

! Important

IBM Quantum Platform is moving and this version will be sunset on July 1. To get started on the new platform, [read the migration guide](#).

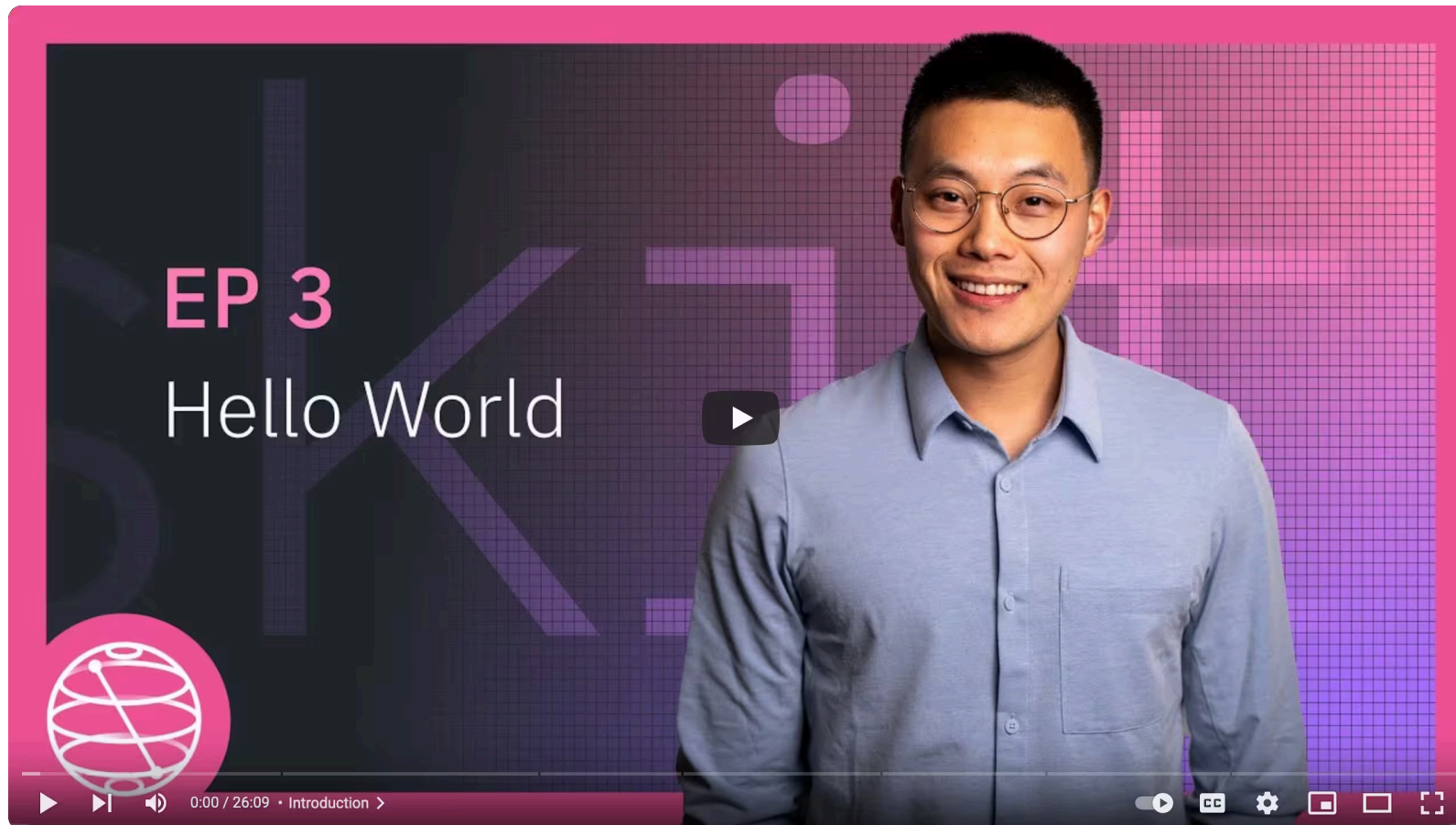
Hello world

i Note

This documentation is relevant to IBM Quantum® Platform Classic. If you need the newer version, go to the new [IBM Quantum Platform documentation](#). ↗

Package versions

This example contains two parts. You will first create a simple quantum program and run it on a quantum processing unit (QPU). Because actual quantum research requires much more robust programs, in the second section ([Scale to large numbers of qubits](#)), you will scale the simple program up to utility level. You can also follow along with the Hello World episode of the Coding with Qiskit 1.0 video series.



Coding with Qiskit 1.x, Episode 3: Hello world

Note

This video uses the `QiskitRuntimeService.get_backend` method, which has since been deprecated. Use `QiskitRuntimeService.backend` instead.

Before you begin

Follow the [Install and set up](#) instructions if you haven't already, including the steps to [Set up to use IBM Quantum® Platform](#).

The code examples found in the IBM Quantum Platform documentation were developed by using [Jupyter](#) [notebooks](#). To follow along with the examples, it is recommended that you set up an environment to run Jupyter notebooks [locally](#) or [online](#). Be sure to install the recommended extra visualization support (`'qiskit[visualization]'`). You'll also need the `matplotlib` package for the second part of this example.

To learn about quantum computing in general, visit the [Basics of quantum information course](#) [in IBM Quantum Learning](#).

IBM® is committed to the responsible development of quantum computing. Learn more about responsible quantum at IBM and review our responsible quantum principles in the [Responsible quantum computing and inclusive tech](#) topic.

Create and run a simple quantum program

The four steps to writing a quantum program using Qiskit patterns are:

1. Map the problem to a quantum-native format.
2. Optimize the circuits and operators.
3. Execute using a quantum primitive function.
4. Analyze the results.

Step 1. Map the problem to a quantum-native format

In a quantum program, *quantum circuits* are the native format in which to represent quantum instructions, and *operators* represent the observables to be measured. When creating a circuit, you'll usually create a new `QuantumCircuit` object, then add instructions to it in sequence.

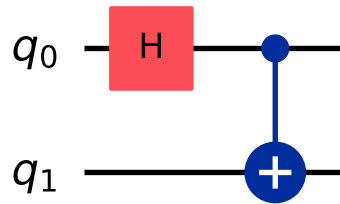
The following code cell creates a circuit that produces a *Bell state*, which is a state wherein two qubits are fully entangled with each other.

Note: bit ordering

The Qiskit SDK uses the LSb 0 bit numbering where the n^{th} digit has value $1 \ll n$ or 2^n . For more details, see the [Bit-ordering in the Qiskit SDK](#) topic.

```
1 from qiskit import QuantumCircuit
2 from qiskit.quantum_info import SparsePauliOp
3 from qiskit.transpiler import generate_preset_pass_manager
4 from qiskit_ibm_runtime import EstimatorV2 as Estimator
5
6 # Create a new circuit with two qubits
7 qc = QuantumCircuit(2)
8
9 # Add a Hadamard gate to qubit 0
10 qc.h(0)
11
12 # Perform a controlled-X gate on qubit 1, controlled by qubit 0
13 qc.cx(0, 1)
14
15 # Return a drawing of the circuit using Matplotlib ("mpl").
16 # These guides are written by using Jupyter notebooks, which
17 # display the output of the last line of each cell.
18 # If you're running this in a script, use `print(qc.draw())` to
19 # print a text drawing.
20 qc.draw("mpl")
```

Output:



See [QuantumCircuit](#) in the documentation for all available operations.

When creating quantum circuits, you must also consider what type of data you want returned after execution. Qiskit provides two ways to return data: you can obtain a probability distribution for a set of qubits you choose to measure, or you can obtain the expectation value of an observable. Prepare your workload to measure your circuit in one of these two ways with [Qiskit primitives](#) (explained in detail in [Step 3](#)).

This example measures expectation values by using the `qiskit.quantum_info` submodule, which is specified by using operators (mathematical objects used to represent an action or process that changes a quantum state). The following code cell creates six two-qubit Pauli operators: `IZ`, `IX`, `ZI`, `XI`, `ZZ`, and `XX`.

```
1 | # Set up six different observables.
2 |
3 | observables_labels = ["IZ", "IX", "ZI", "XI", "ZZ", "XX"]
4 | observables = [SparsePauliOp(label) for label in observables_labels]
```

Operator Notation

Here, something like the `ZZ` operator is a shorthand for the tensor product $Z \otimes Z$, which means measuring Z on qubit 1 and Z on qubit 0 together, and obtaining information about the correlation between qubit 1 and qubit 0. Expectation values like this are also typically written as $\langle Z_1 Z_0 \rangle$.

If the state is entangled, then the measurement of $\langle Z_1 Z_0 \rangle$ should be different from the measurement of $\langle I_1 \otimes Z_0 \rangle \langle Z_1 \otimes I_0 \rangle$. For the specific entangled state created by our circuit described above, the measurement of $\langle Z_1 Z_0 \rangle$ should be 1 and the measurement of $\langle I_1 \otimes Z_0 \rangle \langle Z_1 \otimes I_0 \rangle$ should be zero.

Step 2. Optimize the circuits and operators

When executing circuits on a device, it is important to optimize the set of instructions that the circuit contains and minimize the overall depth (roughly the number of instructions) of the circuit. This ensures that you obtain the best results possible by reducing the effects of error and noise. Additionally, the circuit's instructions must conform to a backend device's [Instruction Set Architecture \(ISA\)](#) and must consider the device's basis gates and qubit connectivity.

The following code instantiates a real device to submit a job to and transforms the circuit and observables to match that backend's ISA. It requires that you have already [saved your credentials](#)

```
1  from qiskit_ibm_runtime import QiskitRuntimeService
2
3  service = QiskitRuntimeService()
4
5  backend = service.least_busy(simulator=False, operational=True)
6
7  # Convert to an ISA circuit and layout-mapped observables.
8  pm = generate_preset_pass_manager(backend=backend, optimization_level=1)
9  isa_circuit = pm.run(qc)
10
11 isa_circuit.draw("mpl", idle_wires=False)
```

Output:

Step 3. Execute using the quantum primitives

Quantum computers can produce random results, so you usually collect a sample of the outputs by running the circuit many times. You can estimate the value of the observable by using the `Estimator` class. `Estimator` is one of two **primitives**; the other is `Sampler`, which can be used to get data from a quantum computer. These objects possess a `run()` method that executes the selection of circuits, observables, and parameters (if applicable), using a **primitive unified bloc (PUB)**.

```
1  # Construct the Estimator instance.
2
3  estimator = Estimator(mode=backend)
4  estimator.options.resilience_level = 1
5  estimator.options.default_shots = 5000
6
7  mapped_observables = [
8      observable.apply_layout(isa_circuit.layout) for observable in observables
9  ]
10
11 # One pub, with one circuit to run against five different observables.
12 job = estimator.run([(isa_circuit, mapped_observables)])
13
14 # Use the job ID to retrieve your job data later
15 print(f">>> Job ID: {job.job_id()}")
```

Output:

```
>>> Job ID: czz0bqerxz8g008f8fkg
```


After a job is submitted, you can wait until either the job is completed within your current python instance, or use the `job_id` to retrieve the data at a later time. (See the [section on retrieving jobs](#) for details.)

After the job completes, examine its output through the job's `result()` attribute.

```
1  # This is the result of the entire submission. You submitted one Pub,
2  # so this contains one inner result (and some metadata of its own).
3  job_result = job.result()
4
5  # This is the result from our single pub, which had six observables,
6  # so contains information on all six.
7  pub_result = job.result()[0]
```


Alternative: run the example using a simulator

When you run your quantum program on a real device, your workload must wait in a queue before it runs. To save time, you can instead use the following code to run this small workload on the `fake_provider` with the Qiskit Runtime local testing mode. Note that this is only possible for a small circuit. When you scale up in the next section, you will need to use a real device.



```
1
2 # Use the following code instead if you want to run on a simulator:
3
4 from qiskit_ibm_runtime.fake_provider import FakeAlmadenV2
5 backend = FakeAlmadenV2()
6 estimator = Estimator(backend)
7
8 # Convert to an ISA circuit and layout-mapped observables.
9
10 pm = generate_preset_pass_manager(backend=backend, optimization_level=1)
11 isa_circuit = pm.run(qc)
12 mapped_observables = [
13     observable.apply_layout(isa_circuit.layout) for observable in observables
14 ]
15
16 job = estimator.run([(isa_circuit, mapped_observables)])
17 result = job.result()
18
19 # This is the result of the entire submission. You submitted one Pub,
20 # so this contains one inner result (and some metadata of its own).
21
22 job_result = job.result()
23
24 # This is the result from our single pub, which had five observables,
25 # so contains information on all five.
26
27 pub_result = job_result[0]
```

Step 4. Analyze the results

The analyze step is typically where you might postprocess your results using, for example, measurement error mitigation or zero noise extrapolation (ZNE). You might feed these results into another workflow for further analysis or prepare a plot of the key values and data. In general, this step is specific to your problem. For this example, plot each of the expectation values that were measured for our circuit.

The expectation values and standard deviations for the observables you specified to Estimator are accessed through the job result's `PubResult.data.evs` and `PubResult.data.stds` attributes. To obtain the results from Sampler, use the `PubResult.data.meas.get_counts()` function, which will return a `dict` of measurements in the form of bitstrings as keys and counts as their corresponding values. For more information, see [Get started with Sampler](#).

```
1 | # Plot the result
2 |
3 | from matplotlib import pyplot as plt
4 |
5 | values = pub_result.data.evs
6 |
7 | errors = pub_result.data.stds
8 |
9 | # plotting graph
10 | plt.plot(observables_labels, values, "-o")
11 | plt.xlabel("Observables")
12 | plt.ylabel("Values")
13 | plt.show()
```

Output:

Notice that for qubits 0 and 1, the independent expectation values of both X and Z are 0, while the correlations ($\langle XX \rangle$ and $\langle ZZ \rangle$) are 1. This is a hallmark of quantum entanglement.

Scale to large numbers of qubits

In quantum computing, utility-scale work is crucial for making progress in the field. Such work requires computations to be done on a much larger scale; working with circuits that might use over 100 qubits and over 1000 gates. This example demonstrates how you can accomplish utility-scale work on IBM® QPUs by creating

and analyzing a 100-qubit GHZ state. It uses the Qiskit patterns workflow and ends by measuring the expectation value $\langle Z_0 Z_i \rangle$ for each qubit.

Step 1. Map the problem

Write a function that returns a `QuantumCircuit` that prepares an n -qubit GHZ state (essentially an extended Bell state), then use that function to prepare a 100-qubit GHZ state and collect the observables to be measured.

```
1 from qiskit import QuantumCircuit
2
3
4 def get_qc_for_n_qubit_GHZ_state(n: int) -> QuantumCircuit:
5     """This function will create a qiskit.QuantumCircuit (qc) for an n-qubit GHZ stat
6
7     Args:
8         n (int): Number of qubits in the n-qubit GHZ state
9
10    Returns:
11        QuantumCircuit: Quantum circuit that generate the n-qubit GHZ state, assuming
12    """
13    if isinstance(n, int) and n >= 2:
14        qc = QuantumCircuit(n)
15        qc.h(0)
16        for i in range(n - 1):
17            qc.cx(i, i + 1)
18    else:
19        raise Exception("n is not a valid input")
20    return qc
21
22
23 # Create a new circuit with two qubits (first argument) and two classical
24 # bits (second argument)
25 n = 100
26 qc = get_qc_for_n_qubit_GHZ_state(n)
```

Next, map to the operators of interest. This example uses the `ZZ` operators between qubits to examine the behavior as they get farther apart. Increasingly inaccurate (corrupted) expectation values between distant qubits would reveal the level of noise present.


```
1 from qiskit.transpiler import generate_preset_pass_manager
2 from qiskit_ibm_runtime import QiskitRuntimeService
3
4 service = QiskitRuntimeService()
5
6 backend = service.least_busy(
7     simulator=False, operational=True, min_num_qubits=100
8 )
9 pm = generate_preset_pass_manager(optimization_level=1, backend=backend)
10
11 isa_circuit = pm.run(qc)
12 isa_operators_list = [op.apply_layout(isa_circuit.layout) for op in operators]
```

Step 3. Execute on hardware

Submit the job and enable error suppression by using a technique to reduce errors called [dynamical decoupling](#). The resilience level specifies how much resilience to build against errors. Higher levels generate more accurate results, at the expense of longer processing times. For further explanation of the options set in the following code, see [Configure error mitigation for Qiskit Runtime](#).

```
1 from qiskit_ibm_runtime import EstimatorOptions
2 from qiskit_ibm_runtime import EstimatorV2 as Estimator
3
4 options = EstimatorOptions()
5 options.resilience_level = 1
6 options.dynamical_decoupling.enable = True
7 options.dynamical_decoupling.sequence_type = "XY4"
8
9 # Create an Estimator object
10 estimator = Estimator(backend, options=options)
```



```
1 | # Submit the circuit to Estimator
2 | job = estimator.run([isa_circuit, isa_operators_list])
3 | job_id = job.job_id()
4 | print(job_id)
```

Output:

```
czz0f446rr3g008mjckg
```

Step 4. Post-process results

After the job completes, plot the results and notice that $\langle Z_0 Z_i \rangle$ decreases with increasing i , even though in an ideal simulation all $\langle Z_0 Z_i \rangle$ should be 1.

```
1 import matplotlib.pyplot as plt
2 from qiskit_ibm_runtime import QiskitRuntimeService
3
4 # data
5 data = list(range(1, len(operators) + 1)) # Distance between the Z operators
6 result = job.result()[0]
7 values = result.data.evs # Expectation value at each Z operator.
8 values = [
9     v / values[0] for v in values
10 ] # Normalize the expectation values to evaluate how they decay with distance.
11
12 # plotting graph
13 plt.plot(data, values, marker="o", label="100-qubit GHZ state")
14 plt.xlabel("Distance between qubits $i$")
15 plt.ylabel(r"$\langle Z_i Z_0 \rangle / \langle Z_1 Z_0 \rangle$")
16 plt.legend()
17 plt.show()
```

Output:

The previous plot shows that as the distance between qubits increases, the signal decays because of the presence of noise.

Next steps

Recommendations

- Learn how to [build circuits](#) in more detail.

- Try a [tutorial](#) in IBM Quantum Learning.

Was this page helpful?

Yes

No

Report a bug or request content on [GitHub](#).

© IBM Corp., 2017-2025