Qiskit Pocket Guide

Quantum Development with Qiskit

James L. Weaver and Frank J. Harkins

Beijing • Boston • Farnham • Sebastopol • Tokyo
O'REILLY®

Qiskit Pocket Guide

by James L. Weaver and Frank J. Harkins

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Suzanne McQuade
Development Editor: Shira Evans
Production Editor: Katherine Tozer
Copyeditor: Piper Editorial Consulting, LLC
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

July 2022: First Edition

Revision History for the First Release

2022-06-15: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098112479* for release details.

978-1-098-11247-9

[TK]

Table of Contents

Preface

Qiskit is an open source SDK (software development kit) for working with quantum computers at the level of pulses, circuits, and application modules. The purpose of this book is to provide a succinct guide for developers to use while creating applications that leverage quantum computers and simulators.

How This Book Is Structured

Our goal in this guide is to address much of the functionality of Qiskit that application developers will routinely use. Some of this Qiskit functionality is considered to be fundamental to quantum computing. Other Qiskit functionality supports quantum computing concepts such as quantum information and quantum algorithms. Qiskit has additional functionality that we've deemed essential for quantum application development. We've structured the book at a high-level according the aforementioned functionality, with individual chapters drilling into the specifics:

Part I: Qiskit Fundamentals

In the first part of the book, we show you how to use Qiskit to create quantum circuits. Quantum circuits contain instructions and gates, so we discuss how to use the ones provided in Qiskit, as well as how to create your own. We then show how to run quantum circuits on quantum computers and simulators, and demonstrate how to visualize results. To round out Part I, we discuss the *transpiler* and how it converts a quantum circuit into instructions that run on a target quantum computer or simulator.

Part II: Quantum Information and Algorithms

In the second part of this book, we discuss Qiskit modules responsible for implementing quantum information concepts (specifically states, operators, channels and measures). We also present facilities in Qiskit that implement quantum algorithms, as well as a facility known in Qiskit as *operator flow*. A developer may use some of the functionality in Part II to develop quantum applications at higher levels of abstraction than quantum circuits.

Part III: Additional Essential Functionality

In the third and final part of this book we cover essential Qiskit functionality, some of which drills into information already discussed, and some of which is newly presented. Specifically, we explore the *standard operations* of the Qiskit circuit library, and new ground is uncovered when we discuss how to work with quantum providers and backends. In addition, we'll introduce QASM 3.0, and demonstrate how to create quantum programs with this quantum assembly language.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

NOTE

This element signifies a general note.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at qisk.it/pocket-guide.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Qiskit Pocket Guide* by James L. Weaver and ~~Francis~~ Harkins (O'Reilly). Copyright 2022 James Weaver and ~~Francis~~ Harkins, 978-1-098-11247-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/qiskit-pocket-guide*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Follow us on Twitter: *https://twitter.com/oreillymedia*.

Watch us on YouTube: *https://www.youtube.com/oreillymedia*.

Acknowledgments

~~This book would not have been possible without a supporting team of innovative people at IBM Quantum and in the larger quantum computing community. The authors would like to thank the amazing O'Reilly team, including Kristen Brown, Danny Elfanbaum, Shira Evans, Zan McQuade, Jonathon Owen, and Katherine Tozer. The authors also appreciate the invaluable contributions made by technical reviewers Nick Bronn, Barry Burd, Eric Johnston, Robert Loredo, and Iskandar Sitdikov.~~

James Weaver would like to thank Julie, Lori, Kelli, Kaleb, Jillian, Levi, and Oliver for their understanding and encouragement while working on this book. As a lifelong classical developer, James is also thankful that quantum mechanical phenomena baked into nature may potentially be leveraged to solve problems not possible with classical computers. "The heavens declare the glory of God, and the sky above proclaims his handiwork~~.~~" (Psalm 19:1)

Frank Harkins would like to thank the Qiskit team for all their great work on Qiskit and its documentation~~,~~ and for answering all his questions. Frank would also like to thank Rose, Matt, Joanne, Keith, Libby, and Martha for their constant support over the course of writing this book.

PART I
Qiskit Fundamentals

Underlying all programs developed using Qiskit are some fundamental concepts and modules. In the first part of this book we'll explore these fundamentals, beginning with Chapter 1, "Quantum Circuits and Operations". In that chapter we'll demonstrate how to create quantum circuits, populate them with commonly used gates and instructions, obtain information about quantum circuits, and manipulate them.

In Chapter 2, "Running Quantum Circuits", we'll demonstrate how to use Qiskit classes and functions to run quantum circuits on quantum simulators and devices. We'll also show how to monitor the status of a job, as well as to obtain its results. Then in Chapter 3, "Visualizing Quantum Measurements and States", we'll show how to leverage graphical features of Qiskit to visualize quantum states and results.

Finally in Chapter 4, "Using the Transpiler", we'll discuss the process of *transpilation* in which the operations of a quantum circuit are converted into instructions for running on a particular quantum simulator or device.

CHAPTER 1

Quantum Circuits and Operations

In Qiskit, quantum programs are normally expressed with quantum circuits that contain quantum operations. Quantum circuits are represented by the `QuantumCircuit` class, and quantum operations are represented by subclasses of the `Instruction` class.

Constructing Quantum Circuits

A quantum circuit may be created by supplying an argument that indicates the number of desired quantum wires (qubits) for that circuit. This is often supplied as an integer:

```
QuantumCircuit(2)
```

Optionally, the number of desired classical wires (bits) may also be specified. The first argument refers to the number of quantum wires, and the second argument the number of classical wires:

```
QuantumCircuit(2, 2)
```

The number of desired quantum and classical wires may also be expressed by supplying instances of `QuantumRegister` and `ClassicalRegister` as arguments to `QuantumCircuit`. These classes are addressed in "Using the QuantumRegister Class" on page 24 and "Using the ClassicalRegister Class" on page 25.

Using the QuantumCircuit Class

The `QuantumCircuit` class contains a large number of methods and attributes. The purpose of many of its methods is to apply quantum operations to a quantum circuit. Most of its other methods and attributes either manipulate~~,~~ or report information about~~,~~ a quantum circuit.

Commonly used gates

Table 1-1 contains some commonly used single-qubit gates and code examples. Variable `qc` refers to an instance of `QuantumCircuit` that contains at least four quantum wires:

Table 1-1. Commonly used single-qubit gates in Qiskit

Names	Example	Notes
H, Hadamard	`qc.h(0)`	Applies H gate to qubit 0. See "HGate" on page 153.
I, Identity	`qc.id(2)` or `qc.i(2)`	Applies I gate to qubit 2. See "IGate" on page 153.
P, Phase	`qc.p(math.pi/2,0)`	Applies P gate with π/2 phase rotation to qubit 0. See "PhaseGate" on page 154.
RX	`qc.rx(math.pi/4,2)`	Applies RX gate with π/4 rotation to qubit 2. See "RXGate" on page 154.
RY	`qc.ry(math.pi/8,0)`	Applies RY gate with π/8 rotation to qubit 0. See "RYGate" on page 154.
RZ	`qc.rz(math.pi/2,1)`	Applies RZ gate with π/2 rotation to qubit 1. See "RZGate" on page 155.
S	`qc.s(3)`	Applies S gate to qubit 3. Equivalent to P gate with π/2 phase rotation. See "SGate" on page 155.

Names	Example	Notes
S†	`qc.sdg(3)`	Applies S† gate to qubit 3. Equivalent to P gate with 3π/2 phase rotation. See "SdgGate" on page 155.
SX	`qc.sx(2)`	Applies SX (square root of X) gate to qubit 2. ~~This is~~ equivalent to ~~an~~ RX gate with ~~a~~ π/2 rotation. See "SXGate" on page 156.
T	`qc.t(1)`	Applies T gate to qubit 1. Equivalent to P gate with π/4 phase rotation. See "TGate" on page 156.
T†	`qc.tdg(1)`	Applies T† gate to qubit 1. Equivalent to P gate with 7π/4 phase rotation. See "TdgGate" on page 157.
U	`qc.u(math.pi/2,0,math.pi,1)`	Applies rotation with 3 Euler angles to qubit 1. See "UGate" on page 157.
X	`qc.x(3)`	Applies X gate to qubit 3. See "XGate" on page 157.
Y	`qc.y([0,2,3])`	Applies Y gates to qubits 0, 2 and 3. See "YGate" on page 157.
Z	`qc.z(2)`	Applies Z gate to qubit 2. Equivalent to P gate with π phase rotation. See "ZGate" on page 158.

Figure 1-1 contains a nonsensical circuit with all of the single-qubit gate examples from Table 1-1

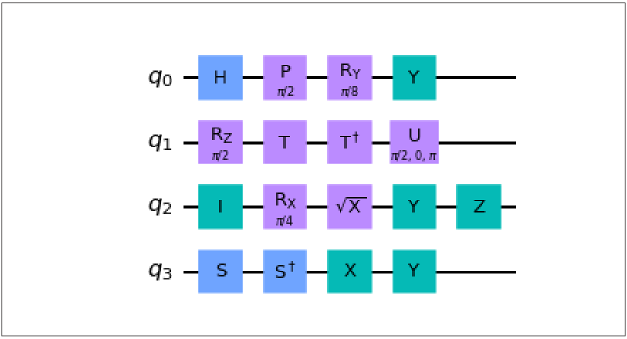

Figure 1-1. Nonsensical circuit with single-qubit gate examples

Table 1-2 contains some commonly used multi-qubit gates and code examples. Variable `qc` refers to an instance of `QuantumCircuit` that contains at least four quantum wires:

Table 1-2. Commonly used multi-qubit gates in Qiskit

Names	Example	Notes
CCX, Toffoli	`qc.ccx(0,1,2)`	Applies the X gate to quantum wire 2, subject to the state of the control qubits on wires 0 and 1. See "CCXGate" on page 159.
CH	`qc.ch(0,1)`	Applies the H gate to quantum wire 1, subject to the state of the control qubit on wire 0. See "CHGate" on page 159.
CP, Control-Phase	`qc.cp(math.pi/4,0,1)`	Applies the Phase gate to quantum wire 1, subject to the state of the control qubit on wire 0. See "CPhaseGate" on page 160.
CRX, Control-RX	`qc.crx(math.pi/2,2,3)`	Applies the RX gate to quantum wire 3, subject to the state of the control qubit on wire 2. See "CRXGate" on page 160.

Names	Example	Notes
CRY, Control-RY	`qc.cry(math.pi/8,2,3)`	Applies the RY gate to quantum wire 3, subject to the state of the control qubit on wire 2. See "CRYGate" on page 160.
CRZ	`qc.crz(math.pi/4,0,1)`	Applies the RZ gate to quantum wire 1, subject to the state of the control qubit on wire 0. See "CRZGate" on page 161.
CSwap, Fredkin	`qc.cswap(0,2,3)` or `qc.fredkin(0,2,3)`	Swaps the qubit states of wires 2 & 3, subject to the state of the control qubit on wire 0. See "CSwapGate" on page 161.
CSX	`qc.csx(0,1)`	Applies the SX (square root of X) gate to quantum wire 1, subject to the state of the control qubit on wire 0. See "CSXGate" on page 161.
CU	`qc.cu(math.pi/2,0,math.pi,0,0,1))`	Applies the U gate with an additional global phase argument to quantum wire 1, subject to the state of the control qubit on wire 0. See "CUGate" on page 161.
CX, CNOT	`qc.cx(2,3)` or `qc.cnot(2,3)`	Applies the X gate to quantum wire 3, subject to the state of the control qubit on wire 2. See "CXGate" on page 162.
CY, Control-Y	`qc.cy(2,3)`	Applies the Y gate to quantum wire 3, subject to the state of the control qubit on wire 2. See "CYGate" on page 162.
CZ, Control-Z	`qc.cz(1,2)`	Applies the Z gate to quantum wire 2, subject to the state of the control qubit on wire 1. See "CYGate" on page 162.

Names	Example	Notes
DCX	`qc.dcx(2,3)`	Applies two CNOT gates whose control qubits are on wires 2 and 3. See "DCXGate" on page 163.
iSwap	`qc.iswap(0,1)`	Swaps the qubit states of wires 0 ~~& 1~~, and changes the phase of the $\|01\rangle$ and $\|10\rangle$ amplitudes by i. See "iSwapGate" on page 163.
MCP, Multi-control Phase	`qc.mcp(math.pi/4, [0,1,2],3)`	Applies the Phase gate to quantum wire 3, subject to the state of the control qubits on wires 0, 1 and 2. See "MCPhaseGate" on page 163.
MCX, Multi-control X	`qc.mcx([0,1,2],3)`	Applies the X gate to quantum wire 3, subject to the state of the control qubits on wires 0, 1 and 2. See "MCXGate" on page 164.
Swap	`qc.swap(2,3)`	Swaps the qubit states of wires 2 ~~&~~ 3. See "SwapGate" on page 164.

Figure 1-2 contains a nonsensical circuit with all of the multi-qubit gate examples from Table 1-2.

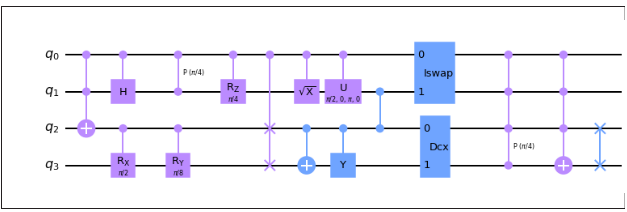

Figure 1-2. Nonsensical circuit with multi-qubit gate examples

Drawing a Quantum Circuit

The draw() method draws a quantum circuit in various formats.

Using the draw() method. The following code snippet uses the draw() method in the default format.

```
qc = QuantumCircuit(3)
qc.h(0)
qc.cx(0, 1)
qc.cx(0, 2)
qc.draw()
```

Figure 1-3 shows the drawn circuit.

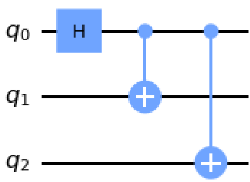

Figure 1-3. Example circuit visualization using the draw() method

Creating a Barrier

The barrier() method places a *barrier* on a circuit (shown in Figure 1-4), providing both visual and functional separation between gates on a quantum circuit. Gates on either side of a barrier are not candidates for being optimized together as the circuit is converted to run on quantum hardware or a simulator.

> **NOTE**
>
> The set of gates expressed using Qiskit represents an abstraction for the actual gates implemented on a given quantum computer or simulator. Qiskit *transpiles* the gates into those implemented on the target platform, combining gates where possible to optimize the circuit.

Using the barrier() method. The `barrier()` method takes as an optional argument the qubit wires on which to place a barrier. If no argument is supplied, a barrier is placed across all of the quantum wires. This method creates a `Barrier` instance (see “Barrier” on page 151).

The following code snippet demonstrates using the `barrier()` method with and without arguments.

```
qc = QuantumCircuit(2)
qc.h([0,1])
qc.barrier()
qc.x(0)
qc.x(0)
qc.s(1)
qc.barrier([1])
qc.s(1)
```

Figure 1-4 shows the resultant circuit.

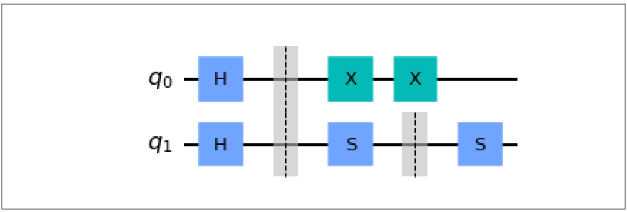

Figure 1-4. Example circuit using the barrier() method

Notice that the **S** gates in the circuit are separated by a barrier, and therefore not candidates to be combined into a **Z** gate. However, the **X** gates may be combined by removing both of them, as they cancel ~~each other~~.

Measuring a quantum circuit

The methods commonly used to measure quantum circuits are `measure()` and `measure_all()`. The former is useful when the quantum circuit contains classical wires on which to receive the result of a measurement. The latter is useful when the quantum circuit doesn't have any classical wires. These methods create `Measure` instances (see "Measure" on page 152).

Using the measure() method. The `measure()` method takes two arguments:

- the qubit wires to be measured
- the classical wires on which to store the resulting bits

This code snippet uses the `measure()` method, and Figure 1-5 shows a drawing of the resultant circuit.

```
qc = QuantumCircuit(3, 3)
qc.h([0,1,2])
qc.measure([0,1,2], [0,1,2])
qc.draw()
```

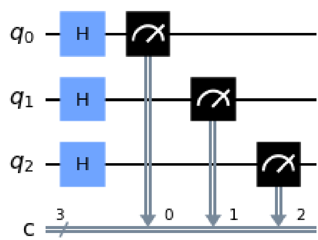

Figure 1-5. Example circuit using the measure() method

Notice that the `measure()` method appended the requested measurement operations to the circuit.

Using the measure_all() method. The `measure_all()` method may be called with no arguments. This code snippet uses the `measure_all()` method, and Figure 1-6 shows a drawing of the resultant circuit.

```
qc = QuantumCircuit(3)
qc.h([0,1,2])
qc.measure_all()
qc.draw()
```

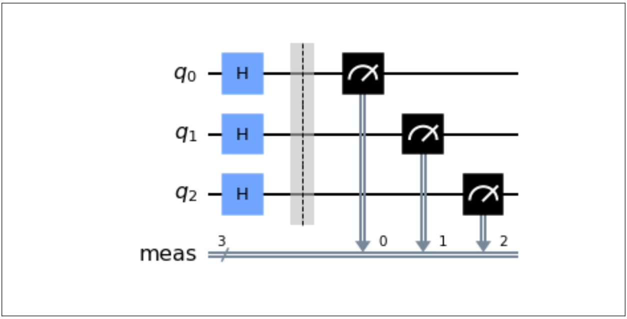

Figure 1-6. Example circuit using the measure_all() method

Notice that the `measure_all()` method created three classical wires, and added a barrier to the circuit, before appending the measurement operations.

Obtaining information about a quantum circuit

Methods commonly used to obtain information about a quantum circuit include `depth()`, `size()`, and `width()`. These are listed in Table 1-3. Note that variable `qc` refers to an instance of `QuantumCircuit`.

Table 1-3. Methods commonly used to obtain information about a quantum circuit

Names	Example	Notes
`depth`	`qc.depth()`	Returns the depth (critical path) of a circuit if directives such as barrier were removed
`size`	`qc.size()`	Returns the total number of gate operations in a circuit
`width`	`qc.width()`	Return the sum of qubits wires and classical wires in a circuit

Attributes commonly used to obtain information about a quantum circuit include `clbits`, `data`, `global_phase`, `num_clbits`, `num_qubits`, and `qubits`. These are listed in Table 1-4. Note that variable `qc` refers to an instance of `QuantumCircuit`.

Table 1-4. Attributes commonly used to obtain information about a quantum circuit

Names	Example	Notes
`clbits`	`qc.clbits`	Obtains the list of classical bits in the order that the registers were added
`data`	`qc.data`	Obtains a list of the operations (e.g., gates, barriers, and measurement operations) in the circuit
`global_phase`	`qc.global_phase`	Obtains the global phase of the circuit in radians
`num_clbits`	`qc.num_clbits`	Obtains the number of classical wires in the circuit
`num_qubits`	`qc.num_qubits`	Obtains the number of quantum wires in the circuit
`qubits`	`qc.qubits`	Obtains the list of quantum bits in the order that the registers were added

Manipulating a quantum circuit

Methods commonly used to manipulate quantum circuits include `append()`, `bind_parameters()`, `compose()`, `copy()`, `decompose()`, `from_qasm_file()`, `from_qasm_str()`, `initialize()`, `reset()`, `qasm()`, `to_gate()`, and `to_instruction()`.

Using the append() method. The `append()` method appends an instruction or gate to the end of the circuit on specified wires, modifying the circuit in place. The following code snippet uses the append() method, and Figure 1-7 shows a drawing of the resultant circuit.

```
qc = QuantumCircuit(2)
qc.h(1)
cx_gate = CXGate()
qc.append(cx_gate, [1,0])
qc.draw()
```

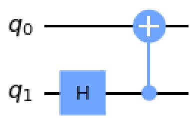

Figure 1-7. Example circuit resulting from the append() method

> **NOTE**
>
> The CXGate class (see "CXGate" on page 162) used in the code snippet is one of the gates defined in the qiskit.circuit.library package. The reader is advised to add the appropriate import statements to this and other code snippets contained in this book.

Using the bind_parameters() method. The bind_parameters() method binds parameters (see "Creating a Parameter Instance" on page 32) to a quantum circuit. The following code snippet creates a circuit in which there are three parameterized phase gates. Note that the arguments to the Parameter constructors in this code snippet are strings, in this case ones that contain theta characters. Figure 1-8 shows a drawing of the circuit.

```
theta1 = Parameter('θ1')
theta2 = Parameter('θ2')
```

```
theta3 = Parameter('θ3')

qc = QuantumCircuit(3)
qc.h([0,1,2])
qc.p(theta1,0)
qc.p(theta2,1)
qc.p(theta3,2)

qc.draw()
```

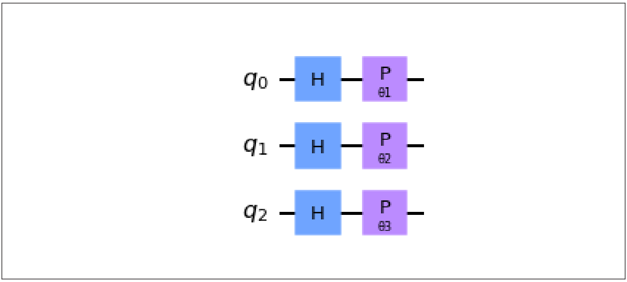

Figure 1-8. Example parameterized circuit

To bind the parameter values to a new circuit, we'll pass a dictionary that contains the parameter references and desired values to the `bind_parameters()` method. The following code snippet uses this technique, and Figure 1-9 shows the bound circuit in which the phase gate parameters are replaced with the supplied values.

```
b_qc = qc.bind_parameters({theta1: math.pi/8,
                           theta2: math.pi/4,
                           theta3: math.pi/2})

b_qc.draw()
```

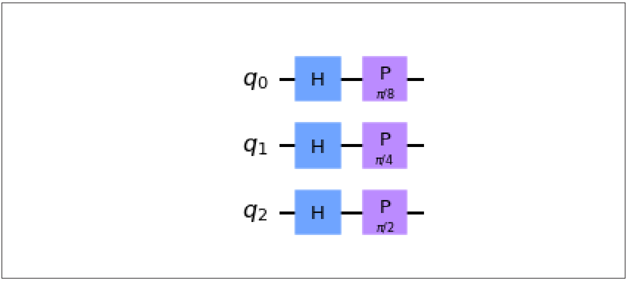

Figure 1-9. Example of bound circuit with the supplied phase gate rotation values

Using the compose() method. The `compose()` method returns a new circuit composed of the original and another circuit. The following code snippet uses the `compose()` method, and Figure 1-10 shows a drawing of the resultant circuit.

```
qc = QuantumCircuit(2,2)
qc.h(0)
another_qc = QuantumCircuit(2,2)
another_qc.cx(0,1)
bell_qc = qc.compose(another_qc)
bell_qc.draw()
```

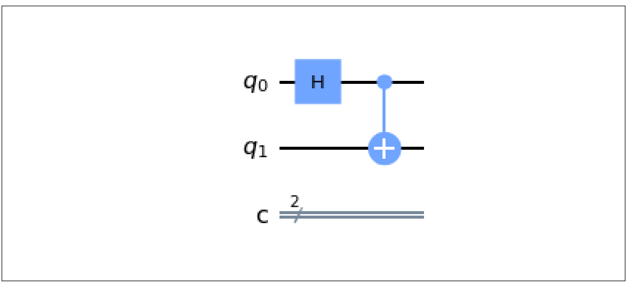

Figure 1-10. Example circuit resulting from the compose() method

Note that a circuit passed into the `compose()` method is allowed to have fewer quantum or classical wires than the original circuit.

Using the copy() method. The `copy()` method returns a copy of the original circuit. The following code snippet uses the `copy()` method:

```
qc = QuantumCircuit(3)
qc.h([0,1,2])
new_qc = qc.copy()
```

Using the decompose() method. The `decompose()` method returns a new circuit after having decomposed the original circuit one level. The following code snippet uses the `decompose()` method. Figure 1-11 shows a drawing of the resultant circuit in which **S**, **H**, and **X** gates are decomposed into the more fundamental **U** gate operations. See "UGate" on page 157:

```
qc = QuantumCircuit(2)
qc.h(0)
qc.s(0)
qc.x(1)
decomposed_qc = qc.decompose()
decomposed_qc.draw()
```

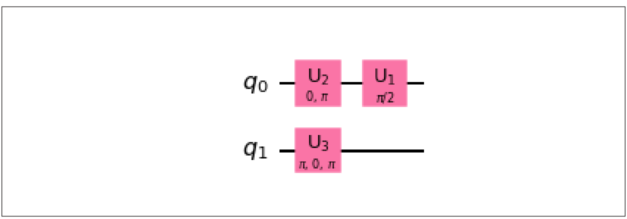

Figure 1-11. Example circuit resulting from the decompose() method

Using the from_qasm_file() method. The `from_qasm_file()` method returns a new circuit from a file that contains a quantum assembly-language (OpenQASM) program. The following code snippet uses the `from_qasm_file()` method:

```
new_qc = QuantumCircuit.from_qasm_file("file.qasm")
```

Using the from_qasm_str() method. The `from_qasm_str()` method returns a new circuit from a string that contains an OpenQASM program. The following code snippet uses the `from_qasm_str()` method, and Figure 1-12 shows a drawing of the resultant circuit:

```
qasm_str = """
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[2];
h q[0];
cx q[0],q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
"""
new_qc = QuantumCircuit.from_qasm_str(qasm_str)
new_qc.draw()
```

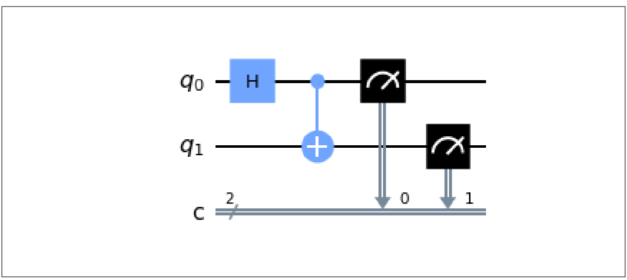

Figure 1-12. Example circuit resulting from the from_qasm_str() method

Using the initialize() method. The `initialize()` method initializes qubits of a quantum circuit to a given state, and is not a unitary operation. The following code snippet uses the `initialize()` method, and Figure 1-13 shows a drawing of the resultant circuit. In this code snippet, the circuit is initialized to the normalized statevector $|11\rangle$.

```
qc = QuantumCircuit(2)
qc.initialize([0, 0, 0, 1])
qc.draw()
```

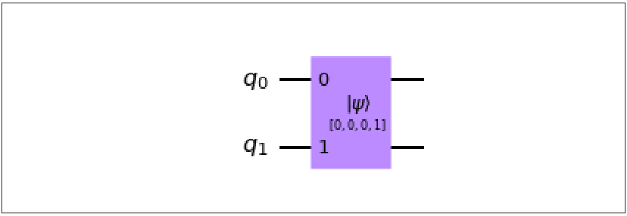

Figure 1-13. Example circuit resulting from the initialize() method

Using the reset() method. The `reset()` method resets a qubit in a quantum circuit to the $|0\rangle$ state~~,~~ and is not a unitary operation. The following code snippet uses the `reset()` method, and Figure 1-14 shows a drawing of the resultant circuit. Note that the qubit state is $|1\rangle$ before the reset operation. This method creates a `Reset` instance (see "Reset" on page 152).

```
qc = QuantumCircuit(1)
qc.x(0)
qc.reset(0)
qc.draw()
```

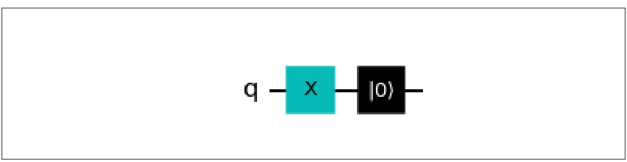

Figure 1-14. Example circuit resulting from ~~the~~ using the reset() method

Using the qasm() method. The `qasm()` method returns an OpenQASM program that represents the quantum circuit. The following code snippet uses the `qasm()` method, and Example 1-1 shows the resultant OpenQASM program.

```
qc = QuantumCircuit(2, 2)
qc.h(0)
qc.cx(0, 1)
qasm_str = qc.qasm()
print(qasm_str)
```

Example 1-1. OpenQASM program resulting from the using the qasm() method

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[2];
h q[0];
cx q[0],q[1];
```

Using the to_gate() method. The `to_gate()` method creates a custom *gate* (see "The Gate Class" on page 27) from a quantum circuit. The following code snippet creates a circuit that will be converted to a gate, and Figure 1-15 shows a drawing of the circuit.

NOTE

A *gate* represents a unitary operation. To create a custom operation that isn't unitary, use the `to_instruction()` method shown in "Using the to_instruction() method" on page 22.

```
anti_cnot_qc = QuantumCircuit(2)
anti_cnot_qc.x(0)
```

```
anti_cnot_qc.cx(0,1)
anti_cnot_qc.x(0)

anti_cnot_qc.draw()
```

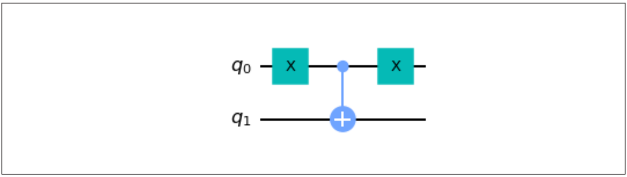

Figure 1-15. Example circuit that will be converted to a gate

This custom gate will implement an anti-control NOT gate in which the **X** gate is applied only when the control qubit is $|0\rangle$. The following code snippet creates a circuit that uses this custom gate, and Figure 1-16 shows a decomposed drawing of this circuit.

```
anti_cnot_gate = anti_cnot_qc.to_gate()

qc = QuantumCircuit(3)
qc.x([0,1,2])
qc.append(anti_cnot_gate, [0,2])

qc.decompose().draw()
```

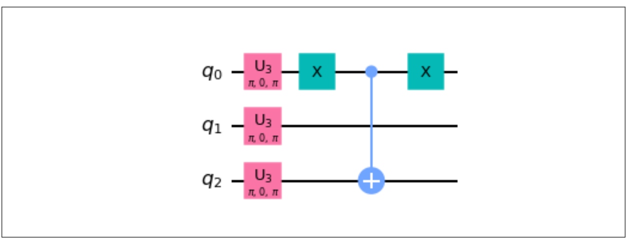

Figure 1-16. Decomposed circuit that uses a gate created by the to_gate() method

Using the to_instruction() method. The `to_instruction()` method creates a custom *instruction* (see "The Instruction Class" on page 26) from a quantum circuit. The following code snippet ~~of~~ creates a circuit that will be converted to an instruction, and Figure 1-17 shows a drawing of the circuit.

> **NOTE**
>
> An *instruction* represents an operation that isn't necessarily unitary. To create a custom operation that is unitary, use the `to_gate()` method shown in "Using the to_gate() method" on page 20.

```
reset_one_qc = QuantumCircuit(1)
reset_one_qc.reset(0)
reset_one_qc.x(0)

reset_one_qc.draw()
```

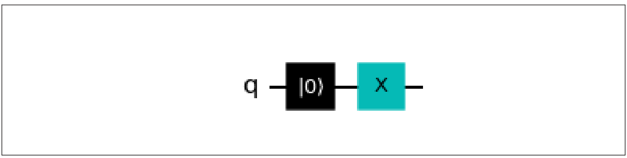

Figure 1-17. Example circuit that will be converted to an instruction

This custom instruction will reset a qubit and apply an **X** gate, in effect resetting the qubit to state $|1\rangle$. The following code snippet creates a circuit that uses this custom instruction, and Figure 1-18 shows a decomposed drawing of this circuit.

```
reset_one_inst = reset_one_qc.to_instruction()

qc = QuantumCircuit(2)
qc.h([0,1])
qc.reset(0)
qc.append(reset_one_inst, [1])
```

```
qc.decompose().draw()
```

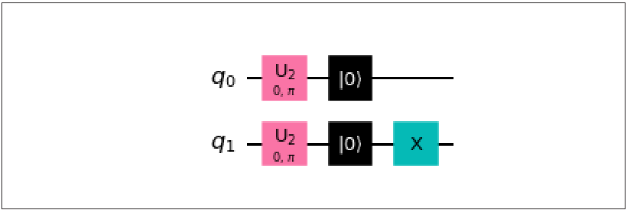

Figure 1-18. Circuit that uses an instruction created by the to_instruction() method

Saving state when running a circuit on AerSimulator

When running a circuit on an `AerSimulator` backend (see "Using the Aer Simulators" on page 41), simulator state may be saved in the circuit instance by using the `QuantumCircuit` methods in Table 1-5. Please note that these methods are available after obtaining an `AerSimulator` backend.

Table 1-5. Methods used to save simulator state in a circuit instance

Method name	Description
`save_state`	Saves the simulator state as appropriate for the simulation method
`save_density_matrix`	Saves the simulator state as a density matrix
`save_matrix_product_state`	Saves the simulator state as a matrix product state tensor
`save_stabilizer`	Saves the simulator state as a Clifford stabilizer
`save_statevector`	Saves the simulator state as a statevector
`save_superop`	Saves the simulator state as a superoperator matrix of the run circuit
`save_unitary`	Saves the simulator state as a unitary matrix of the run circuit

Using the QuantumRegister Class

It is sometimes useful to treat groups of quantum or classical wires as a unit. For example, the control qubits of the CNOT gates in the quantum circuit expressed in the following code snippet, as well as Figure 1-19, expect three qubits in equal superpositions. The additional quantum wire in the circuit is used as a scratch area whose output is disregarded.

```
qr = QuantumRegister(3, 'q')
scratch = QuantumRegister(1, 'scratch')
cr = ClassicalRegister(3, 'c')
qc = QuantumCircuit(qr, scratch, cr)

qc.h(qr)
qc.x(scratch)
qc.h(scratch)
qc.cx(qr[0], scratch)
qc.cx(qr[2], scratch)
qc.barrier(qr)
qc.h(qr)
qc.measure(qr, cr)

qc.draw()
```

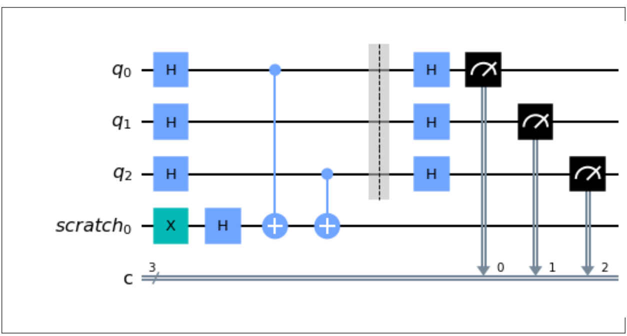

Figure 1-19. Example circuit using the QuantumRegister and ClassicalRegister classes

By defining a `QuantumRegister` consisting of three qubits, methods such as `h()`, `barrier()`, and `measure()` may be applied to all three wires by passing a `QuantumRegister` reference. Similarly, defining a `ClassicalRegister` (see "Using the ClassicalRegister Class" on page 25) consisting of three bits enables the `measure()` method to specify all three classical wires by passing a `ClassicalRegister` reference. Additionally, the names supplied to the `QuantumRegister` and `ClassicalRegister` constructors are displayed on the circuit drawing.

Some QuantumRegister attributes

Commonly used `QuantumRegister` attributes include `name`, and `size`. These are listed in Table 1-6. Note that variable `qr` refers to an instance of `QuantumRegister`.

Table 1-6. Some QuantumRegister attributes

Names	Example	Notes
`name`	`qr.name`	Obtains the name of the quantum register
`size`	`qr.size`	Obtains the number of qubit wires in the quantum register

Using the ClassicalRegister Class

Please refer to "Using the QuantumRegister Class" on page 24 for also motivating use of the `ClassicalRegister` class.

Some ClassicalRegister attributes

Commonly used `ClassicalRegister` attributes include `name`, and `size`. These are listed in Table 1-7. Note that variable `cr` refers to an instance of `ClassicalRegister`.

Table 1-7. Some ClassicalRegister attributes

Names	Example	Notes
`name`	`cr.name`	Obtains the name the classical register
`size`	`cr.size`	Obtains the number of bit wires in the classical register

Instructions and Gates

In Qiskit, all operations that may be applied to a quantum circuit are derived from the `Instruction` class (see "The Instruction Class" on page 26). Unitary operations are derived from the `Gate` class (see "The Gate Class" on page 27), which is a subclass of `Instruction`. Controlled-unitary operations are derived from the `ControlledGate` class (see "The ControlledGate Class" on page 28), which is a subclass of `Gate`. These classes may be used to define new instructions, unitary gates, and controlled-unitary gates, respectively.

The Instruction Class

The non-unitary operations in Qiskit (such as `Measure`, and `Reset`) are direct subclasses of `Instruction`. Although it is possible to define your own custom instructions by subclassing `Instruction`, another way is to use the `to_instruction()` method of the `QuantumCircuit` class (see an example of this in "Using the to_instruction() method" on page 22).

Methods in the `Instruction` class include `copy()`, `repeat()`, and `reverse_ops()`. These are listed in Table 1-8. Note that variable `inst` refers to an instance of `Instruction`.

Table 1-8. Commonly used methods in the Instruction class

Names	Example	Notes
`copy`	`inst.copy("My inst")`	Returns a copy of the instruction, giving the supplied name to the copy
`repeat`	`inst.repeat(2)`	Returns an instruction with this instruction repeated a given number of times
`reverse_ops`	`inst.reverse_ops()`	Returns an instruction with its operations in reverse order

Commonly used attributes in the `Instruction` class include `definition`, and `params`. These are listed in Table 1-9. Note that variable `inst` refers to an instance of `Instruction`.

Table 1-9. Commonly used attributes in the Instruction class

Names	Example	Notes
`definition`	`inst.definition`	Returns the definition in terms of basic gates
`params`	`inst.params`	Obtains the parameters to the instruction

The Gate Class

The unitary operations in Qiskit (such as `HGate`, and `XGate`) are subclasses of `Gate`. Although it is possible to define your own custom gates by subclassing `Gate`, another way is to use the `to_gate()` method of the `QuantumCircuit` class (see an example of this in "Using the to_gate() method" on page 20).

Commonly used methods in the `Gate` class include the `Instruction` methods listed in Table 1-8 as well as `control()`, `inverse()`, `power()`, and `to_matrix()`. These are all listed in Table 1-10. Note that variable `gate` refers to an instance of `Gate`.

Table 1-10. Commonly used methods in the Gate class

Names	Example	Notes
`control`	`gate.control(1)`	Given a number of control qubits, returns a controlled version of the gate
`copy`	`gate.copy("My gate")`	Returns a copy of the gate, giving the supplied name to the copy
`inverse`	`gate.inverse()`	Returns the inverse of the gate
`power`	`gate.power(2)`	Returns the gate raised to a given floating-point power.

Names	Example	Notes
repeat	gate.repeat(3)	Returns a gate with this gate repeated a given number of times
reverse_ops	gate.reverse_ops()	Returns a gate with its operations in reverse order
to_matrix	gate.to_matrix()	Returns an array for the gate's unitary matrix

Commonly used attributes in the `Gate` class include the `Instruction` attributes listed in Table 1-9 as well as `label`. These are all listed in Table 1-11. Note that variable `gate` refers to an instance of `Gate`.

Table 1-11. Commonly used attributes in the Gate class

Names	Example	Notes
definition	gate.definition	Returns the definition in terms of basic gates
label	gate.label	Obtains the label for the instruction
params	gate.params	Obtains the parameters to the instruction

The ControlledGate Class

The controlled-unitary operations in Qiskit (such as `CZGate`, and `CCXGate`) are subclasses of `ControlledGate`, which is a subclass of `Gate`.

Commonly used methods in the `ControlledGate` class are the `Gate` methods listed in Table 1-10

Commonly used attributes in the `ControlledGate` class include the `Gate` attributes listed in Table 1-11 as well as `num_ctrl_qubits`, and `ctrl_state`.

Using the num_ctrl_qubits attribute

The `num_ctrl_qubits` attribute holds an integer that represents the number of control qubits in a `ControlledGate`. The following code snippet, whose printed output would be 2, uses the `num_ctrl_qubits` attribute of a Toffoli gate.

```
toffoli = CCXGate()
print(toffoli.num_ctrl_qubits)
```

Using the ctrl_state() method

A `ControlledGate` may have one or more control qubits, each of which may actually be either control or *anti-control* qubits (see anti-control example in "Using the to_gate() method" on page 20). The `ctrl_state` attribute holds an integer whose binary value represents which qubits are control qubits, and which are anti-control qubits. Specifically, the binary digit `1` represents a control qubit, and the binary digit `0` represents an anti-control qubit. The `ctrl_state` attribute supports both accessing and modifying its value. The following code snippet uses the `ctrl_state` attribute in which the binary value `10` causes the topmost control qubit to be an anti-control qubit. Figure 1-20 shows a drawing of the resultant circuit.

```
toffoli = CCXGate()
toffoli.ctrl_state = 2

toffoli.definition.draw()
```

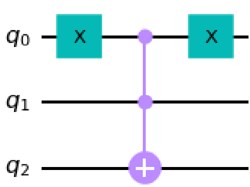

Figure 1-20. Toffoli gate with a control qubit and an anti-control qubit

Defining a custom controlled gate

Although it is possible to define your own custom controlled gates by subclassing `ControlledGate`, another way is to follow these two steps:

1. Create a custom gate with the `to_gate()` method of the `QuantumCircuit` class (see an example of this in "Using the to_gate() method" on page 20).
2. Add control qubits to your custom gate with the `control()` method shown in Table 1-10.

We'll follow those two steps to define a custom controlled gate that applies a π/16 phase rotation when both of its control qubits are $|1\rangle$. First, the following code snippet defines a circuit that contains a π/16 **P** gate and converts it to a custom gate, with Figure 1-21 showing a drawing of the custom gate.

```
p16_qc = QuantumCircuit(1)
p16_qc.p(math.pi/16, 0)

p16_gate = p16_qc.to_gate()

p16_gate.definition.draw()
```

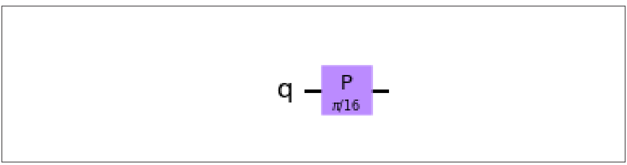

Figure 1-21. Custom π/16 phase gate drawing

Second, the following code snippet uses the `control()` method to create a `ControlledGate` from our custom gate, and Figure 1-22 shows a drawing of the custom controlled gate.

```
ctrl_p16 = p16_gate.control(2)

ctrl_p16.definition.draw()
```

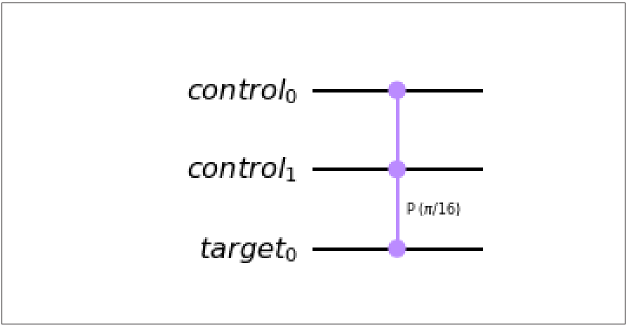

Figure 1-22. Custom controlled π/16 phase gate drawing

We'll leverage the `append()` method (see "Using the append() method" on page 13) in the following code snippet to use our custom controlled gate in a quantum circuit. Figure 1-23 shows a drawing of the circuit.

```
qc = QuantumCircuit(4)
qc.h([0,1,2,3])
qc.append(ctrl_p16,[0,1,3])

qc.decompose().draw()
```

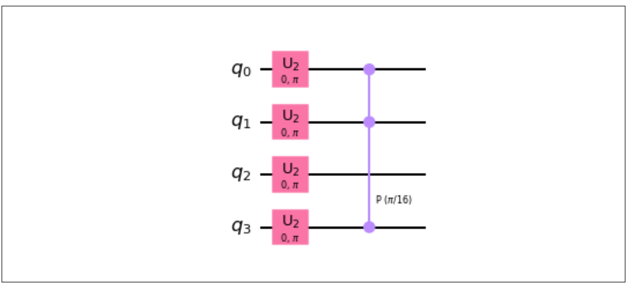

Figure 1-23. Decomposed circuit that uses the custom controlled gate

Parameterized Quantum Circuits

It is sometimes useful to create a quantum circuit in which values may be supplied at runtime. This capability is available in Qiskit using *parameterized circuits*, implemented in part by the `Parameter`~~,~~ and `ParameterVector` classes.

Creating a Parameter Instance

The `Parameter` class is used to represent a parameter in a quantum circuit. See “Using the bind_parameters() method” on page 14 for an example of defining and using a parameterized circuit. As shown in that example, a parameter may be created by supplying a unicode string to its constructor as follows:

```
theta1 = Parameter("θ1")
```

The `Parameter` object reference named `theta1` may subsequently be used in the `bind_parameters()`, or alternatively the `assign_parameters()`~~,~~ method of the `QuantumCircuit` class.

Using the ParameterVector Class

The `ParameterVector` class may be leveraged to create and use parameters as a collection instead of individual variables. The following code snippet creates a circuit in which there are three parameterized phase gates. Figure 1-24 shows a drawing of the circuit.

```
theta = ParameterVector('θ', 3)

qc = QuantumCircuit(3)
qc.h([0,1,2])
qc.p(theta[0],0)
qc.p(theta[1],1)
qc.p(theta[2],2)

qc.draw()
```

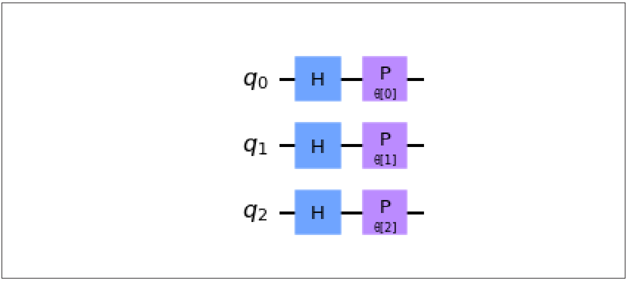

Figure 1-24. Example parameterized circuit leveraging ParameterVector

To bind the parameter values to a new circuit, we'll pass a dictionary that contains the `ParameterVector` reference and desired list of values to the `bind_parameters()` method. The following code snippet shows this technique, and Figure 1-25 shows the bound circuit in which the phase gate parameters are replaced with the supplied values.

```
b_qc = qc.bind_parameters({theta: [math.pi/8,
                                   math.pi/4,
                                   math.pi/2]})

b_qc.draw()
```

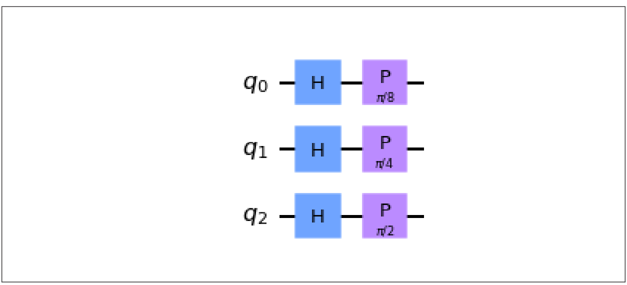

Figure 1-25. Example of bound circuit with the supplied phase gate rotation values

CHAPTER 2

Running Quantum Circuits

Qiskit supports running quantum circuits on a wide variety of quantum simulators and devices. We'll explore relevant classes and functions, most of which are located in the following modules:

- The `qiskit.providers.basicaer` module contains a basic set of simulators implemented in Python, often referred to as *BasicAer* simulators.
- The `qiskit.providers.aer` module contains a comprehensive set of high performance simulators, often referred to as *Aer* simulators.
- The `qiskit.providers` module contains classes that support these simulators as well as access to real quantum devices.

Regardless of the quantum simulator or device on which you choose to run a circuit, you may follow the steps ~~listed below~~:

1. Identify the appropriate *provider* (either `BasicAer`, `Aer`, or a quantum device provider). A provider's purpose is to get *backend* objects that enable executing circuits on a quantum simulator or device.

2. Obtain a reference to the desired *backend* from the provider. A backend provides the interface between Qiskit and the hardware or simulator that will execute circuits.
3. Using the backend, run the circuit on the simulator or device. This returns an object that represents the *job* in which the circuit is being run.
4. Interact with the job for purposes such as checking status, and getting its *result* after completing.

Using the BasicAer Simulators

As with any backend provider, a list of available `BasicAer` backends may be obtained by calling the provider's `backends()` method as shown in the following code snippet.

```
from qiskit import BasicAer

print(BasicAer.backends())


output:
  [<QasmSimulatorPy('qasm_simulator')>,
   <StatevectorSimulatorPy('statevector_simulator')>,
   <UnitarySimulatorPy('unitary_simulator')>]
```

Notice that the `output` shows a Python list containing three `BasicAer` backends, each of which represents a simulator implemented by a corresponding class. A reference to the desired backend may be obtained by calling the provider's `get_backend()` method as shown in the following code snippet. Notice that the desired backend in this example is the `qasm_simulator`, whose main purpose is to run a circuit and hold its measurement outcomes.

Using the BasicAer qasm_simulator Backend

```
from qiskit import QuantumCircuit,BasicAer,transpile
```

```
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all() ❶

backend = BasicAer.get_backend("qasm_simulator") ❷
tqc = transpile(qc, backend) ❸
job = backend.run(tqc, shots=1000) ❹
result = job.result() ❺
counts = result.get_counts(tqc) ❻
print(counts)


output: ❼
  {'00': 495, '11': 505}
```

Let's take a closer look at some relevant lines in the code snippet.

❶ The `BasicAer` `qasm_simulator` backend is useful for circuits that contain measurement instructions.

❷ A reference to the `qasm_simulator` backend (implemented by the `QasmSimulatorPy` class) is obtained.

❸ The circuit is *transpiled* with the `transpile()` function to use only gates available on the `BasicAer` `qasm_simulator`.

❹ The transpiled circuit and number of `shots` to perform is passed to the `run()` method of the `BasicAer` `qasm_simulator` backend. The `run()` method returns a `BasicAerJob` instance.

❺ The result of running the circuit (held in a `qiskit.Result` instance) is obtained with the `result()` method of the `BasicAerJob` instance.

❻ A Python dictionary containing the measurement outcomes per basis state is obtained with the `get_counts()` method of the `Result` instance.

❼ The measurement outcomes are printed in the output.

Next we'll take a look at the statevector_simulator in the code snippet ~~below~~, whose main purpose is to run a circuit and hold its resultant statevector.

Using the BasicAer statevector_simulator Backend

```
from qiskit import QuantumCircuit,BasicAer,transpile

qc = QuantumCircuit(2) ❶
qc.h(0)
qc.cx(0, 1)

backend = BasicAer.get_backend("statevector_simulator") ❷
tqc = transpile(qc, backend) ❸
job = backend.run(tqc) ❹
result = job.result() ❺
statevector = result.get_statevector(tqc, 4) ❻
print(statevector)


output: ❼
  [0.7071+0.j 0.+0.j 0.+0.j 0.7071+0.j]
```

Let's take a closer look at some relevant lines in the code snippet.

❶ Because measurement instructions collapse quantum states, the BasicAer statevector_simulator backend is most useful for circuits without measurement instructions. The statevector_simulator is "one-shot", so if there are measurements you could get a different statevector each time.

❷ A reference to the statevector_simulator backend (implemented by the StatevectorSimulatorPy class) is obtained.

❸ The circuit is *transpiled* with the `transpile()` function to use only gates available on the `BasicAer statevector_simulator`.

❹ The transpiled circuit is passed to the `run()` method of the `BasicAer statevector_simulator` backend. The `run()` method returns a `BasicAerJob` instance.

❺ The result of running the circuit (held in a `qiskit.Result` instance) is obtained with the `result()` method of the `BasicAerJob` instance.

❻ A list of complex probability amplitudes containing up to four decimal places that express a statevector is obtained with the `get_statevector()` method of the `Result` instance.

❼ The statevector is printed in the `output`.

To complete our tour of `BasicAer` backends, we'll take a look at the `unitary_simulator` in the code snippet ~~below~~, whose main purpose is to run a circuit and hold a unitary matrix that represents the circuit.

Using the BasicAer unitary_simulator Backend

```
from qiskit import QuantumCircuit,BasicAer,transpile

qc = QuantumCircuit(2) ❶
qc.h(0)
qc.cx(0, 1)

backend = BasicAer.get_backend("unitary_simulator") ❷
tqc = transpile(qc, backend) ❸
job = backend.run(tqc) ❹
result = job.result() ❺
unitary = result.get_unitary(tqc, 4) ❻
print(unitary)
```

```
output: ❼
  [[ 0.7071+0.0000j  0.7071-0.0000j
     0.0000+0.0000j  0.0000+0.0000j]
   [ 0.0000+0.0000j  0.0000+0.0000j
     0.7071+0.0000j -0.7071+0.0000j]
   [ 0.0000+0.0000j  0.0000+0.0000j
     0.7071+0.0000j  0.7071-0.0000j]
   [ 0.7071+0.0000j -0.7071+0.0000j
     0.0000+0.0000j  0.0000+0.0000j]]
```

Let's take a closer look at some relevant lines in the code snippet.

❶ The `BasicAer unitary_simulator` backend is ~~only~~ useful for circuits without measurement or reset instructions, as they are not supported by the `unitary_simulator`.

❷ A reference to the `unitary_simulator` backend (implemented by the `UnitarySimulatorPy` class) is obtained.

❸ The circuit is *transpiled* with the `transpile()` function to use only gates available on the `BasicAer unitary_simulator`.

❹ The transpiled circuit is passed to the `run()` method of the `BasicAer unitary_simulator` backend. The `run()` method returns a `BasicAerJob` instance.

❺ The result of running the circuit (held in a `qiskit.Result` instance) is obtained with the `result()` method of the `BasicAerJob` instance.

❻ A square matrix of complex numbers that express the circuit's unitary (transition amplitudes) is obtained with the `get_unitary()` method of the `Result` instance.

❼ The unitary matrix is printed in the `output`.

NOTE

As an alternative to calling the `run()` method of any of the simulator backends, you could call the `execute()` function. This function is located in the `qiskit.execute_function` module, and it relieves you of the responsibility of calling the `transpile()` function.

Now we'll turn our attention to the `qiskit.providers.aer` module, which contains a comprehensive set of high performance simulators often referred to as *Aer* simulators.

Using the Aer Simulators

As with any backend provider, a list of available `Aer` backends may be obtained by calling the provider's `backends()` method as shown in the following code snippet:

```
from qiskit import Aer

print(Aer.backends())


output:
  [AerSimulator('aer_simulator'),
   AerSimulator('aer_simulator_statevector'),
   AerSimulator('aer_simulator_density_matrix'),
   AerSimulator('aer_simulator_stabilizer'),
   AerSimulator('aer_simulator_matrix_product_state'),
   AerSimulator('aer_simulator_extended_stabilizer'),
   AerSimulator('aer_simulator_unitary'),
   AerSimulator('aer_simulator_superop'),
   QasmSimulator('qasm_simulator'),
   StatevectorSimulator('statevector_simulator'),
   UnitarySimulator('unitary_simulator'),
   PulseSimulator('pulse_simulator')]
```

We'll examine several of these `Aer` simulator backends, beginning with the three legacy simulators that resemble their `Basi` cAer counterparts. These legacy simulators are faster than the Python implemented BasicAer simulators, but have very similar APIs.

Using the Aer Legacy Simulators

The Aer provider has received greatly enhanced functionality with the introduction of the AerSimulator and PulseSimulator classes. In addition, three of the Aer legacy simulator backends remain. These backends are qasm_simulator, statevector_simulator, and unitary_simulator.

The code for using these Aer legacy simulators is nearly identical to the code for using their BasicAer counterparts. The only difference is that instead of using the BasicAer provider, you'd use the Aer provider. To try this out, run each of the code snippets in "Using the BasicAer Simulators" on page 36, substituting BasicAer with Aer.

Let's move on to the main simulator backend of the Aer provider, named AerSimulator.

Using the AerSimulator Backend

The AerSimulator backend is very versatile, offering many types of *simulation methods*, the default being automatic. The automatic simulation method allows the simulation method to be selected automatically based on the circuit and noise model.

Using the AerSimulator to hold measurement results

In the following code snippet the simulator will hold measurement results due to the presence of measurement instructions in the circuit.

```
from qiskit import QuantumCircuit,Aer,transpile

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all() ❶
```

```
backend = Aer.get_backend("aer_simulator") ❷
tqc = transpile(qc, backend) ❸
job = backend.run(tqc, shots=1000) ❹
result = job.result() ❺
counts = result.get_counts(tqc) ❻
print(counts)


output: ❼
  {'00': 516, '11': 484}
```

Let's take a closer look at some relevant lines in the code snippet.

❶ The `AerSimulator` with `automatic` simulation method will hold measurement results when measurement instructions are present.

❷ A reference to an `AerSimulator` backend with the `automatic` simulation method is obtained by passing `"aer_simulator"` into the `get_backend()` method.

❸ The circuit is *transpiled* with the `transpile()` function to use only gates available on this backend.

❹ The transpiled circuit and number of `shots` to perform is passed to the `run()` method of the backend. The `run()` method returns an `AerJob` instance.

❺ The result of running the circuit (held in a `qiskit.Result` instance) is obtained with the `result()` method of the `AerJob` instance.

❻ A Python dictionary containing the measurement outcomes per basis state is obtained with the `get_counts()` method of the `Result` instance.

❼ The measurement outcomes are printed in the `output`.

Next we'll use the AerSimulator as a statevector simulator.

Using the AerSimulator to calculate and hold a statevector

In the following code snippet the simulator will calculate and hold a statevector:

```
from qiskit import QuantumCircuit,Aer,transpile

qc = QuantumCircuit(2) ❶
qc.h(0)
qc.cx(0, 1)

backend = Aer.get_backend("aer_simulator") ❷
qc.save_statevector() ❸

tqc = transpile(qc, backend) ❹
job = backend.run(tqc) ❺
result = job.result() ❻
statevector = result.get_statevector(tqc, 4) ❼
print(statevector)


output: ❽
  [0.7071+0.j 0.+0.j 0.+0.j 0.7071+0.j]
```

Let's take a closer look at some relevant lines in the code snippet:

❶ Because measurement instructions collapse quantum states, AerSimulator statevector simulator functionality is most useful for circuits without measurement instructions.

❷ A reference to an AerSimulator backend with the automatic simulation method is obtained by passing "aer_simulator" into the get_backend() method.

❸ The save_statevector() method saves the current simulator quantum state as a statevector. See "Saving state when running a circuit on AerSimulator" on page 23 for other methods that save simulator state in a quantum circuit.

❹ The circuit is *transpiled* with the `transpile()` function to use only gates available on this backend.

❺ The transpiled circuit is passed to the `run()` method of the backend. The `run()` method returns an `AerJob` instance.

❻ The result of running the circuit (held in a `qiskit.Result` instance) is obtained with the `result()` method of the `AerJob` instance.

❼ A list of complex probability amplitudes that express the saved statevector is obtained with the `get_statevector()` method of the `Result` instance.

❽ The statevector is printed in the `output`.

Now we'll use the `AerSimulator` as a unitary simulator.

Using the AerSimulator to calculate and hold a unitary

In the following code snippet the simulator will calculate and hold a circuit's unitary.

```
from qiskit import QuantumCircuit,Aer,transpile

qc = QuantumCircuit(2) ❶
qc.h(0)
qc.cx(0, 1)

backend = Aer.get_backend("aer_simulator") ❷
qc.save_unitary() ❸

tqc = transpile(qc, backend) ❹
job = backend.run(tqc) ❺
result = job.result() ❻
unitary = result.get_unitary(qc, 4) ❼
print(unitary)
```

```
output: ❽
  [[ 0.7071+0.0000j  0.7071-0.0000j
     0.0000+0.0000j  0.0000+0.0000j]
   [ 0.0000+0.0000j  0.0000+0.0000j
     0.7071+0.0000j -0.7071+0.0000j]
   [ 0.0000+0.0000j  0.0000+0.0000j
     0.7071+0.0000j  0.7071-0.0000j]
   [ 0.7071+0.0000j -0.7071+0.0000j
     0.0000+0.0000j  0.0000+0.0000j]]
```

Let's take a closer look at some relevant lines in the code snippet.

❶ `AerSimulator` unitary simulator functionality is only useful for circuits without measurement or reset instructions.

❷ A reference to an `AerSimulator` backend with the `automatic` simulation method is obtained by passing `"aer_simulator"` into the `get_backend()` method.

❸ The `save_unitary()` method saves the circuit's unitary matrix. See "Saving state when running a circuit on AerSimulator" on page 23 for other methods that save simulator state in a quantum circuit.

❹ The circuit is *transpiled* with the `transpile()` function to use only gates available on this backend.

❺ The transpiled circuit is passed to the `run()` method of the backend. The `run()` method returns an `AerJob` instance.

❻ The result of running the circuit (held in a `qiskit.Result` instance) is obtained with the `result()` method of the `AerJob` instance.

❼ A square matrix of complex numbers that express the saved unitary is obtained with the `get_unitary()` method of the `Result` instance.

8 The unitary matrix is printed in the `output`.

Now we'll discuss using the `AerSimulator` for additional simulation methods.

Using the AerSimulator for additional simulation methods

So far we've examined examples of using the `AerSimulator` backend with the `automatic` simulation method to run a circuit and ~~either~~ hold its measurement results, statevector, or unitary matrix. The `AerSimulator` backend is capable of additional simulation methods, automatically selecting them based on the circuit and noise model. Simulation methods may also be set explicitly in one of the following ways:

Using set_options() to update the simulation method. The simulation method for an `AerSimulator` backend may be explicitly updated by calling `set_options()`, passing the desired simulation method from Table 2-1 as a keyword argument. For example, the following code snippet may be used to update an `AerSimulator` backend to use the `density_matrix` simulation method:

```
backend = Aer.get_backend("aer_simulator")
backend.set_options(method="density_matrix")
```

Getting a backend with a pre-configured simulation method. Each of the `Aer` simulation methods has a corresponding string that may be passed into the `get_backend()` method. These strings are output in the first code snippet of "Using the Aer Simulators" on page 41 and may be formed by appending a simulation method from Table 2-1 to `"aer_simulator_"`. For example, the following code snippet may be used to get an `AerSimulator` backend pre-configured with the `density_matrix` simulation method:

```
Aer.get_backend("aer_simulator_density_matrix")
```

Passing a simulation method into run(). The simulation method for an `AerSimulator` backend may be explicitly overridden for a single execution. This is achieved by passing the desired simulation method from Table 2-1 as a keyword argument into the `run()` method. For example, the following code snippet, in which `tqc` is a transpiled circuit, may be used to override the simulation method of an `AerSimulator` backend to use the `density_matrix` simulation method:

```
backend = Aer.get_backend("aer_simulator")
backend.run(tqc, method="density_matrix")
```

Table 2-1 contains a list of the `AerSimulator` simulation methods.

Table 2-1. `AerSimulator` simulation methods

Name	Description
`automatic`	Default simulation method that selects the simulation method automatically based on the circuit and noise model.
`density_matrix`	Density matrix simulation that may sample measurement outcomes from noisy circuits with all measurements at end of the circuit.
`extended_stabilizer`	An approximate simulation for Clifford + T circuits based on a state decomposition into ranked-stabilizer state.
`matrix_product_state`	A tensor-network statevector simulator that uses a Matrix Product State (MPS) representation for the state.
`stabilizer`	An efficient Clifford stabilizer state simulator that can simulate noisy Clifford circuits if all errors in the noise model are also Clifford errors.
`statevector`	Statevector simulation that can sample measurement outcomes from ideal circuits with all measurements at end of the circuit. For noisy simulations each shot samples a randomly sampled noisy circuit from the noise model.

Name	Description
`superop`	Superoperator matrix simulation of an ideal or noisy circuit. This simulates the superoperator matrix of the circuit itself rather than the evolution of an initial quantum state.
`unitary`	Unitary matrix simulation of an ideal circuit. This simulates the unitary matrix of the circuit itself rather than the evolution of an initial quantum state.

Notice that some of the simulation method descriptions in Table 2-1 mention simulating noisy circuits. The `AerSimulator` supports this by allowing a noise model to be supplied that expresses error characteristics of a real or hypothetical quantum device.

Supplying a noise model to an AerSimulator backend

In the following code snippet a simple custom noise model is created and supplied to an `AerSimulator` backend:

```
from qiskit import QuantumCircuit, Aer, transpile
from qiskit.providers.aer.noise import \
        NoiseModel, depolarizing_error

err_1 = depolarizing_error(0.95, 1) ❶
err_2 = depolarizing_error(0.01, 2)
noise_model = NoiseModel()
noise_model.add_all_qubit_quantum_error(err_1,
                                         ['u1', 'u2', 'u3'])
noise_model.add_all_qubit_quantum_error(err_2,
                                                      ['cx'])

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all()

backend = Aer.get_backend("aer_simulator")
backend.set_options(noise_model=noise_model) ❷
tqc = transpile(qc, backend)
```

```
job = backend.run(tqc, shots=1000)
result = job.result()
counts = result.get_counts(tqc)
print(counts)

output: ❸
  {'00': 508, '01': 3, '10': 3, '11': 486}
```

Let's take a closer look at some relevant lines in the code snippet:

❶ Build a simple noise model using classes and functions from the `qiskit.providers.aer.noise` module.

❷ Supply the noise model to the `AerSimulator` backend with its `set_options` method.

❸ Measurement outcomes printed in the `output` reflect the circuit noise.

This example supplied a noise model to an `AerSimulator` backend. In the next section, we'll create an `AerSimulator` backend from the characteristics of a real quantum device.

Creating an AerSimulator backend from a real device

To simulate a real quantum device, mimicking its configuration and noise model, you may use the `from_backend()` method of the `AerSimulator` class as shown in the following code snippet:

```
from qiskit import QuantumCircuit, transpile
from qiskit.providers.aer import AerSimulator
from qiskit.test.mock import FakeVigo

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all()

device_backend = FakeVigo() ❶
backend = AerSimulator.from_backend(device_backend) ❷
```

```
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
counts = result.get_counts(tqc)
print(counts)

output: ❸
  {'00': 494, '01': 49, '10': 39, '11': 418}
```

Let's take a closer look at some relevant lines in the code snippet.

❶ Because available real hardware devices are continually updating, we're using device configuration and noise data that exists in a Qiskit library for this example. To obtain a device backend from a quantum device provider, you could use the following code snippet, where `provider` is a reference to the provider, and `device` is the name of the device:

```
device_backend = provider.get_backend("device")
```

❷ An `AerSimulator` backend is created using the supplied device backend.

❸ The measurement outcomes are printed in the `output`.

Monitoring Job Status and Obtaining Results

When running a quantum circuit, a reference to a job (currently a subclass of `qiskit.providers.JobV1`) is returned. This job reference may be used to monitor its status as well as to obtain a reference to a `qiskit.result.Result` instance. This `Result` reference may be used to obtain relevant results data from the experiment. Table 2-2, Table 2-3, and Table 2-4 describe some of the commonly used methods and attributes in these classes.

Table 2-2. Commonly used `qiskit.providers.JobV1` methods

Method name	Description
`job_id`	Returns a unique identifier for this job.
`backend`	Returns a reference to a subclass of `qiskit.providers.BackendV1` used for this job.
`status`	Returns the status of this job, for example `JobStatus.QUEUED`, `JobStatus.RUNNING`, or `JobStatus.DONE`.
`cancel`	Makes an attempt to cancel the job.
`cancelled`	Returns a boolean that indicates whether the job has been cancelled.
`running`	Returns a boolean that indicates whether the job is actively running on the quantum simulator or device.
`done`	Returns a boolean that indicates whether the job has successfully run.
`in_final_state`	Returns a boolean that indicates whether the job has finished. If so, it is in one of the final states: `JobStatus.CANCELLED`, `JobStatus.DONE`, or `JobStatus.ERROR`.
`wait_for_final_state`	Polls the job status for a given duration at a given interval, calling an optional callback method. Returns when the job is in one of the final states, or the given duration has expired.
`result`	Returns an instance of `qiskit.result.Result` that holds relevant results data from the experiment.

NOTE

Methods in Table 2-2 could be leveraged to create a job monitoring facility. There is already a basic job monitoring facility in the `qiskit.tools` package, implemented in the `job_monitor` function.

Table 2-3. Commonly used `qiskit.result.Result` methods

Method name	Description
`get_counts`	Returns a dictionary containing the count of measurement outcomes per basis state, if available.
`get_memory`	Returns a list containing a basis state resulting from each shot, if available. Requires that the `memory` option is `True`.
`get_statevector`	Returns a list of complex probability amplitudes that express a saved statevector, if available.
`get_unitary`	Return a unitary matrix of complex numbers that represents the circuit, if available.
`data`	Returns the raw data for an experiment.
`to_dict`	Returns a dictionary representation of the `results` attribute (see Table 2-4).

Table 2-4. Commonly used `qiskit.result.Result` attributes

Attribute name	Description
`backend_name`	Holds the name of the backend quantum simulator or device.
`backend_version`	Holds the version of the backend quantum simulator or device.
`job_id`	Holds a unique identifier for the job that produced this result.
`results`	List containing results of experiments run. Note that all of our examples run just one circuit at a time.
`success`	Indicates whether experiments ran successfully.

CHAPTER 3
Visualizing Quantum Measurements and States

In addition to drawing circuits (see "Drawing a Quantum Circuit" on page 8), Qiskit provides visualizations for data such as measurement counts~~,~~ and quantum states.

Visualizing Measurement Counts

To visualize experiments that result in measurement counts, Qiskit contains the `plot_histogram()` function.

Using the plot_histogram Function

The `plot_histogram()` function takes a dictionary containing measurement counts and plots them in a bar graph with one bar per basis state. We'll demonstrate this function in Example 3-1 by plotting the measurement counts from the example in "Using the AerSimulator to hold measurement results" on page 42.

Example 3-1. Using the `plot_histogram()` function to plot measurement counts

```
from qiskit import QuantumCircuit,Aer,transpile

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all()

backend = Aer.get_backend("aer_simulator")
tqc = transpile(qc, backend)
job = backend.run(tqc, shots=1000)
result = job.result()
counts = result.get_counts(tqc)

plot_histogram(counts)
```

Figure 3-1 shows a bar graph with the counts expressed as probabilities.

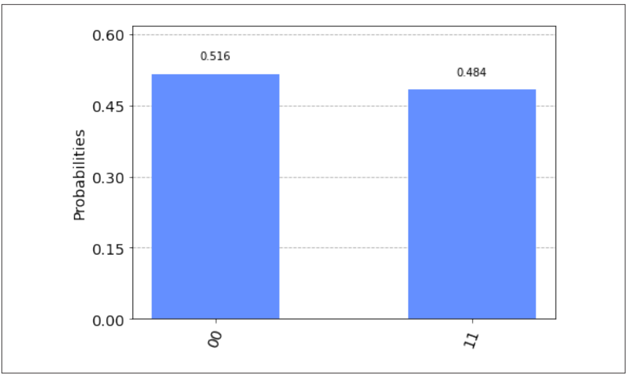

Figure 3-1. Example bar graph using the `plot_histogram()` function

Table 3-1 contains a list of commonly used `plot_histogram` parameters. This is implemented with `matplotlib`, which uses parameters shown in the table such as `figsize`.

Table 3-1. Commonly used `plot_histogram` parameters

Param name	Description
`data`	Dictionary, or list of dictionaries, containing measurement counts.
`figsize`	Tuple containing figure size in inches.
`color`	String or list of strings for bar colors.
`legend`	A list of strings to label the data. The number of entries must match the number of dictionaries in the data parameter.
`bar_labels`	Boolean that causes each bar to be labeled with probability values.
`title`	A string to label the plot title.

Visualizing Quantum States

Qiskit contains several functions for visualizing statevectors and density matrices, including `plot_state_qsphere()`, `plot_state_city()`, `plot_bloch_multivector()`, `plot_state_hinton()`, and `plot_state_paulivec()`.

Using the plot_state_qsphere Function

The `plot_state_qsphere()` function takes a statevector or density matrix and represents it on a *Q-sphere*. Often confused with a Bloch sphere, the Q-sphere is great for visualizing multi-qubit quantum states. We'll demonstrate this function in Example 3-2 by plotting the statevector from a circuit containing a *Quantum Fourier Transform* (QFT).

Example 3-2. Using the `plot_state_qsphere()` function to plot a statevector

```
from qiskit import QuantumCircuit,Aer,transpile
from qiskit.visualization import plot_state_qsphere
from math import pi
```

```
backend = Aer.get_backend("aer_simulator_statevector")

qc = QuantumCircuit(3)
qc.rx(pi, 0)
qc.ry(pi/8, 2)
qc.swap(0, 2)
qc.h(0)
qc.cp(pi/2,0, 1)
qc.cp(pi/4, 0, 2)
qc.h(1)
qc.cp(pi/2, 1, 2)
qc.h(2)
qc.save_statevector()

tqc = transpile(qc, backend)
job = backend.run(tqc)
result = job.result()
statevector = result.get_statevector()

plot_state_qsphere(statevector)
```

Figure 3-2 shows a Q-sphere with a point for each basis state. The size of each point is proportional to its measurement probability, and a point's color corresponds to its phase. Notice that the basis states are placed on the poles and latitude lines of the sphere according to their *Hamming weights* (number of ones in their basis states), starting from all-zeros at the top, and all-ones at the bottom.

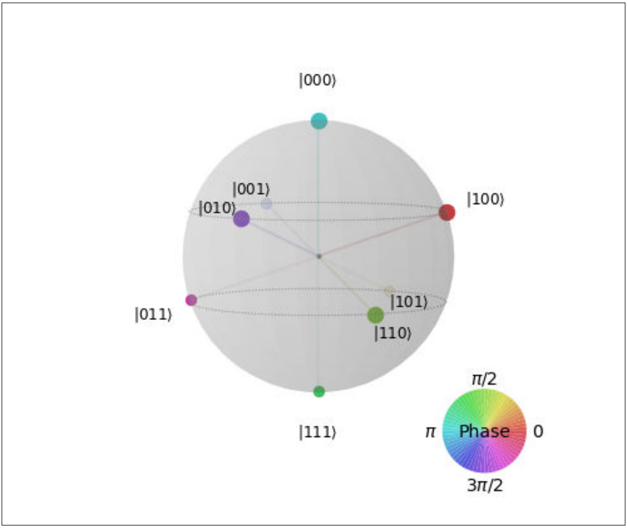

Figure 3-2. Example Q-sphere produced by the `plot_state_qsphere()` *function*

Table 3-2 contains a list of commonly used `plot_state_qsphere` parameters.

Table 3-2. Commonly used `plot_state_qsphere` *parameters*

Param name	Description
`state`	`Statevector`, `DensityMatrix`, or `ndarray` (a ~~numpy~~ type) containing complex numbers that represents a pure or mixed quantum state.
`figsize`	Tuple containing figure size in inches.
`show_state_labels`	Boolean indicating whether to show labels for each basis state.
`show_state_phases`	Boolean indicating whether to show the phase for each basis state.
`use_degrees`	Boolean indicating whether to use degrees for the phase values in the plot.

Using the plot_state_city Function

The `plot_state_city()` function takes a statevector or density matrix and represents it on a pair of three-dimensional bar graphs also known as a *cityscape*. We'll demonstrate this function in Example 3-3 by plotting the density matrix from the *mixed state* example in "Using the DensityMatrix Class" on page 91.

Example 3-3. Using the `plot_state_city()` function to plot a density matrix for a mixed state

```
from qiskit.quantum_info import DensityMatrix, \
                                   Operator
from qiskit.visualization import plot_state_city

dens_mat = 0.5*DensityMatrix.from_label('11') + \
  0.5*DensityMatrix.from_label('+0')
tt_op = Operator.from_label('TT')
dens_mat = dens_mat.evolve(tt_op)

plot_state_city(dens_mat)
```

Figure 3-3 shows a pair of 3D bar graphs that represent the complex numbers in the density matrix. The left 3D bar graph represents the real parts, and the right 3D bar graph represents the imaginary parts. See "Using the DensityMatrix Class" on page 91 for a discussion on the *mixed state* in this example.

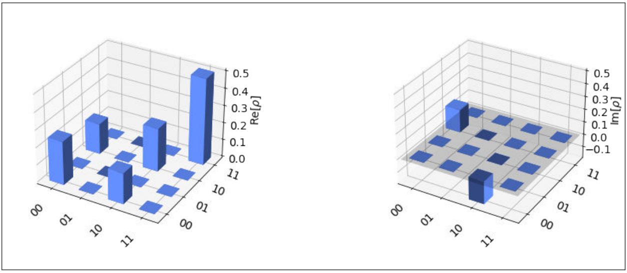

Figure 3-3. 3D bar graphs produced by the `plot_state_city()` function

Table 3-3 contains a list of commonly used `plot_state_city` parameters.

Table 3-3. Commonly used `plot_state_city` parameters

Param name	Description
`state`	`Statevector`, `DensityMatrix`, or `ndarray` (a ~~numpy~~ type) containing complex numbers that represents a pure or mixed quantum state.
`title`	A string to label the plot title.
`figsize`	Tuple containing figure size in inches.
`color`	List with two elements that contain colors for the real and imaginary 3D bars.
`alpha`	Float containing desired transparency for the 3D bars.

Using the plot_bloch_multivector Function

The `plot_bloch_multivector()` function takes a statevector or density matrix and represents it on one or more Bloch spheres. We'll demonstrate this function in Example 3-4 by plotting the statevector from a circuit containing a *Quantum Fourier Transform* (QFT).

Example 3-4. Using the `plot_state_qsphere()` function to plot a statevector

```
from qiskit import QuantumCircuit,Aer,transpile
from qiskit.visualization import plot_bloch_multivector
from math import pi

backend = Aer.get_backend("aer_simulator_statevector")

qc = QuantumCircuit(3)
qc.rx(pi, 0)
qc.ry(pi/8, 2)
qc.swap(0, 2)
qc.h(0)
qc.cp(pi/2,0, 1)
qc.cp(pi/4, 0, 2)
qc.h(1)
qc.cp(pi/2, 1, 2)
qc.h(2)
qc.save_statevector()

tqc = transpile(qc, backend)
job = backend.run(tqc)
result = job.result()
statevector = result.get_statevector()

plot_bloch_multivector(statevector)
```

Figure 3-4 shows one Bloch sphere for each qubit in a quantum state. Note that the arrow doesn't always reach the surface of the Bloch sphere, namely in cases where qubits are entangled or the state is a *mixed state* (see the mixed state example in "Using the DensityMatrix Class" on page 91).

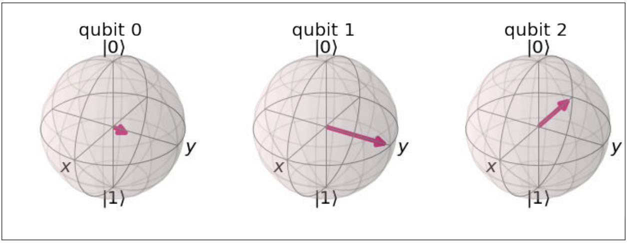

Figure 3-4. Example Bloch spheres produced by the `plot_bloch_multivector()` function

Table 3-4 contains a list of commonly used `plot_bloch_multivector` parameters.

Table 3-4. Commonly used `plot_bloch_multivector` parameters

Param name	Description
`state`	`Statevector`, `DensityMatrix`, or `ndarray` (a ~~numpy~~ type) containing complex numbers that represents a pure or mixed quantum state.
`title`	A string to label the plot title.
`figsize`	Tuple containing figure size in inches.
`reverse_bits`	Boolean indicating whether to show the most significant Bloch sphere on the left.

Using the plot_state_hinton Function

The `plot_state_hinton()` function takes a statevector or density matrix and represents it on a *Hinton diagram*. We'll demonstrate this function in Example 3-5 by plotting the density matrix from the *mixed state* example in "Using the DensityMatrix Class" on page 91.

Example 3-5. Using the `plot_state_hinton()` function to plot a density matrix for a mixed state

```
from qiskit.quantum_info import DensityMatrix, \
                                             Operator
from qiskit.visualization import plot_state_hinton

dens_mat = 0.5*DensityMatrix.from_label('11') + \
  0.5*DensityMatrix.from_label('+0')
tt_op = Operator.from_label('TT')
dens_mat = dens_mat.evolve(tt_op)

plot_state_hinton(dens_mat)
```

Figure 3-5 shows a Hinton diagram that represents the complex numbers in the density matrix. The left half represents the real parts, and the right half represents the imaginary parts. See "Using the DensityMatrix Class" on page 91 for a discussion on the *mixed state* in this example.

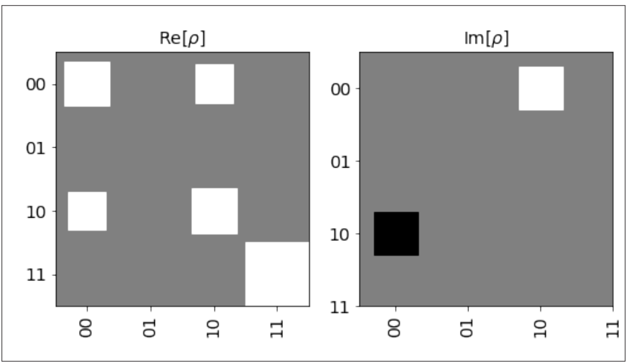

Figure 3-5. Hinton diagram produced by the `plot_state_hinton()` function

Table 3-5 contains a list of commonly used `plot_state_hinton` parameters.

Table 3-5. Commonly used `plot_state_hinton` parameters

Param name	Description
`state`	`Statevector`, `DensityMatrix`, or `ndarray` (a ~~numpy~~ type) containing complex numbers that represents a pure or mixed quantum state~~.~~
`title`	A string to label the plot title~~.~~
`figsize`	Tuple containing figure size in inches~~.~~

Using the plot_state_paulivec Function

The `plot_state_paulivec()` function takes a statevector or density matrix and represents it as a sparse bar graph of *expectation values* over the Pauli matrices. We'll demonstrate this function in Example 3-6 by representing the density matrix from the *mixed state* example in "Using the DensityMatrix Class" on page 91.

Example 3-6. Using the `plot_state_paulivec()` function to represent a density matrix for a mixed state

```
from qiskit.quantum_info import DensityMatrix, \
                                        Operator
from qiskit.visualization import plot_state_paulivec

dens_mat = 0.5*DensityMatrix.from_label('11') + \
  0.5*DensityMatrix.from_label('+0')
tt_op = Operator.from_label('TT')
dens_mat = dens_mat.evolve(tt_op)

plot_state_paulivec(dens_mat)
```

Figure 3-6 shows a sparse bar graph that represents the density matrix as expectation values over the Pauli matrices.

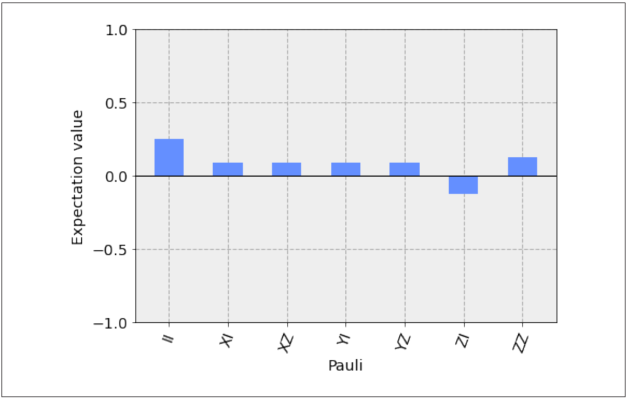

Figure 3-6. Bar graph produced by the `plot_state_paulivec()` function

Table 3-6 contains a list of commonly used `plot_state_paulivec` parameters.

Table 3-6. Commonly used `plot_state_paulivec` parameters

Param name	Description
`state`	Statevector, `DensityMatrix`, or ndarray (a ~~numpy~~ type) containing complex numbers that represents a pure or mixed quantum state.
`title`	A string to label the plot title.
`figsize`	Tuple containing figure size in inches.
`color`	String or list of strings for the expectation value bar colors.

CHAPTER 4

Using the Transpiler

We've been using the `QuantumCircuit` class to represent quantum programs, and the purpose of quantum programs is to run them on real devices and get results from them. When programming, we usually don't worry about the device-specific details, and instead use high-level operations. But most devices (and some simulators) can ~~only~~ carry out a small set of operations, and can ~~only~~ perform multi-qubit gates between certain qubits. This means we need to *transpile* our circuit for the specific device we're running on.

The transpilation process involves converting the operations in the circuit to those supported by the device, and swapping qubits (via swap gates) within the circuit to overcome limited qubit connectivity. Qiskit's transpiler does this job, as well as some optimization to reduce the circuit's gate count where it can.

Quickstart with Transpile

In this section, we'll show how to use the transpiler to get your circuit device-ready. We'll give a brief overview of the transpiler's logic and how we can get the best results from it.

The only required argument for `transpile` is the `QuantumCircuit` we want to transpile, but if we want transpile to do something interesting, we'll need to tell it what we want it to do. The easiest way to get your circuit running on a device is to simply pass `transpile` the backend object and let it grab the properties it needs. Transpile returns a new `QuantumCircuit` object compatible with the backend. The following code snippet shows what the simplest usage of the transpiler looks like:

```
from qiskit import transpile
transpiled_circuit = transpile(circuit, backend)
```

For example, the following code creates a simple `QuantumCircuit` with one qubit, a `YGate` and two `CXGate` s:

```
from qiskit import QuantumCircuit
qc = QuantumCircuit(3)
qc.y(0)
for t in range(2): qc.cx(0,t+1)
qc.draw()
```

Figure 4-1 shows the output of `qc.draw()`.

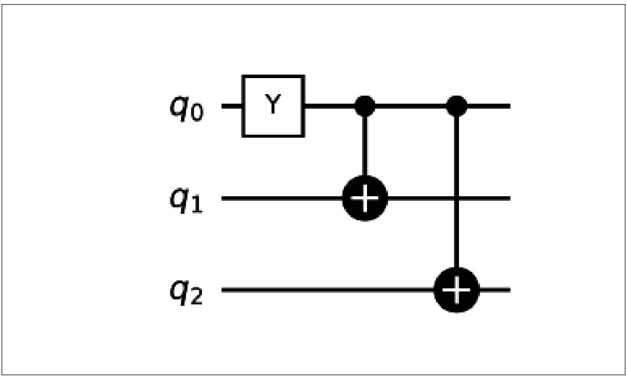

Figure 4-1. Simple circuit with a YGate and two CXGates

In the next code snippet, we decide we want to run `qc` on the mock backend `FakeSantiago` (a mock backend contains properties and noise models of a real system, and uses the `Aer`

Simulator to simulate that system). We can see in the output (shown under the code) that FakeSantiago doesn't understand the YGate operation:

```
from qiskit.test.mock import FakeSantiago
santiago = FakeSantiago()
santiago.configuration().basis_gates

['id', 'rz', 'sx', 'x', 'cx', 'reset']
```

So qc will need transpiling before running. In the next code snippet, we'll see what the transpiler does when we give it qc and tell it to transpile for santiago:

```
t_qc = transpile(qc, santiago)
t_qc.draw()
```

Figure 4-2 shows the output of t_qc.draw().

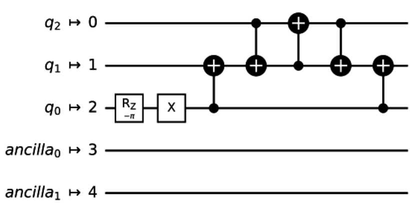

Figure 4-2. Result of transpiling a simple circuit

We can see in Figure 4-2 the transpiler has:

- Mapped (virtual) qubits 0, 1 and 2 in qc to (physical) qubits 2, 1 and 0 in t_qc, respectively
- Added three more CXGate-s to swap (physical) qubits 0 and 1
- Replaced our YGate with an RZGate and an XGate
- Added two extra qubits (as santiago has 5 qubits)

Most of this seems pretty reasonable, except the addition of all those `CXGate`-s. `CXGate`-s are generally quite expensive operations, so we want to avoid them as much as possible. So why has the transpiler done this? In some quantum systems, including `santiago`, not all qubits can communicate directly with ~~eachother~~.

We can check which qubits can talk to ~~eachother~~ through that system's *coupling map* (you can get this by running `backend.configuration().coupling_map`). A quick look at `santiago`-'s coupling map shows us that physical qubit 2 can't talk to physical qubit 0, so we need to add a swap somewhere.

The code ~~below~~ is the output of `santiago.configuration().coupling_map`:

```
[[0, 1], [1, 0], [1, 2], [2, 1], [2, 3],
 [3, 2], [3, 4], [4, 3]]
```

When calling `transpile`, if we set `initial_layout=[1,0,2]` we can change the way `qc` maps to the backend~~,~~ and avoid unnecessary swaps. Here, the index of each element in the list represents the *virtual* qubit (in `qc`), and the value at that index represents the *physical* qubit. This improved layout overrides the transpiler's guess, and it doesn't need to insert any extra `CXGate`-s. The following code snippet shows this:

```
t_qc = transpile(qc, santiago, initial_layout=
                                        [1,0,2])
t_qc.draw()
```

Figure 4-3 shows the output of `t_qc.draw()` in the code snippet ~~above~~.

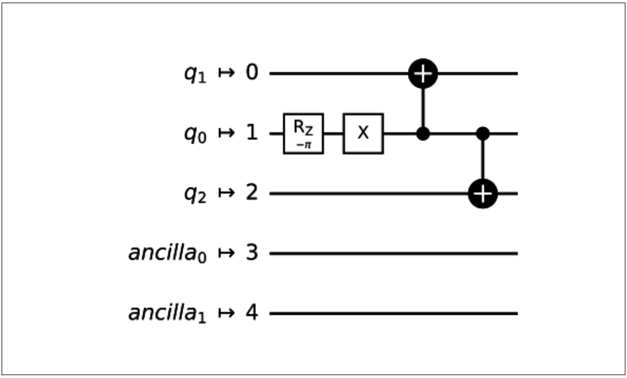

Figure 4-3. Result of transpiling a simple circuit with a smarter initial layout

As `santiago` ~~only~~ has ~~5~~ qubits, it was relatively easy to spot a good layout for this circuit on this device. For larger circuit/device combinations, we will want to do this algorithmically. One option is to set `optimization_level=2` to ask the transpiler to use a smarter (but more expensive) algorithm to select a better layout. The `transpile` function accepts ~~4~~ possible settings for `optimization_level`:

0. With `optimization_level=0`, the transpiler simply does the absolute minimum necessary to get the circuit running on the backend. The initial layout keeps the indices of physical and virtual qubits the same, adds any swaps needed, and converts all gates to basis gates.
1. This is the default value. With `optimization_level=1`, the transpiler makes smarter decisions. For example, if we have ~~less~~ virtual than physical qubits, the transpiler chooses the most well-connected subset of physical qubits and maps the virtual qubits to these. The transpiler also combines/removes sequences of gates where possible (e.g., two `CXGate`s that cancel out).

2. With `optimization_level=2`, the transpiler will search for an initial layout that doesn't need any swaps to execute the circuit, or failing this go for the most well connected subset of qubits. Like level 1, the transpiler also tries to collapse and cancel out gates where possible.
3. This is the highest value we can set. In addition to the measures taken with `optimization_level=2`, with `optimization_level=3` the transpiler will use smarter algorithms to cancel out gates.

Transpiler Passes

Depending on your use case, the transpiler is often invisible. Functions like `execute` call it automatically, and thanks to the transpiler we can usually ignore the specific device we're working on when creating circuits. Despite this low profile, the transpiler can have a huge effect on the performance of a circuit. In this section, we'll look at the decisions the transpiler makes, and see how to change its behaviour when we need to.

The PassManager

We build a transpilation routine from a bunch of smaller "passes". Each pass is a program that performs a small task (e.g. deciding the initial layout, or inserting swap gates), and we use a `PassManager` object to organise our sequence of passes. In this section, we'll show a simple example using the `BasicSwap` pass.

First, we need a quantum circuit to transpile. The following code snippet creates a simple circuit we'll use as an example:

```
from qiskit import QuantumCircuit
qc = QuantumCircuit(3)
qc.h(0)
qc.cx(0, 2)
qc.cx(2, 1)
qc.draw()
```

Figure 4-4 shows the output of `qc.draw()`.

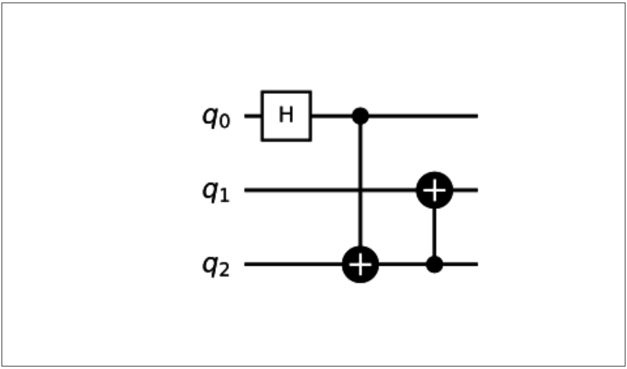

Figure 4-4. Simple circuit containing two CXGates

Next, we need to import and construct the `PassManager`~~,~~ and the passes we want to use. The `BasicSwap` constructor asks for the coupling map of the device we want to run our circuit on. In the following code snippet, we'll pretend we want to run this on a device in which qubit 0 can't interact with qubit 2 (but qubit 1 can interact with both). The `PassManager` constructor asks for the passes we want to apply to our circuit, which in this case is just the `basic_swap` pass we created in the line ~~above~~.

```
from qiskit.transpiler import PassManager,
                              CouplingMap
from qiskit.transpiler.passes import BasicSwap

coupling_map = CouplingMap([[0,1], [1,2]])
basic_swap_pass = BasicSwap(coupling_map)
pm = PassManager(basic_swap_pass)
```

Now we've created our transpilation procedure, we can apply it to the circuit using the ~~code~~ following snippet.

```
routed_qc = pm.run(qc)
routed_qc.draw()
```

Figure 4-5 shows the output of `routed_qc.draw()`.

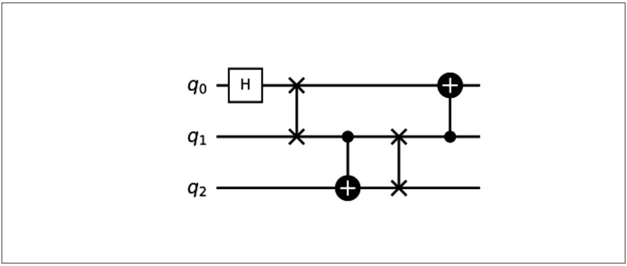

Figure 4-5. Simple circuit containing two CXGates, and two swaps needed to execute on hardware

In Figure 4-5, we can see the `basic_swap` pass has added in two swap gates to carry out the `CXGate`-s, though note that it hasn't returned the qubits to their original order.

Compiling-/-Translating Passes

To get a circuit running on a device, we need to convert all the operations in our circuit to instructions the device supports. This can involve breaking high-level gates into lower-level gates (a form of compiling), or translating one set of low-level gates to another. Figure 4-6 shows how the transpiler might break a multi-controlled-X gate down to smaller gates.

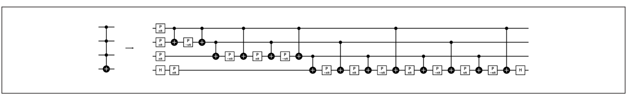

Figure 4-6. Example of a multi-controlled-X-Gate decomposed into H, Phase, and CXGates

At the time of writing, Qiskit has two ways of working out how to break a gate down into smaller gates. The first is through the gate's `definition` attribute. If set, this attribute contains a `QuantumCircuit` equal to that gate. The `Decompose` and `Unroller` passes both use this definition to expand circuits. The `Decompose` pass ~~only~~ expands circuit by one-level, i.e. it won't then try to decompose the definitions we replaced each gate with. The `.decompose()` method of the `QuantumCircuit` class uses the `Decompose` pass. The `Unroller` pass is similar, but it will continue decomposing the definitions of each gate recursively until the circuit ~~only~~ contains the basis gates we specify when we construct it.

The second way of breaking down gates is by consulting an `EquivalenceLibrary`. This library can store many definition circuits for each instruction, allowing passes to choose how to decompose each circuit. This has the advantage of not being tied to one specific set of basis gates. The `BasisTranslator` constructor needs an `EquivalenceLibrary`~~,~~ and a list of gate name labels. If the circuit contains gates *not* in the equivalence library, then we have no option but to use those gates' built-in definitions. The `UnrollCustomDefinitions` pass looks at the `EquivalenceLibrary`, and if each gate does not have an entry in the library, it unrolls that gate using it~~'~~s `.definition` attribute. In the preset transpiler routines (which we'll see later in the chapter), we'll usually see the `UnrollCustomDefinitions` pass immediately before the `BasisTranslator` pass.

Routing Passes

Some devices can ~~only~~ perform multi~~-~~qubit gates between specific subsets of qubits. IBM's hardware tends to ~~only~~ allow one multi~~-~~qubit gate (the `CXGate`)~~,~~ and can ~~only~~ perform these gates between specific pairs of qubits. We call a list of each pair of possible two-qubit interactions a ~~“~~coupling map”~~.~~ We saw an example of this in the `PassManager` section, earlier in this chapter. In that example, we overcame this limitation by using swap gates to move qubits around in the coupling map. Figure 4-7 shows an example of a coupling map.

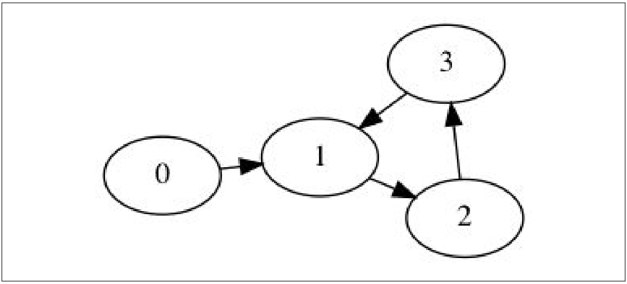

Figure 4-7. Drawing of a coupling map: [[0,1], [1,2], [2,3], [3,1]]

Qiskit has a few algorithms to add these swap gates. Table 4-1 lists each of the available swapping passes, with a brief description of the pass.

Table 4-1. Swapping transpiler passes available in Qiskit

Name	Explanation
`BasicSwap`	This pass does the least computational work needed to get the circuit running on the backend.
`LookaheadSwap`	Unlike `BasicSwap`, this pass uses a smarter algorithm to reduce the number of swap gates. It does a best-first search through all the potential combinations of swaps
`StochasticSwap`	This is the swap pass used in the preset pass managers. This pass is not deterministic, so might not produce the same circuit each time.
`SabreSwap`	This pass uses the "SABRE" ("SWAP-based BidiREctional heuristic search") algorithm to try and reduce the number of swaps needed.
`BIPMapping`	This pass solves ~~both~~ the initial layout~~,~~ and swaps at the same time. The pass maps these problems to a BIP (Binary Integer Programming) problem, which it solves using external programs (`docplex` and `CPLEX`) you will need to install. Additionally, this pass does not cope well with large coupling maps (>~ 10 qubits)

Optimization Passes

The transpiler acts partly as a compiler, and like most compilers, it also includes some optimization passes. The biggest problem in modern quantum computers is noise, and the focus of these optimization passes is to reduce the noise in the output circuit as much as possible. Most of these optimization passes try to reduce noise and running time by minimizing gate count.

The simplest optimizations look for sequences of gates that have no effect, so we can safely remove them. For example, two `CXGates` back-to-back would have no effect on the unitary of the circuit, so the `CXCancellation` pass removes them. Similarly, the `RemoveDiagonalGatesBeforeMeasure` pass does as it says on the tin and removes any gates with diagonal unitaries immediately before a measurement (as they won't change measurements in the computational basis). The `OptimizeSwapBeforeMeasure` pass removes SWAP gates immediately before a measurement and remaps the measurements to the classical register to preserve the output bit-string.

Qiskit also has smarter optimization passes, that attempt to replace groups of gates with smaller or more efficient groups of gates. For example, we can easily collect sequences of single-qubit gates and replace them with a single `U3Gate`, which we can then break back down into an efficient set of basis gates. The `Optimize1qGates` and `Optimize1qGatesDecomposition` passes both do this for different sets of initial gates. We can also do the same for two-qubit gates; `Collect2qBlocks` and `ConsolidateBlocks` find sequences of two-qubit gates and compile them into one two-qubit unitary. The `UnitarySynthesis` pass can then break this back down to the basis gates of our choosing.

For example, Figure 4-8 shows two circuits with identical unitaries, but different numbers of gates.

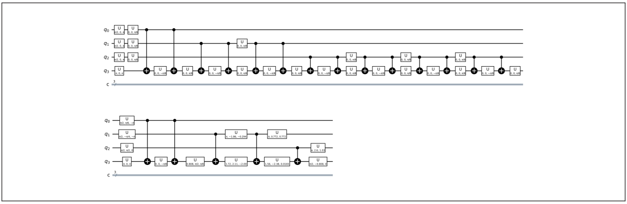

Figure 4-8. Example of the same circuit after going through two different transpilation processes.

Initial Layout Selection Passes

As with routing, we also need to choose how to initially map our virtual circuit qubits to the physical device qubits. Table 4-2 lists some layout selection algorithms Qiskit offers.

Table 4-2. Initial layout transpiler passes available in Qiskit

Name	Explanation
`TrivialLayout`	This pass simply maps circuit qubits to physical qubits via their indexes. E.g. the circuit qubit with index 3 will map to the device qubit with index 3.
`DenseLayout`	This pass finds the most well-connected group of physical qubits, and maps the circuit qubits to this group.
`NoiseAdaptiveLayout`	This pass uses information about the device's noise properties to choose a layout.
`SabreLayout`	This pass uses the SABRE algorithm to find an initial layout requiring as few SWAPs as possible.
`CSPLayout`	This pass converts layout selection to a constraint satisfaction problem (CSP). The pass then uses the `constraint` module's `RecursiveBacktrackingSolver` to try and find the best layout.

Preset PassManagers

When we used the high-level ~~function~~ `transpile` before, we didn't worry about the individual passes and instead set the `optimization_level` parameter. This parameter tells the transpiler to use one of four preset pass managers. Qiskit builds these preset pass managers through functions that take configuration settings~~,~~ and return a `PassManager` object. Now we understand some passes, we can have a look at what the different transpilation routines are doing.

~~Below~~ is the code we used to extract the passes used for a simple transpilation routine in case you want to reproduce it~~.~~

```
from qiskit.transpiler import (PassManagerConfig,
                               CouplingMap)
from qiskit.transpiler.preset_passmanagers import\
                            level_0_pass_manager
from qiskit.test.mock import FakeSantiago

sys_conf = FakeSantiago().configuration()
pm_conf = PassManagerConfig(
    basis_gates=sys_conf.basis_gates,
    coupling_map=CouplingMap(sys_conf.coupling_map))

for i, step in enumerate(
    level_0_pass_manager(pm_conf).passes()):
    print(f'Step {i}:')
    for transpiler_pass in step['passes']:
        print(f'  {transpiler_pass.name()}')
```

Remember that `optimization_level=0` does the bare minimum required to get the circuit running on the device. We can see in *step 1* that the transpiler uses the `TrivialLayout` pass to map the circuit qubits to the device qubits. The transpiler then unrolls the circuit down to single and two-qubit gates, then does the `StochasticSwap` routing pass~~,~~ before fully unrolling the circuit. We have not covered some of the passes ~~below~~ in this chapter because they are analysis passes that do not affect the circuit~~,~~ or because they are housekeeping passes for which we don't have a choice of algorithm. These passes are unlikely to have an avoidable, negative effect on the performance of our circuits. We have also not covered some pulse-level passes that are out of the scope of this chapter.

```
Step 0:
  SetLayout
Step 1:
  TrivialLayout
Step 2:
  FullAncillaAllocation
  EnlargeWithAncilla
  ApplyLayout
Step 3:
  Unroll3qOrMore
Step 4:
  CheckMap
Step 5:
  BarrierBeforeFinalMeasurements
  StochasticSwap
Step 6:
  UnrollCustomDefinitions
  BasisTranslator
Step 7:
  TimeUnitConversion
Step 8:
  ValidatePulseGates
  AlignMeasures
```

Remember that `optimization_level=0` does the bare minimum needed to get the circuit running on the device. Notably, we can see it uses `TrivialLayout` to choose an initial layout, then expands the circuit to have the same number of qubits as the device. The transpiler then unrolls the circuit to single and two-qubit gates, and uses `StochasticSwap` for routing. Finally, it unrolls everything as far as possible, and translates the circuit to the device's basis gates.

~~Whereas for~~ `optimization_level=3` the `PassManager` contains the following passes:

```
Step 0:
  Unroll3qOrMore
Step 1:
  RemoveResetInZeroState
  OptimizeSwapBeforeMeasure
  RemoveDiagonalGatesBeforeMeasure
Step 2:
  SetLayout
Step 3:
  TrivialLayout
  Layout2qDistance
Step 4:
  CSPLayout
Step 5:
  DenseLayout
Step 6:
  FullAncillaAllocation
  EnlargeWithAncilla
  ApplyLayout
Step 7:
  CheckMap
Step 8:
  BarrierBeforeFinalMeasurements
  StochasticSwap
Step 9:
  UnrollCustomDefinitions
  BasisTranslator
Step 10:
  RemoveResetInZeroState
Step 11:
  Depth
  FixedPoint
  Collect2qBlocks
  ConsolidateBlocks
  UnitarySynthesis
  Optimize1qGatesDecomposition
  CommutativeCancellation
  UnrollCustomDefinitions
  BasisTranslator
Step 12:
```

```
    TimeUnitConversion
  Step 13:
    ValidatePulseGates
    AlignMeasures
```

This `PassManager` is quite different. After unrolling to single and two-qubit gates, we can already see some optimization passes in *step 1* removing unnecessary gates. The transpiler then tries a few different layout selection approaches. First it checks if the `TrivialLayout` is optimal (i.e. if it doesn't need any SWAPs inserting to execute on the device). If it isn't, the transpiler then tries to find a layout using `CSPLayout`. If `CSPLayout` fails to find a solution, then the transpiler uses the `DenseLayout` algorithm. Next (*step 6*), the transpiler adds extra qubits (if needed) to make the circuit have the same number of qubits as the device. It then uses the `StochasticSwap` algorithm to make all ~~2~~-qubit gates possible on the device's coupling map. With the routing taken care of, the~~n~~ transpiler then translates the circuit to the device's basis gates~~,~~ before attempting some final optimizations in *step 11*.

Looking at the `optimization_level=3` passes, we can see the transpiler is a very sophisticated program that can have a ~~large~~ influence on the behaviour of your circuits. Fortunately, you now understand the problems the transpiler must solve~~,~~ and some of the algorithms it uses to solve them.

PART II

Quantum Information and Algorithms

In Part I we explored fundamentals of Qiskit, including creating and running quantum circuits, and visualizing their results. Here in Part II we'll discuss modules in Qiskit that leverage these fundamentals to apply quantum mechanical concepts to representing and processing information. In Chapter 5, "Quantum Information", we'll begin this journey by exploring quantum states, operators, channels and measures.

Then in Chapter 6, "Operator Flow", we'll examine a module in Qiskit that facilitates expressing and manipulating quantum states and operations. Finally in Chapter 7, "Quantum Algorithms", we'll explore higher-level features of Qiskit that solve problems using algorithms that leverage the power of quantum information.

CHAPTER 5
Quantum Information

The first three letters in the name ~~“~~Qiskit~~”~~ stand for *quantum information science*, which is the study of how quantum systems may used to represent, process, and transmit information. The `quantum_info` module of Qiskit contains classes and functions that focus on those capabilities.

Using Quantum Information States

The `qiskit.quantum_info` module contains a few classes, shown in ~~the~~ Table 5-1, that represent quantum information states.

Table 5-1. Classes that represent states in the `qiskit.quantum_info` module

Class name	Description
`Statevector`	Represents a statevector
`DensityMatrix`	Represents a density matrix
`StabilizerState`	Simulation of stabilizer circuits

We’ll focus on the two most commonly used of these, namely the `Statevector` and `DensityMatrix` classes.

Using the Statevector Class

The `Statevector` class represents a quantum statevector, and contains functionality for initializing and operating on the statevector. For example, as shown in the following code snippet, a `Statevector` may be instantiated by passing in a `QuantumCircuit` instance.

```
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)

statevector = Statevector(qc)
print(statevector.data)

output:
  [0.70710678+0.j 0.+0.j 0.+0.j 0.7071+0.j]
```

Notice that instead of running the circuit on a quantum simulator to get the statevector (as shown in the code in "Using the AerSimulator to calculate and hold a statevector" on page 44), we simply create an instance of `Statevector` with the desired `QuantumCircuit`.

Another way of creating a `Statevector` is to pass in a normalized complex vector, as shown in the following code snippet.

```
import numpy as np
from qiskit.quantum_info import Statevector

statevector = Statevector([1, 0, 0, 1] / np.sqrt(2))
print(statevector.data)

output:
  [0.70710678+0.j 0.+0.j 0.+0.j 0.7071+0.j]
```

Yet another way of creating a `Statevector` is to pass a string of eigenstate ket labels to the `from_label` method, as shown in the following code snippet.

```
from qiskit.quantum_info import Statevector

statevector = Statevector.from_label('01-')
print(statevector.data)

output:
  [0.+0.j  0.+0.j  0.70710678+0.j -0.70710678+0.j
  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
```

Table ~~5-2 and Table 5-3~~ describe some of the methods and attributes in the `Statevector` class.

Table 5-2. Some `Statevector` methods

Method name	Description
`conjugate`	Returns the complex conjugate of the statevector.
`copy`	Creates and returns a copy of the statevector.
`dims`	Returns a tuple of dimensions.
`draw`	Returns a visualization of the `Statevector`, given the desired output method from the following: *text*, *latex*, *latex_source*, *qsphere*, *hinton*, *bloch*, *city*, or *paulivec*. Also see Chapter 3.
`equiv`	Returns a boolean indicating whether a supplied `Statevector` is equivalent to this one, up to a global phase.
`evolve`	Returns a quantum state evolved by the supplied operator. Also see "Using Quantum Information Operators" on page 95.
`expand`	Returns the reverse-order tensor product state of this statevector and a supplied `Statevector`.
`expectation_value`	Computes and returns the expectation value of a supplied operator.
`from_instruction`	Returns the `Statevector` output of a supplied `Instruction` or `QuantumCircuit` instance.

Method name	Description
`from_label`	Instantiates a `Statevector` given a string of eigenstate ket labels. Each ket label may be `0`, `1`, `+`, `-`, `r`, or `l`, and correspond to the six states found on the X, Y and Z axes of a Bloch sphere.
`inner`	Returns the inner product of this statevector and a supplied `Statevector`.
`is_valid`	Returns a boolean indicating whether this statevector has norm 1.
`measure`	Returns the measurement outcome as well as post-measure state.
`probabilities`	Returns the measurement probability vector.
`probabilities_dict`	Returns the measurement probability dictionary.
`purity`	Returns a number from 0 to 1 indicating the purity of this quantum state. 1.0 indicates that this statevector represents a pure quantum state.
`reset`	Resets to the 0 state.
`reverse_qargs`	Returns a `Statevector` with reversed basis state ordering.
`sample_counts`	Samples the probability distribution a supplied number of times, returning a dictionary of the counts.
`sample_memory`	Samples the probability distribution a supplied number of times, returning a list of the measurement results.
`seed`	Sets the seed for the quantum state random number generator.
`tensor`	Returns the tensor product state of this statevector and a supplied `Statevector`.
`to_dict`	Returns the statevector as a dictionary.
`to_operator`	Returns a rank-1 projector operator by taking the outer product of the statevector with its complex conjugate.