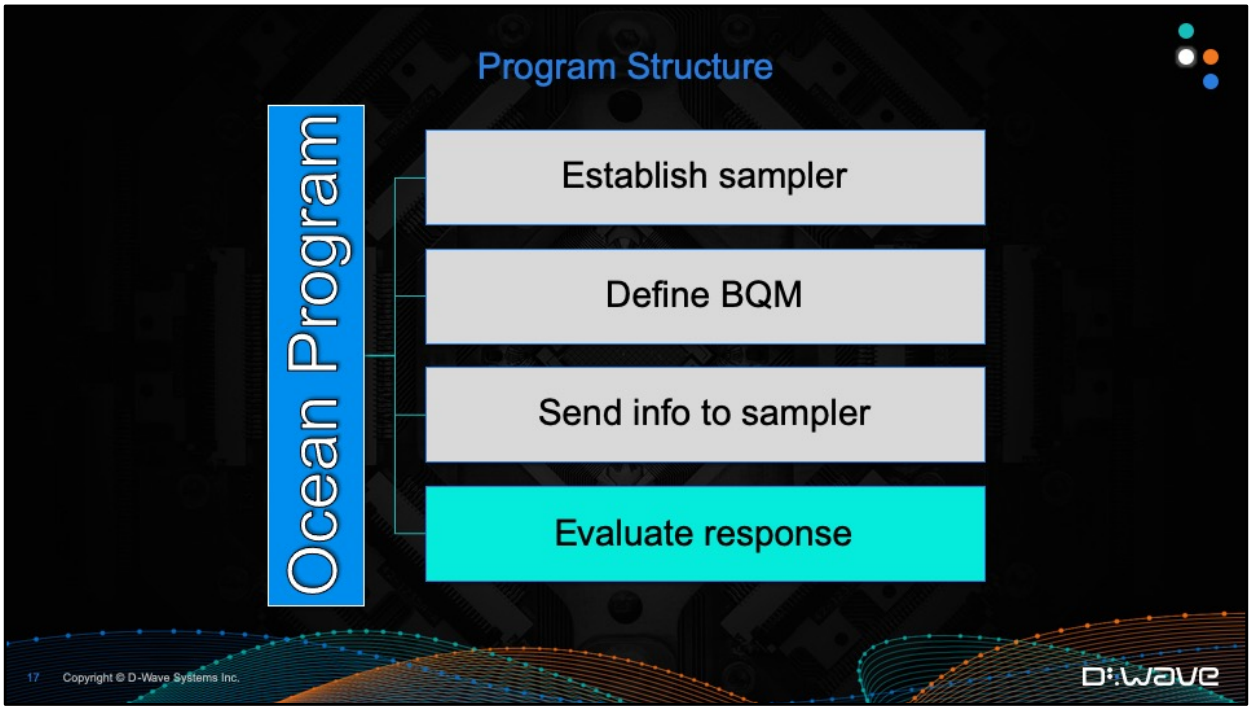


Next we need to send our problem over the sampler we set up.

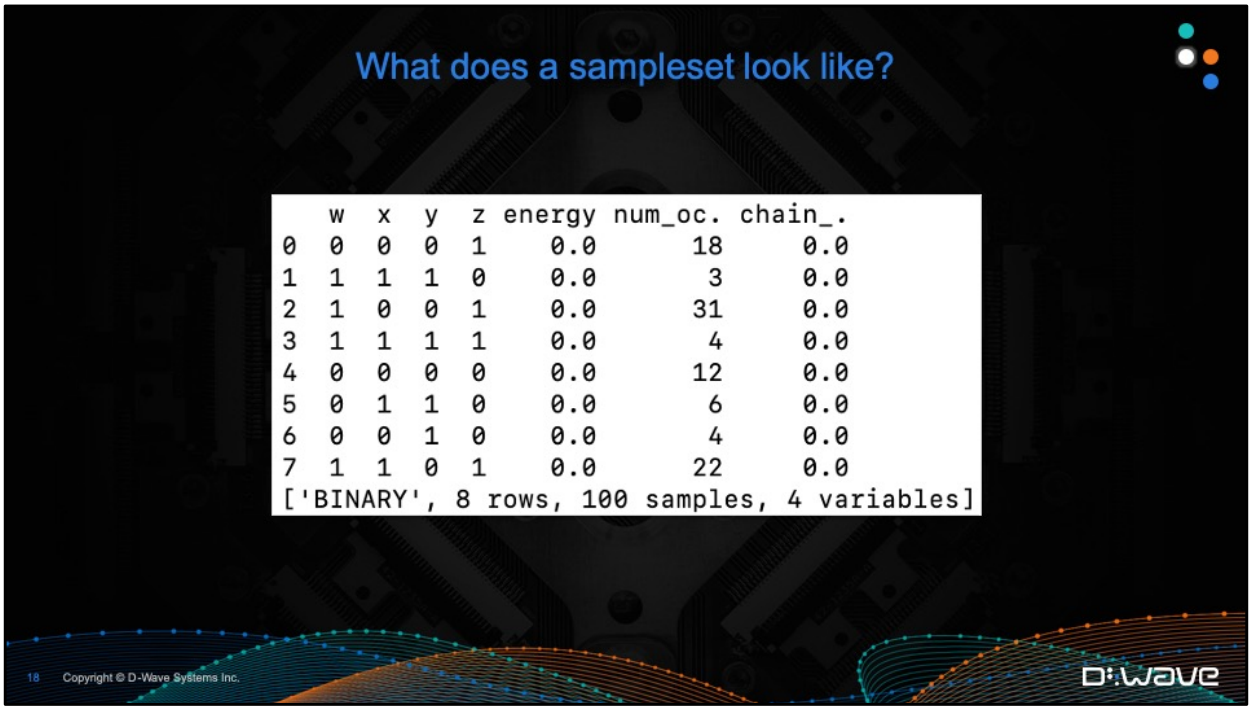


We'll do this using one of the commands shown to sample the model that we've built.

Note that each sampler has many parameters that you can set to tune your results so it's important to take a look at them, as I've shown. The QPU is probabilistic – we are not guaranteed to get the optimal solution when we run a problem on it. For this reason we typically run our problem hundreds or thousands of times, and using the number of reads parameter we can control this.



Last but not least, we look at the results that are returned and decide if we need to make any adjustments.



The object returned from a sample command is called a sampleset. If you print it out, you'll see lots of information provided to you.

First, on the left-hand side we see the numbers 0 through 7 in this example – these are indexing 8 different samples that were returned from my system call. In this problem I ran it 100 times on the QPU. At the top we see the variable names w, x, y, and z. These are the names of my variables in my QUBO or BQM. In each row I can see the value that each of these binary variables took. In the sample at index 0 the system returned w=0, x=0, y=0, and z=1.

The next column, energy, tells me the system energy that corresponds to that answer. Remember that smaller energy is generally better since we are minimizing our QUBO, and your samples will always be sorted from smallest energy to largest.

The next column tells me how many times that sample appeared. This first answer appeared 18 times out of the 100 runs.

The last column tells me what my chain break fraction was. We'll talk more about this in a later module.

How do I get my samples?

Using sampleset.data:

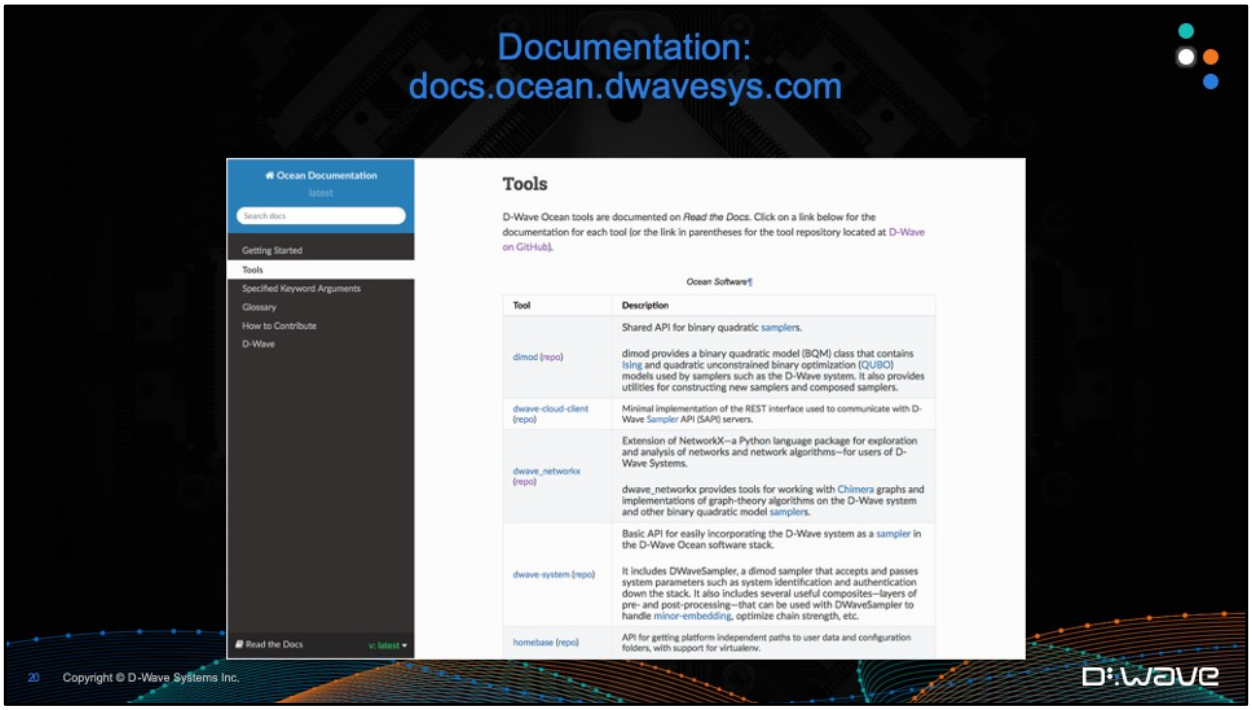
```
>>> for sample, energy in sampleset.data(['sample', 'energy']):
...     print(sample, energy)
...
{'w': 0, 'x': 0, 'y': 1, 'z': 0} 0.0
{'w': 1, 'x': 1, 'y': 0, 'z': 1} 0.0
{'w': 0, 'x': 0, 'y': 0, 'z': 1} 0.0
{'w': 0, 'x': 1, 'y': 1, 'z': 0} 0.0
{'w': 1, 'x': 1, 'y': 1, 'z': 1} 0.0
{'w': 0, 'x': 0, 'y': 0, 'z': 0} 0.0
{'w': 1, 'x': 0, 'y': 0, 'z': 1} 0.0
{'w': 1, 'x': 1, 'y': 1, 'z': 0} 0.0
```

Using sampleset.record:

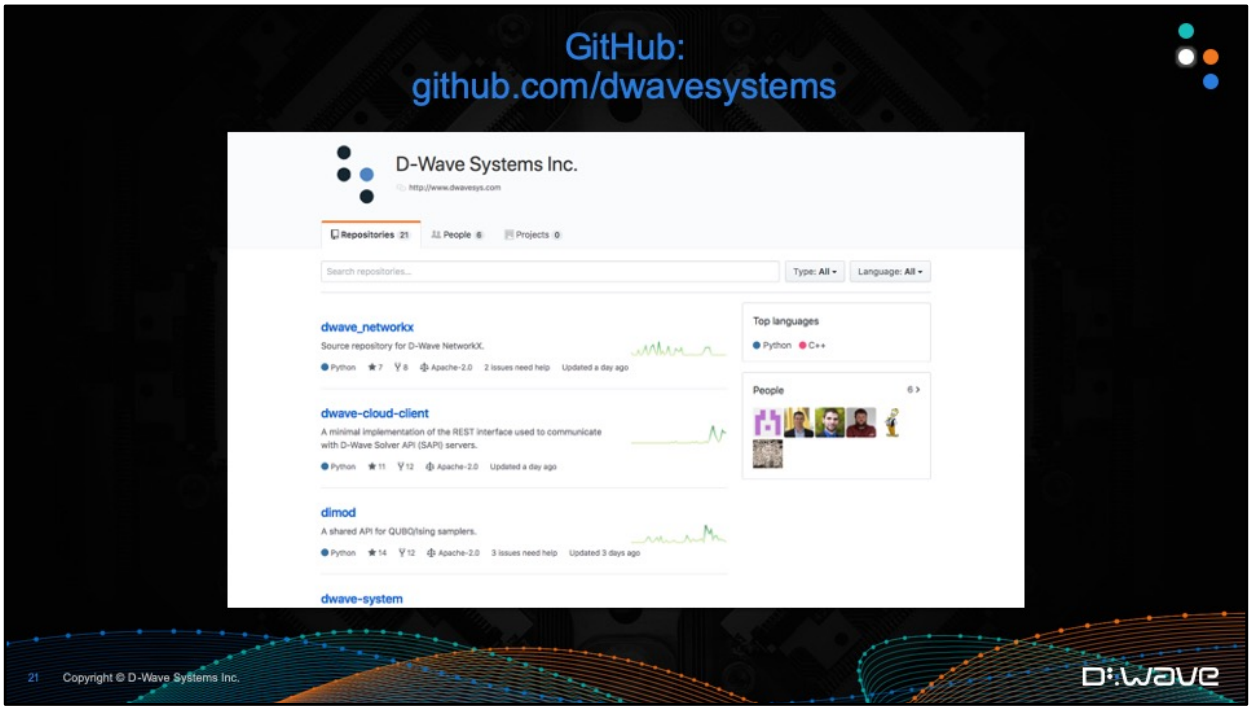
```
>>> sampleset.record.sample
array([[0, 0, 1, 0],
       [1, 1, 0, 1],
       [0, 0, 0, 1],
       [0, 1, 1, 0],
       [1, 1, 1, 1],
       [0, 0, 0, 0],
       [1, 0, 0, 1],
       [1, 1, 1, 0]], dtype=int8)
```

19 Copyright © D-Wave Systems Inc.

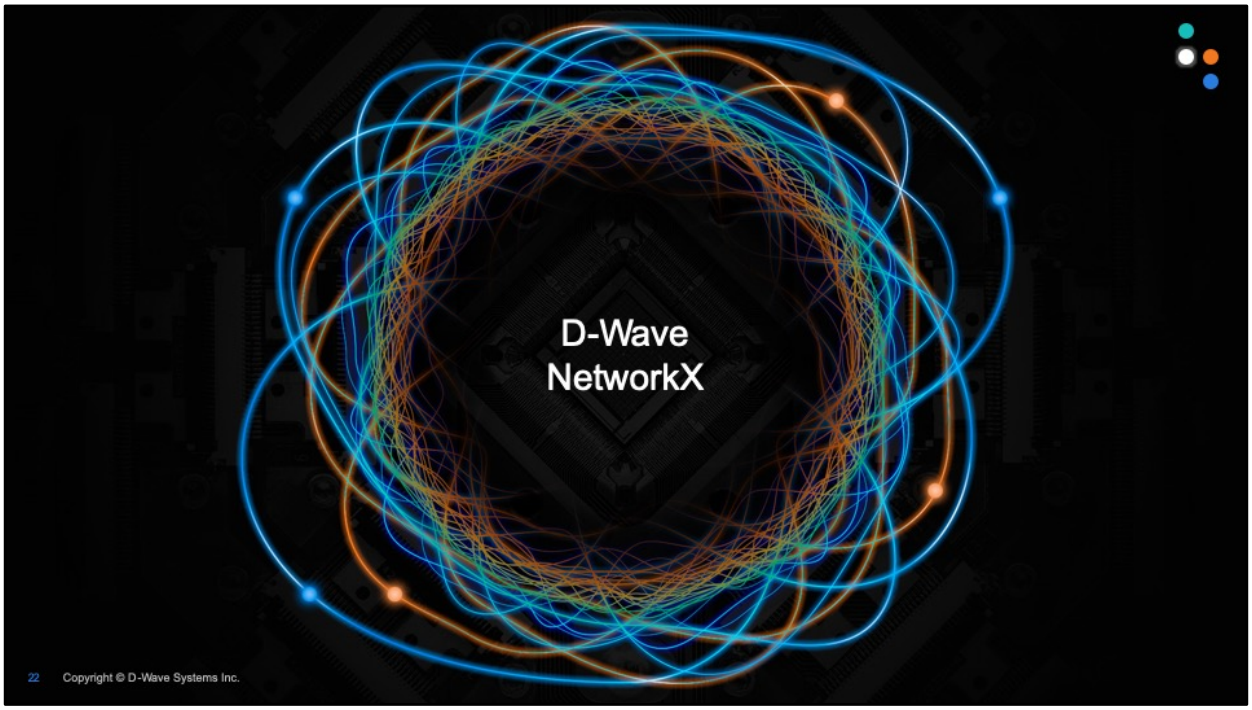
While the sample set object contains a lot of useful information, it can be a bit confusing in the beginning how to get your hands on the actual samples and energy values. Here are two different ways that you can access that information in your python programs. There are many different ways to access the data within a sampleset depending on what you want to do with it. You can read more about these different methods in our documentation – just search for sampleset.



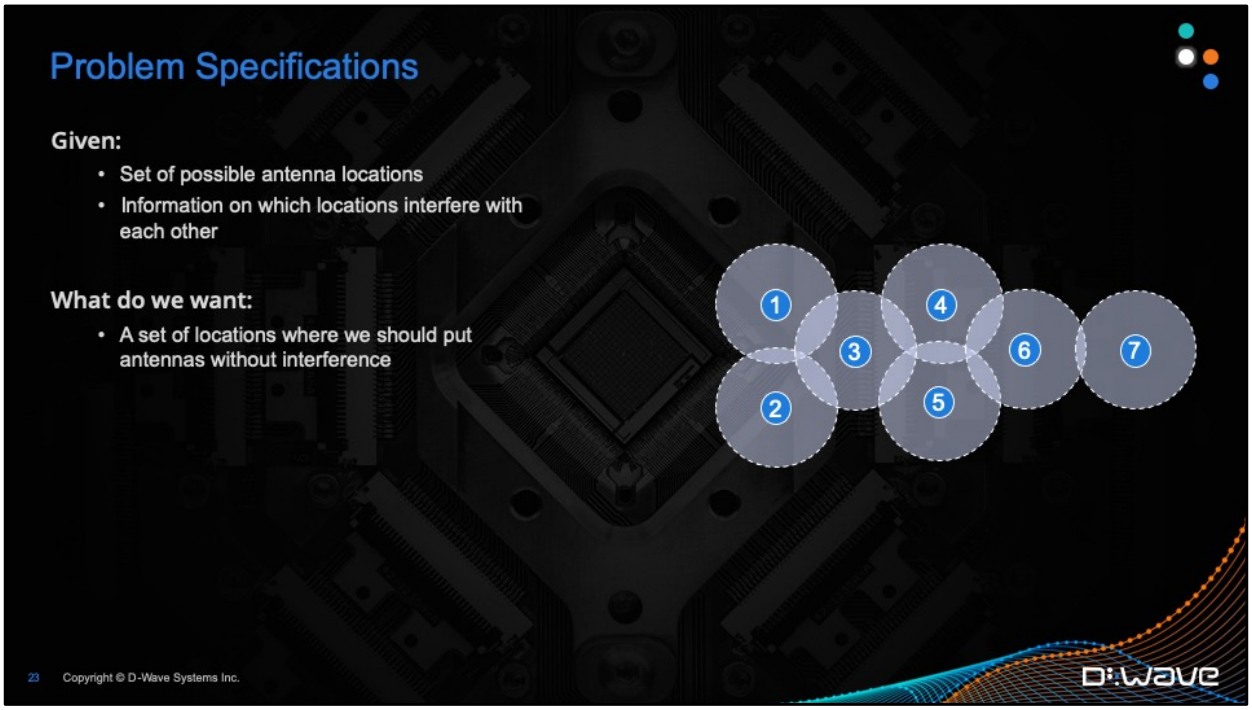
For more information take a look at our Ocean documentation at the link shown here.



You can see the code used in our Ocean SDK on our dwavesystems page on GitHub.



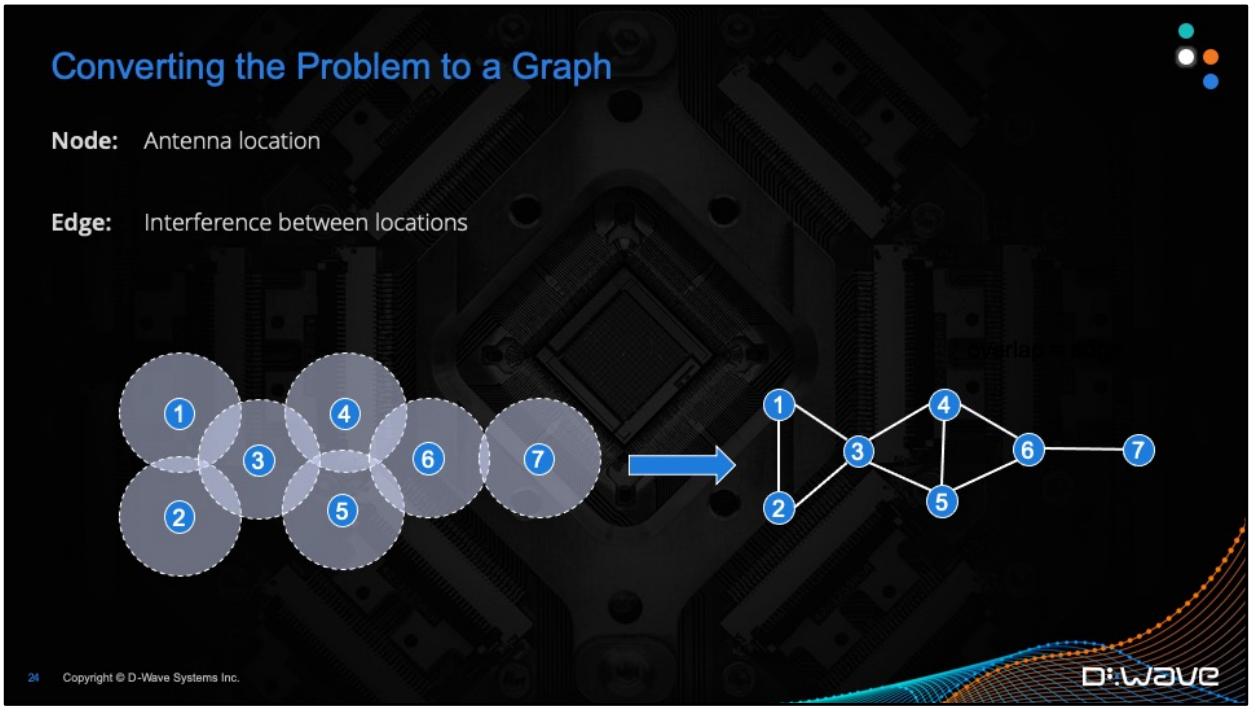
Let’s look at some basic Ocean programs that use some of our algorithm tools in D-Wave NetworkX.



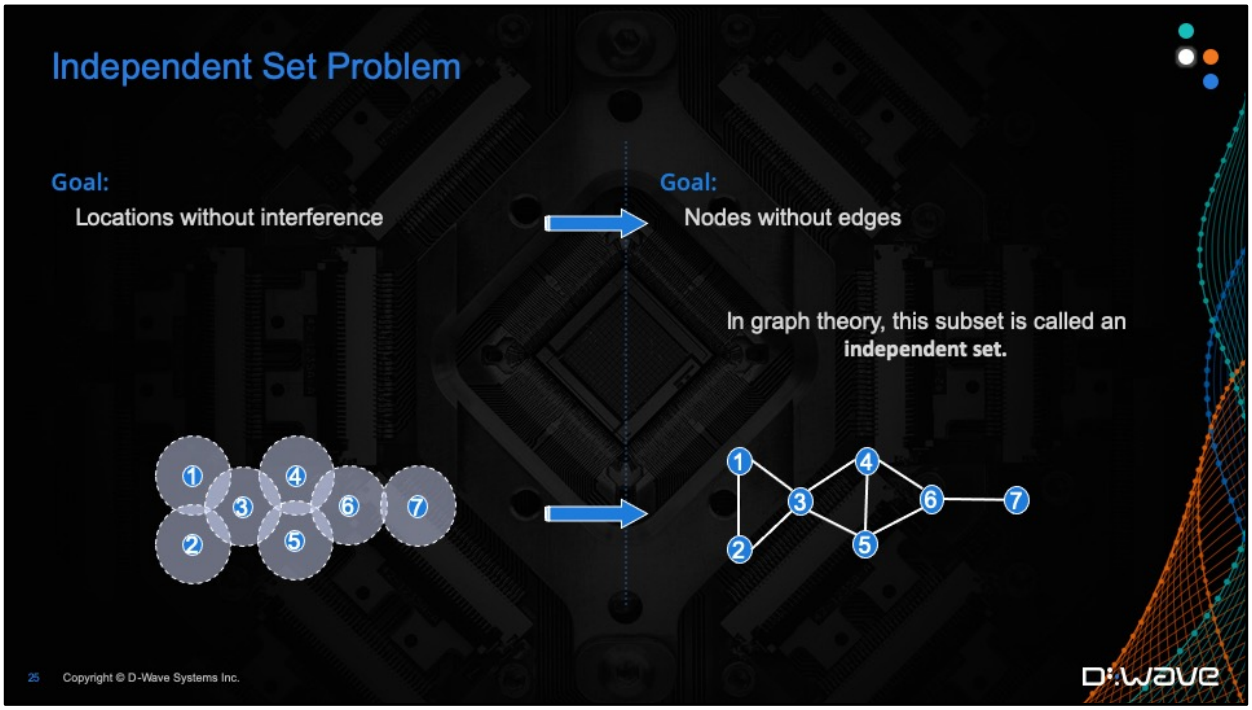
Let's go through a real-world example.

We start with a set of possible antenna locations. The dark blue numbered are all positions where we could possibly build an antenna. We want to choose locations to build antennas so that they don't interfere with each other. If antennas are too close together, they can interfere with each other. These lighter dashed and shaded circles represent the interference for each location. If two of these shaded circles overlap, then if we were to build antennas at both locations, we would have interference. For example, locations one and two have overlapping interference zones, and so those antennas would interfere with each other, which we don't want.

We're looking for the largest set of places to build antennas so that we don't have any interference. In other words, if we look at just the locations where we're going to build antennas, there won't be any of these overlapping circles.

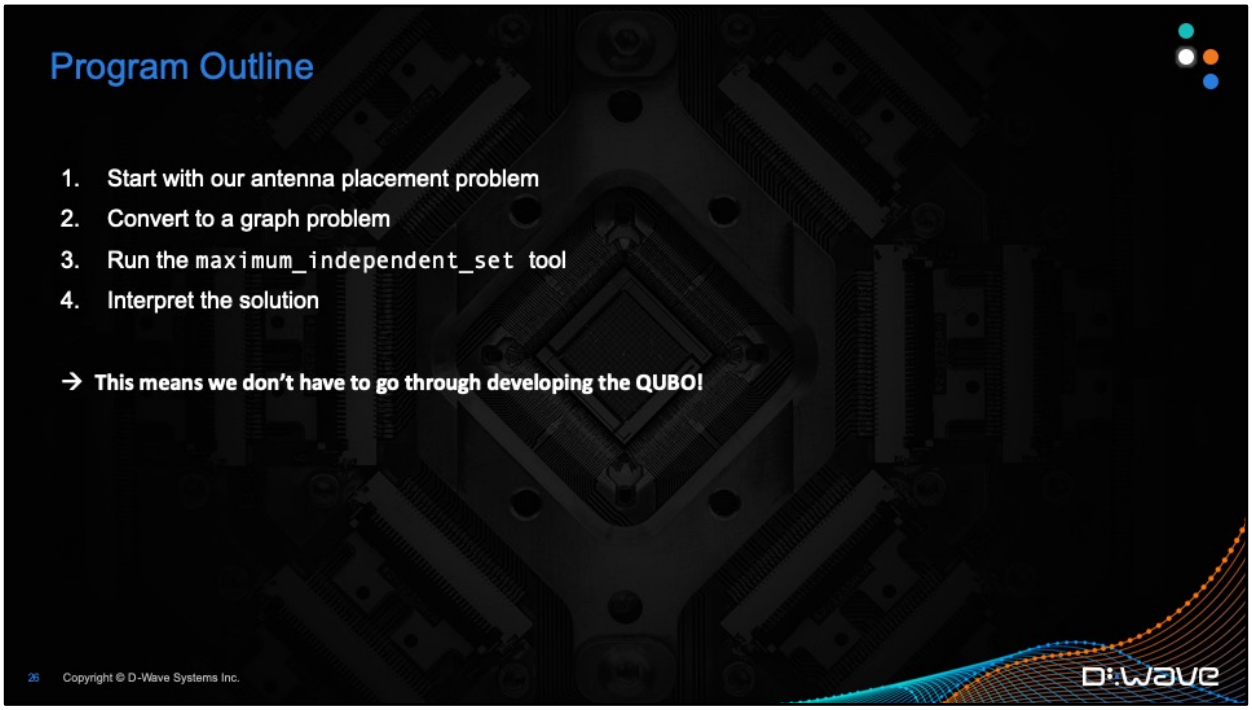


Our first step is to convert this problem into a graph, so that we can visualize what's going on. I take each location, each of these dark blue numbered circles, and make that a node in our graph. If two of the locations have their light circles overlapping, or if the two locations interfere with each other, I'm going to draw an edge between them. This way I can take my real-world problem from the problem domain and map it down into a graph problem. This will help us see what is going on a little bit easier.

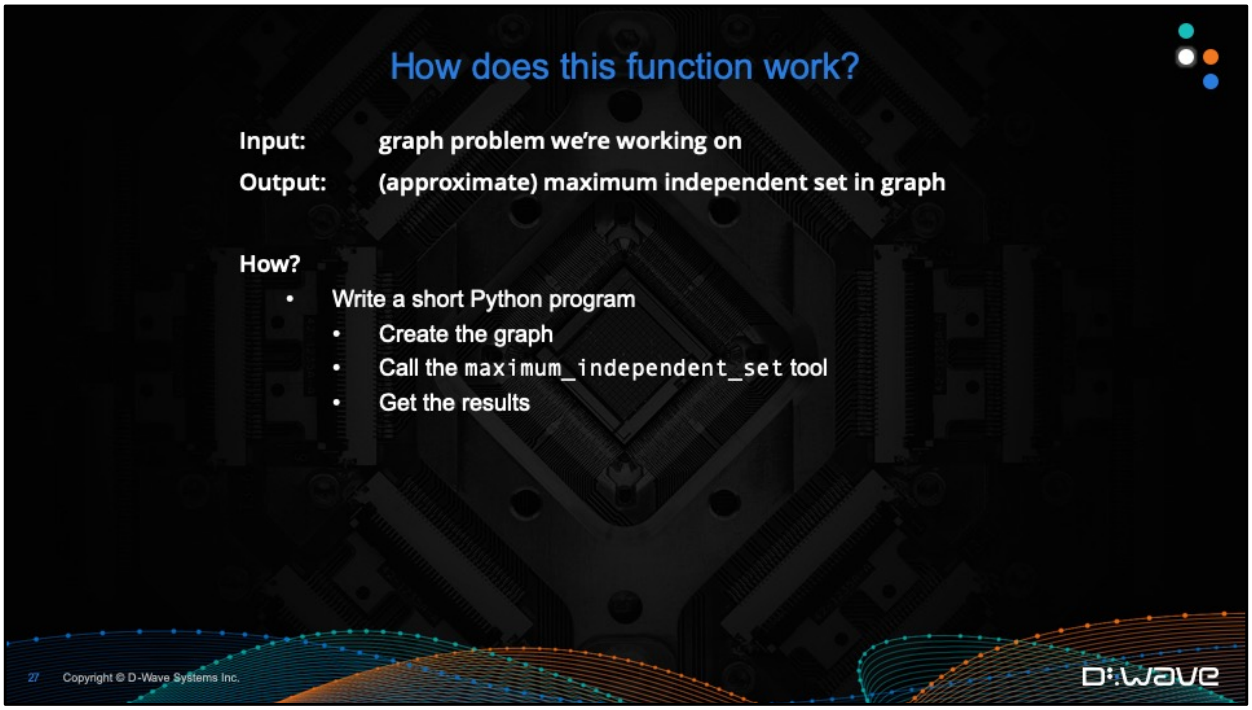


As we convert this to a graph problem, we go from looking for “where do we build antennas”, to “what is the best subset”. Remember interference maps to edge, so no interference maps to no edge.

In the graph problem that means that we are looking for a subset of the nodes with no edges between them, this is what’s called an independent set. This is a known problem in graph theory that has many real-world applications and variations. This falls in that class of NP-complete or NP-hard problems that can be very difficult for classical computers to run.



In this example, we could have gone through building a QUBO for our problem. But, in some of these cases it's good to see if there's already an algorithm tool there for you. If there is, then you can just run the existing tools as shown in step 3 – run the maximum independent set tool – and look at the solution. You can save yourself some work if there's already a QUBO algorithm tool for your problem. Check out the specialized packages in Ocean and see what's there and if you can make it work for your problem before you start from scratch.



Let's look at how we use this maximum independent set algorithm function. We define a graph, call the function on that graph and then it's going to give us back a list of nodes. Notice that we're not going down to binary variables at all, we give it a built graph that's all there and we'll get back an answer like "use nodes 2, 3, and 6".

Python Program using QPU

```
import networkx as nx
import dwave_networkx as dnx
from dwave.system import DWaveSampler, EmbeddingComposite

my_sampler = EmbeddingComposite(DWaveSampler())

G = nx.Graph()
G.add_edges_from([(1,2),(1,3),(2,3),(3,4),(3,5),(4,5),(4,6),
                  (5,6),(6,7)])

S = dnx.maximum_independent_set(G, sampler=my_sampler)
```

Sets the sampler

Creates our graph

Solves the problem

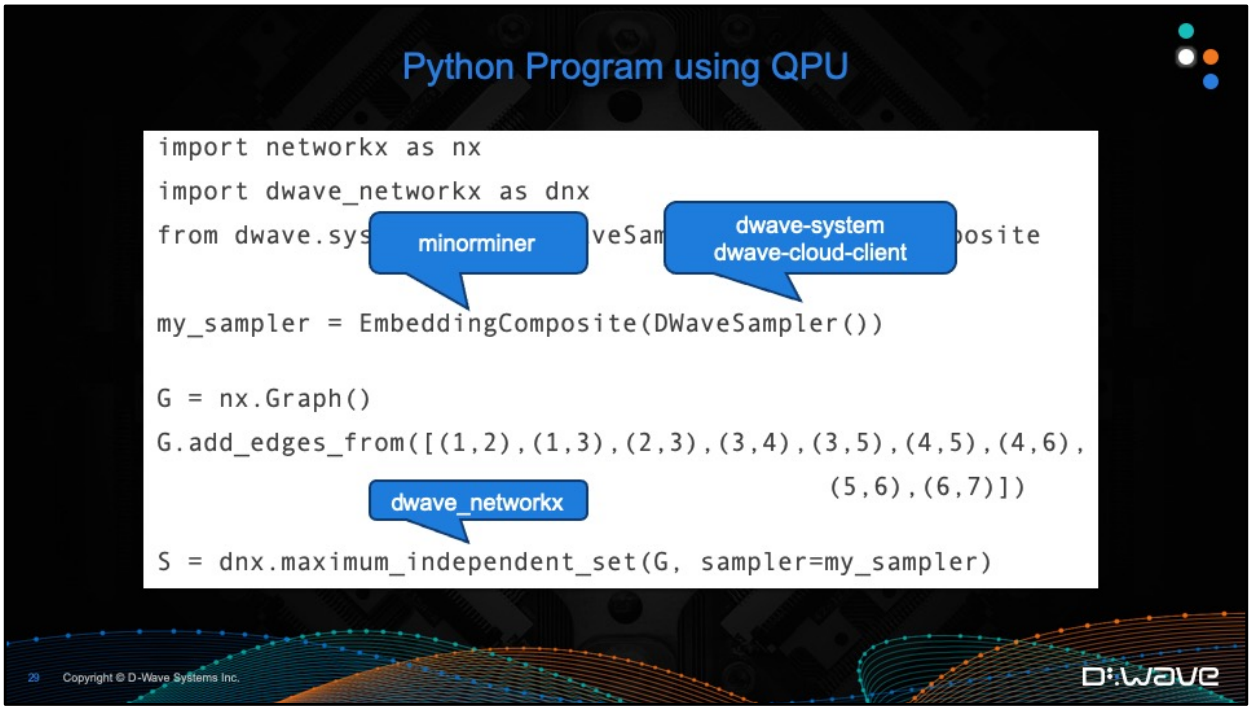
28 Copyright © D-Wave Systems Inc.

Here's an example of what that program might look like. First we import packages at the top and we define our sampler. We're using EmbeddingComposite with DWaveSampler in this example to run our problem on the quantum computer.

We define our graph using the NetworkX package – this is a really nice package if you're working with graph problems.

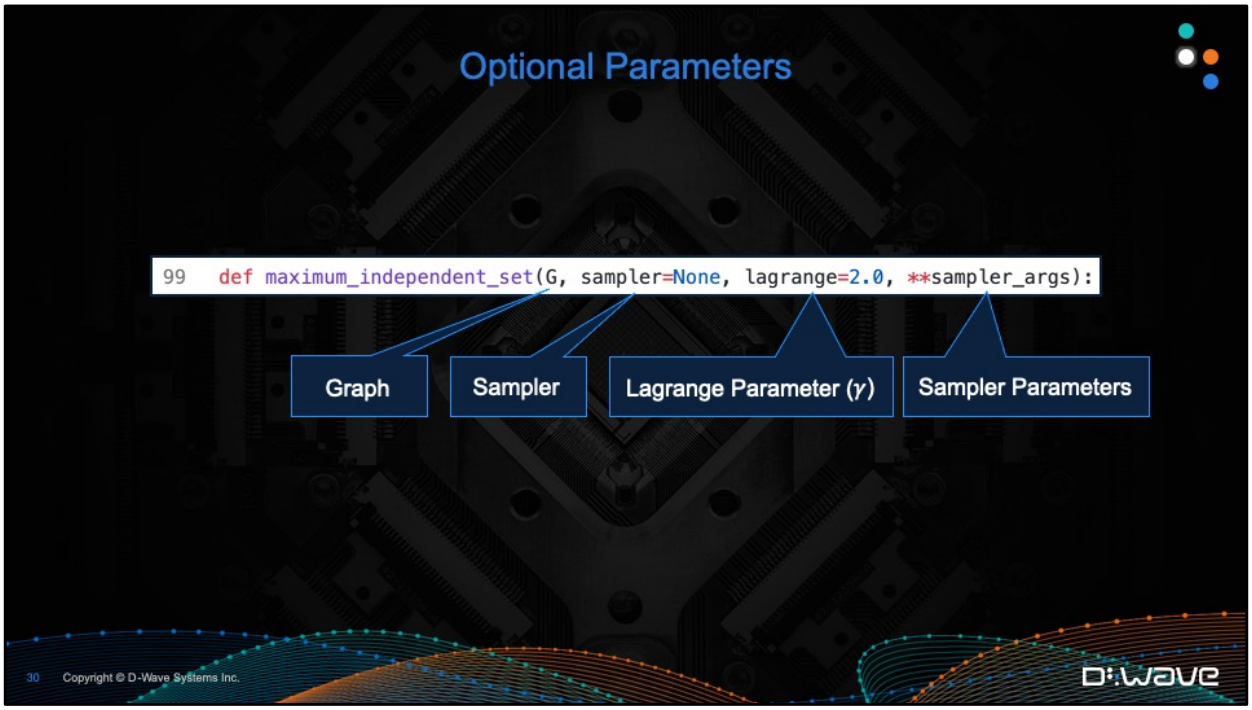
We define an empty graph (that's that first line nx.graph()) and then we build up the graph by adding in edges between nodes. As we add in these edges, any nodes that don't yet exist are added in as well.

Let's now look at the last line – this is where all of the good stuff is going on. This is where the software is defining the QUBO, sending it over to the sampler we've defined, sending the solution back to us, and converting it into this list of nodes in our graph.



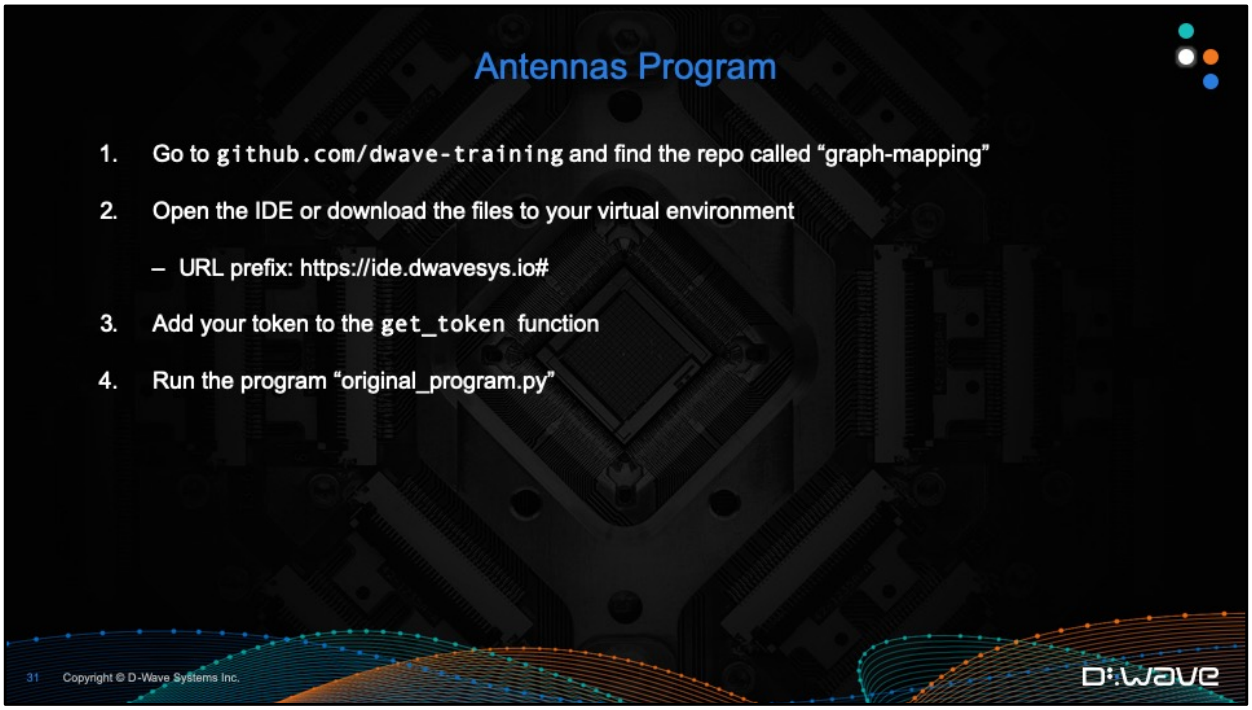
Note that there's lots of different packages used under the hood here. A lot of our packages aren't called directly, but they all contribute under the hood. To read more about this example and try it yourself, please check out the code example linked in Canvas.

Code example: https://github.com/dwave-examples/antenna-selection

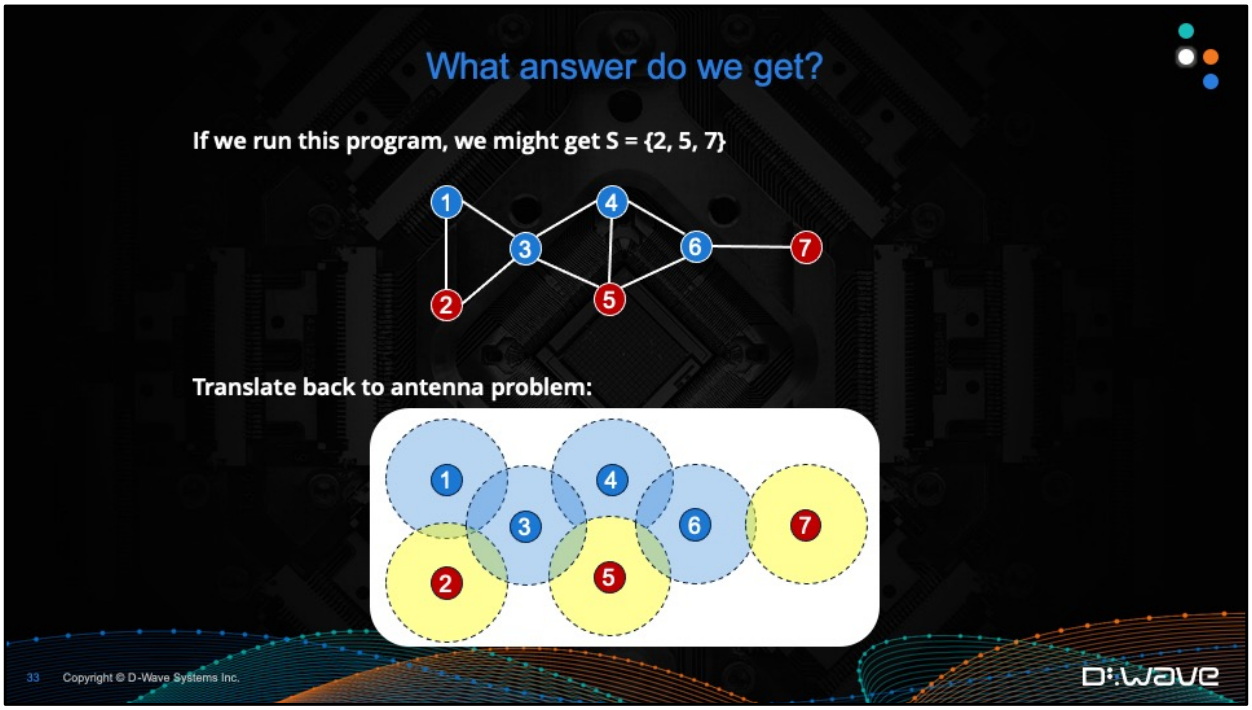


Many of our functions have some optional parameters available. This maximum independent set function requires a graph G, and a sampler so that it knows what to use to solve the problem.

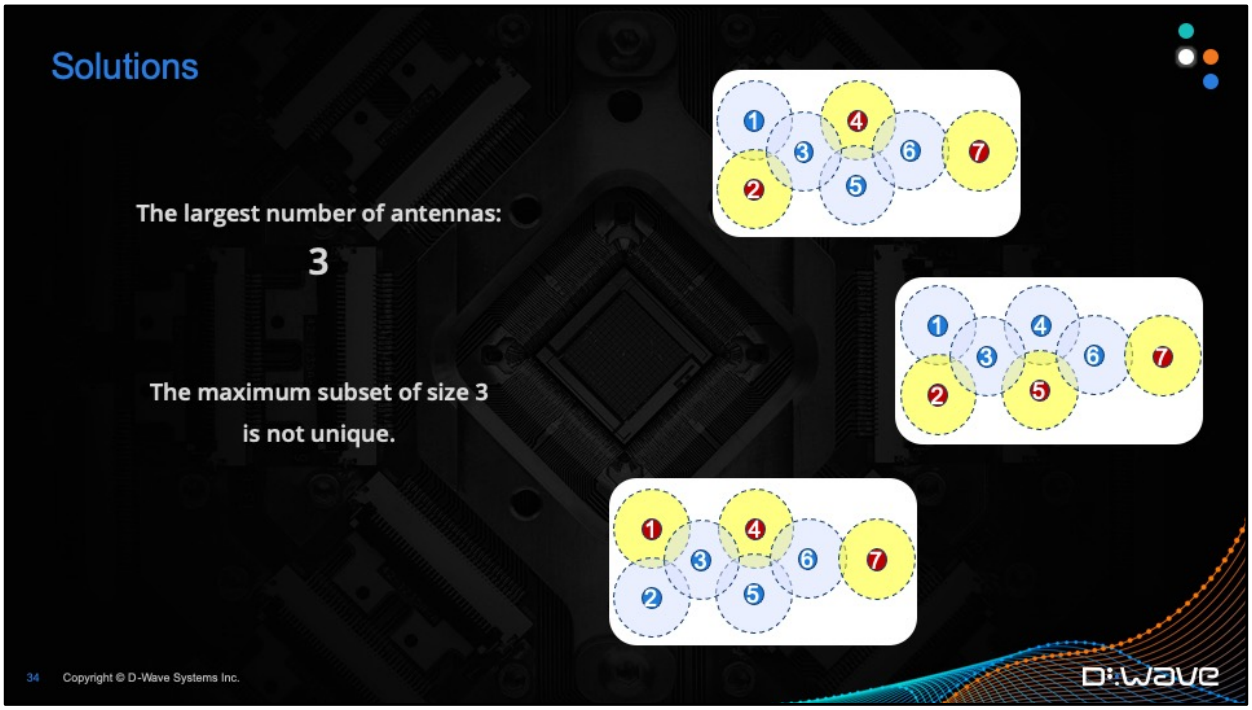
We have an optional parameter called the Lagrange parameter, which is also good to include. There's a default value set but you probably want to adjust it for your problem. We'll explore this parameter more in a later module. We also have parameters for the sampler that we choose to use for our problem. If we're using the QPU these are things like number of reads, or any of the more advanced parameters that you want to use.



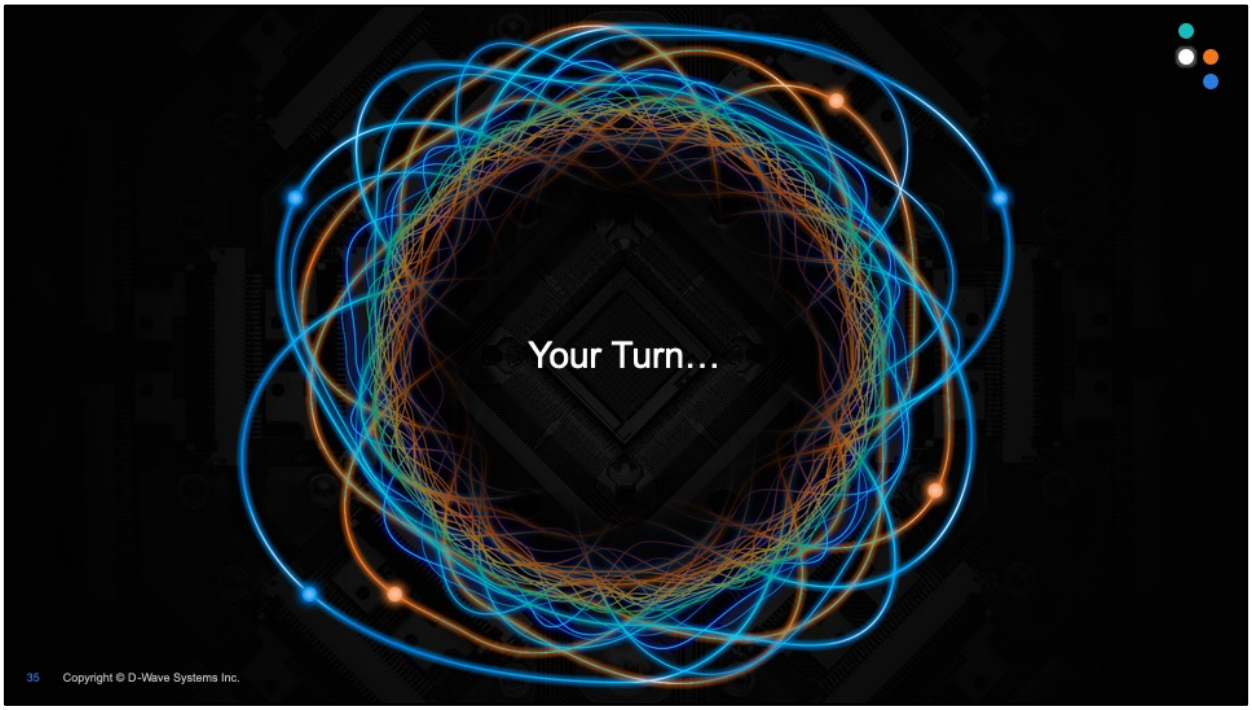
To run our program we’ll go through the following steps.



When I ran the program, the answer that was returned was 2, 5, and 7. For the graph that we're working with, that would look like this graph solution here with the red nodes in our subset. When we translate back to the antennas problem that we started with, this tells us that we want to build our antennas at locations 2, 5, and 7. Keep in mind this is just one possible solution.



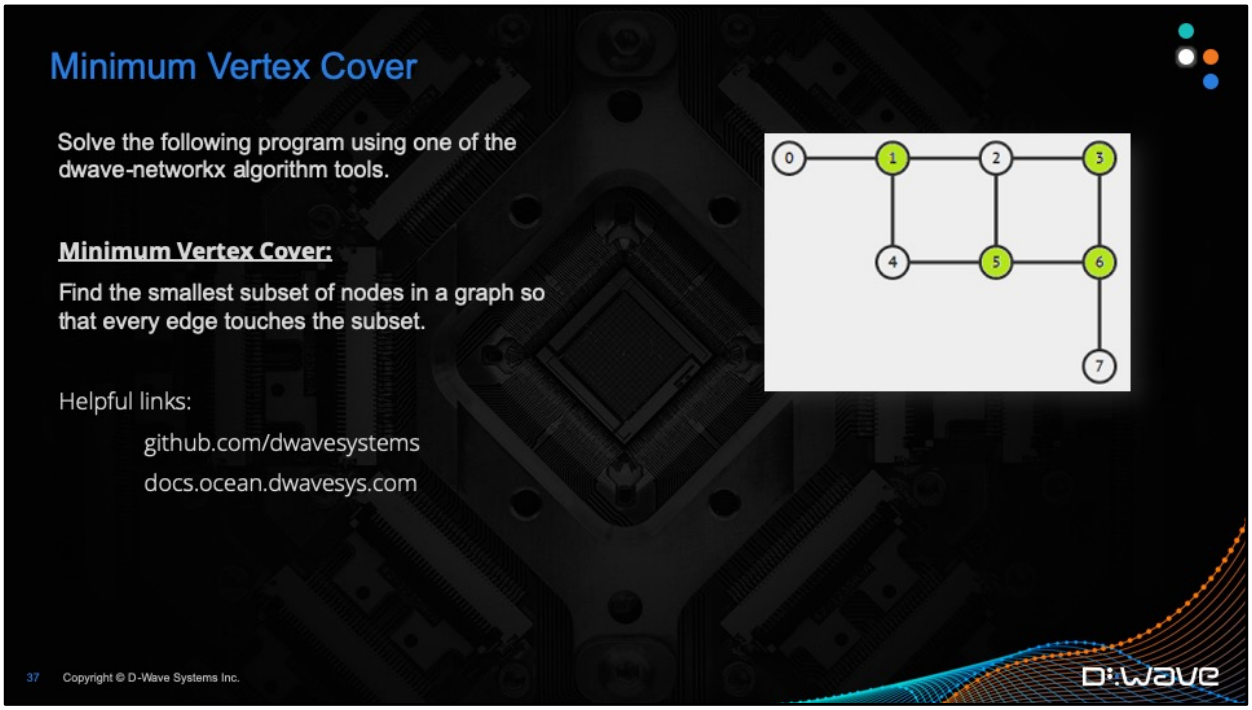
Here are a few other solutions with the same size subset, or number of antennas.



Now it’s your turn!

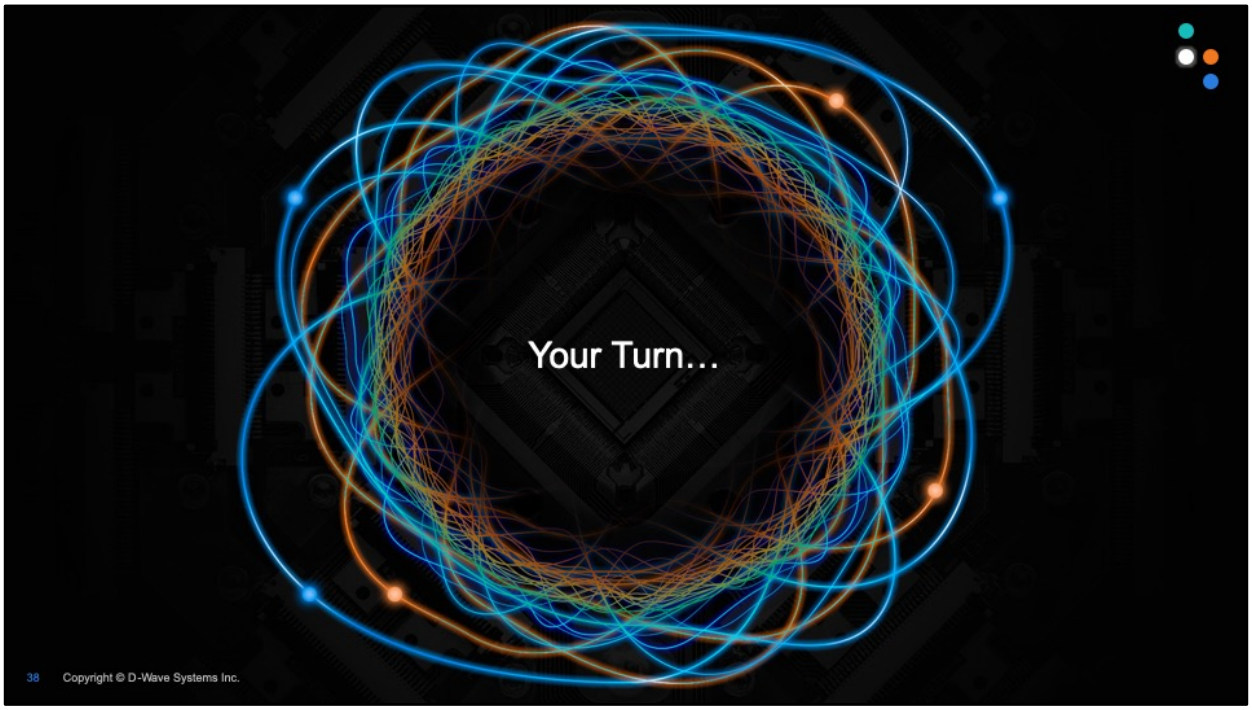
For the programming assignment for this module, you're going to think about a network of power lines and towers, similar to antennas but a little different. This time instead of building antennas and locations, we're looking for a set of junctions from which we can monitor every set of lines between the towers.

Image from https://search.creativecommons.org/photos/2cacc1f3-c02d-4241-b57b-26ff4541f038 (public domain)

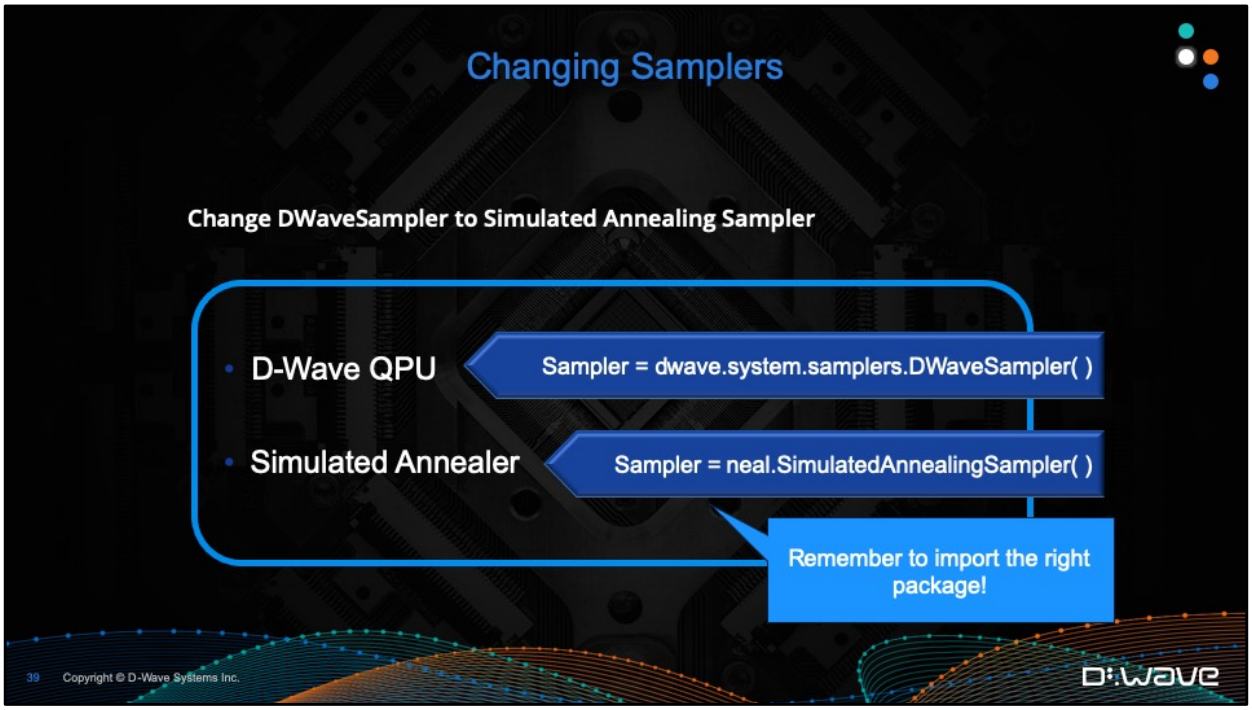


This an application of the minimum vertex cover problem. A vertex cover in a graph is a subset of the nodes so that every edge has at least one endpoint in the subset.

For this programming exercise you will take the provided code in the file "change_problem.py" and fill in the code to solve the minimum vertex cover problem on the graph shown here. The full details are provided in the README file on the GitHub repository. You'll want to look at the documentation and even over on our github page for the Ocean SDK, and see if you can figure out which of the algorithm tools can you use for this problem – for the minimum vertex cover problem.

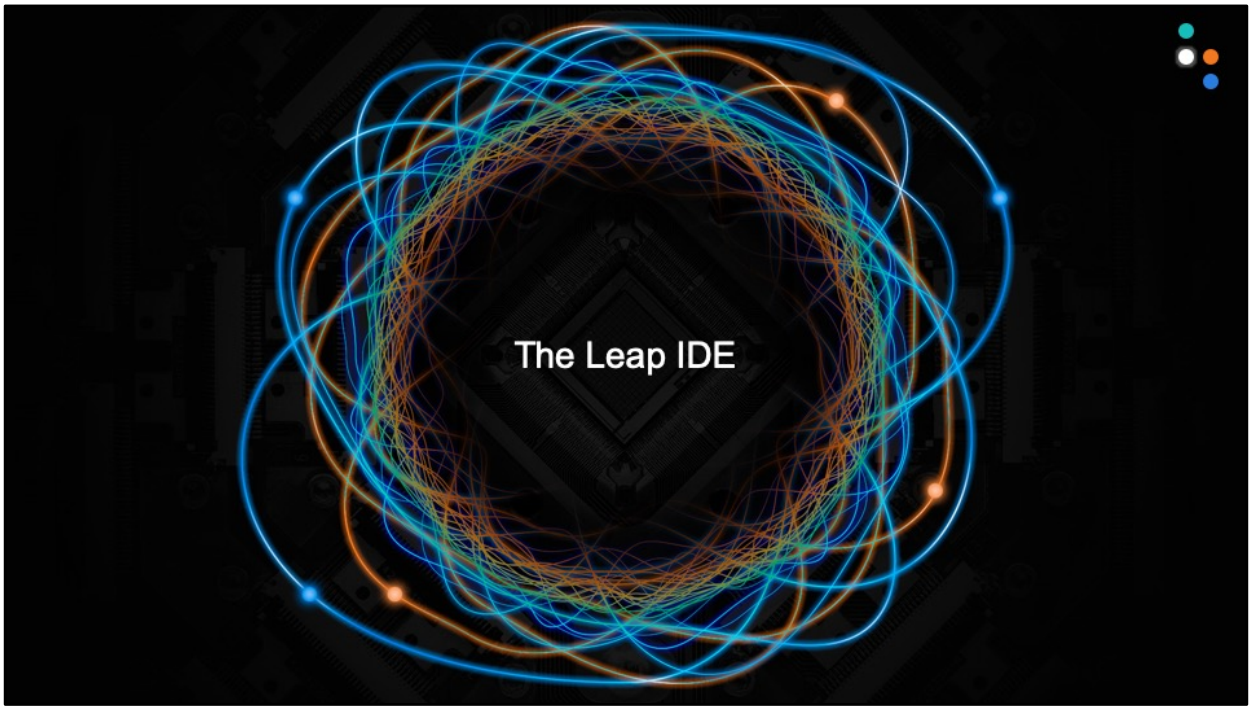


Once you've finished that part, we have another exercise for you to work on.



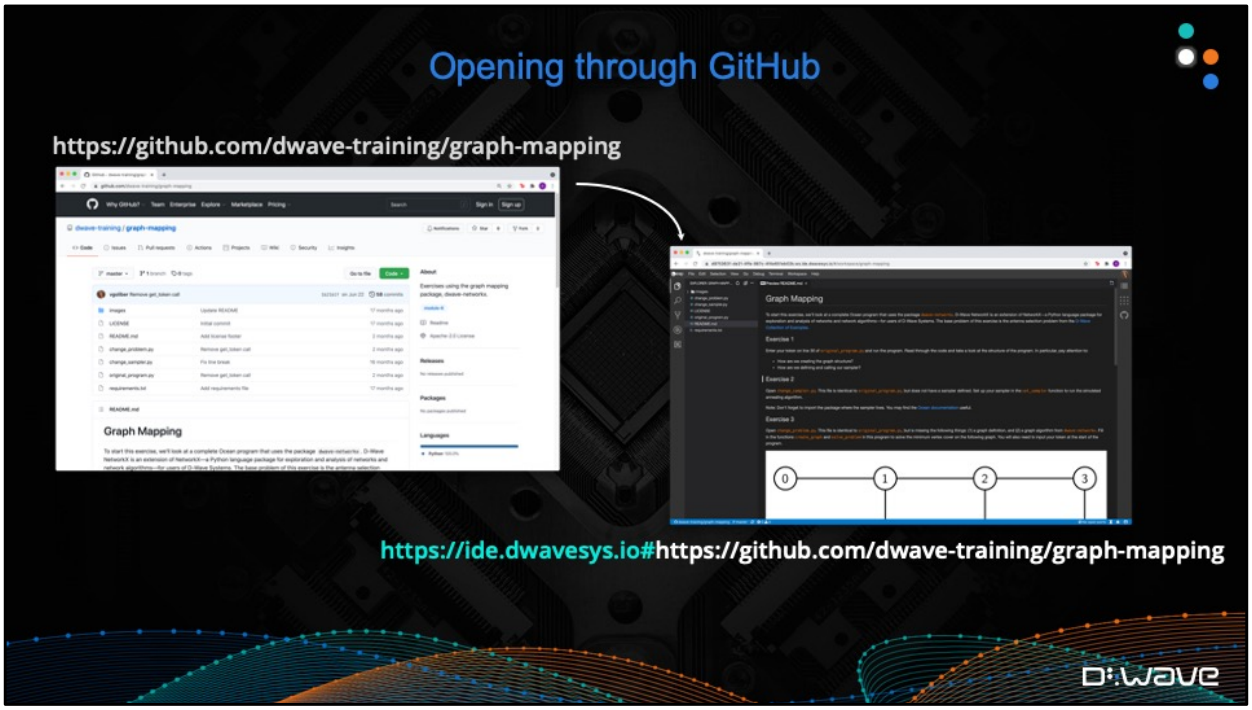
In the GitHub repository there's another python file called "change_sampler.py". In this program you'll fill out the "set_sampler" function to use our simulated annealing sampler in the Ocean SDK. Take a look at the documentation to see what the package is called, how to set up the import statements and define the sampler using simulated annealing.

Don't forget that you can post any questions on Canvas.



To use the Ocean SDK, the easiest way to get started is using the Leap Integrated Developer Environment, or IDE.

The Leap IDE provides an in-browser coding environment that comes with all of the Ocean tools pre-installed. Since it opens in your web browser, you can use it anywhere, on any computer. For this class we will have our programming exercises set up through GitHub, so you can open the exercises right from GitHub.



To open an exercise in the IDE from GitHub, navigate to the exercise page on GitHub and add https://ide.dwavesys.io# at the start of the URL, as shown here. This will prompt you to log in to Leap and start up a workspace with the assignment loaded.