Quantum Computing in Practice with Qiskit® and IBM Quantum Experience®

Practical recipes for quantum computer coding at the gate and algorithm level with Python

Hassi Norlén

Quantum Computing in Practice with Qiskit® and IBM Quantum Experience®

Commissioning Editor: Kunal Chaudhari

Acquisition Editor: Alok Dhuri

Senior Editor: Nitee Shetty

Content Development Editor: Ruvika Rao

Technical Editor: Gaurav Gala

Copy Editor: Safis Editing

Project Coordinator: Deeksha Thakkar

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Prashant Ghare

First published: November 2020

Production reference: 1201120

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83882-844-8

www.packt.com

Table of Contents

5
Touring the IBM Quantum® Hardware with Qiskit®

6
Understanding the Qiskit® Gate Library

7
Simulating Quantum Computers with Aer

Preface

IBM Quantum Experience® with **Qiskit**® together form a popular and easy-to-use quantum computing platform. They let you access and program actual IBM quantum computer hardware in the cloud, but you can also run your code on local and cloud-based simulators.

This book is designed to teach you how to implement quantum programming in a Python® environment, first at an elementary level, and later moving to more advanced examples. The locally installable **Quantum Information Science Toolkit** (**Qiskit**) software is built on Python and represents the most accessible tool available today for learning quantum computing.

Throughout the recipes of this book, we will introduce the Qiskit classes and methods step by step, starting with the very basic concepts such as installing and upgrading Qiskit, checking which version you are running, and so on. We then move on to understanding the building blocks that are required to create and run quantum programs and how to integrate these Qiskit components in your own hybrid quantum/classical programs to leverage Python's powerful programming features.

We'll explore, compare and contrast **Noisy Intermediate-Scale Quantum** (**NISQ**) computers and universal fault-tolerant quantum computers using simulators and actual hardware, looking closely at simulating noisy backends and how to mitigate for noise and errors on actual hardware, implementing the Shor code method for quantum error correction of a single qubit.

Finally, we'll take a look at quantum algorithms to see how they differ from classical algorithms. We will take a closer look at coding Grover's algorithm, and then use Qiskit Aqua to run versions of Grover's and Shor's algorithms to show how you can reuse already constructed algorithms directly in your Qiskit code. We do all of this as a sweeping tour of Qiskit, IBM's quantum information science toolkit, and its constituent layers: Terra, Aer, Ignis, and Aqua.

We will also use the online IBM Quantum Experience® user interface for drag-and-drop quantum computing. Everything we do in this book, and way more, can be coded in the cloud on IBM Quantum Experience®.

Each chapter contains code samples to explain the principles taught in each recipe.

Who this book is for

This book is for developers, data scientists, researchers, and quantum computing enthusiasts who want to understand how to use Qiskit® and IBM Quantum Experience® to implement quantum solutions. Basic knowledge of quantum computing and the Python programming language is beneficial.

What this book covers

This cookbook is a problem-solution- and exploration-based approach to understanding the nuances of programming quantum computers with the help of IBM Quantum Experience®, Qiskit®, and Python.

Chapter 1, *Preparing Your Environment,* walks you through how to install Qiskit® as a Python 3.5 extension on your local workstation. You'll also register with IBM Quantum Experience®, get your API key, and grab the sample code.

Chapter 2, *Quantum Computing and Qubits with Python,* shows how to use Python to code simple scripts to walk you through the concept of bits and qubits and how quantum gates work without Qiskit® or IBM Quantum Experience®.

Chapter 3, *IBM Quantum Experience® – Quantum Drag and Drop,* looks at IBM Quantum Experience®, IBM Quantum's online, cloud-based drag-and-drop tool for programming quantum computers. Here you will code a simple program and learn how to move between Qiskit® and IBM Quantum Experience®.

Chapter 4, *Starting at the Ground Level with Terra,* explores a set of basic quantum programs, or circuits, to examine fundamental concepts such as probabilistic computing, superposition, entanglement, and more. We will also run our first programs on an actual physical IBM quantum computer.

Chapter 5, *Touring the IBM Quantum® Hardware with Qiskit®,* looks at the IBM Quantum® backends, exploring various physical aspects that impact the results of your quantum programs.

Chapter 6, *Understanding the Qiskit® Gate Library,* gives an overview of the quantum gates that are offered out of the box with Qiskit® to see what they do to your qubits. We take a look at the universal quantum gates from which all other quantum gates are built, and also expand from one-qubit gates, to the two-, three-, and more qubit gates needed for more advanced quantum circuits.

Chapter 7, Simulating Quantum Computers with Aer, helps you run your circuits on a collection of simulators that you can use locally or in the cloud. You can even set your simulators up to mimic the behavior of an IBM Quantum® backend, to test your circuits under realistic conditions on your local machine.

Chapter 8, Cleaning Up Your Quantum Act with Ignis, explains how to clean up your measurement results by understanding how our qubits behave, and looks at how we can correct for noise by using noise mitigation circuits such as the Shor code.

Chapter 9, Grover's Search Algorithm, builds Grover's search algorithm, a quadratic speedup of classical search algorithms. We will use a unique quantum tool, quantum phase kickback. We build several different versions of the algorithm to run on both simulators and IBM Quantum® backends.

Chapter 10, Getting to Know Algorithms with Aqua, uses premade Qiskit Aqua versions of two of the most well-known quantum algorithms: Grover's search algorithm and Shor's factoring algorithm. We also take a quick tour of the Qiskit Aqua algorithm library.

To get the most out of this book

It helps if you have acquainted yourself a little bit with basic quantum computing concepts; we will not spend too much time with proofs or the deeper details. Python programming skills are also helpful, especially when we start building slightly more complex hybrid quantum/classical programs. A basic understanding of linear algebra with vector and matrix multiplication will definitely help you understand how quantum gates work, but we let Python and NumPy do the hard work.

Qiskit® supports Python 3.6 or later. The code examples in this book were tested on Anaconda 1.9.12 (Python 3.7.0) using its bundled Spyder editor with Qiskit® 0.21.0, and on the online IBM Quantum Experience® Code lab environment. We recommend our readers to use the same.

Software/hardware covered in the book	**OS requirements**
Python 3.7	Windows, macOS, and Linux (any)
Qiskit 0.21.0	Python 3.5+
Anaconda Navigator 1.9	Windows, macOS, and Linux (any)

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Quantum-Computing-in-Practice-with-Qiskit-and-IBM-Quantum-Experience`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781838828448_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `log_length(oracle_input,oracle_method)` function takes as input the oracle input (log or bin) and the oracle method (logical expression or bit string) and returns the ideal number of iterations the Grover circuit needs to include."

A block of code is set as follows:

```
def log_length(oracle_input,oracle_method):
    from math import sqrt, pow, pi, log
    if oracle_method=="log":
        filtered = [c.lower() for c in oracle_input if
            c.isalpha()]
        result = len(filtered)
```

Any command-line input or output is written as follows:

```
$ conda create -n environment_name python=3
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Clicking the job results box opens the **Result** page, and displays the final result of the job you just ran."

Tips or important notes
Appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Preparing Your Environment

Before you can start working on your quantum programs, you must have a Python environment to execute your code. The examples in this book can be run both on your local machine by using the Qiskit® developer environment provided by IBM Quantum® and in an online environment on IBM Quantum Experience®.

In this chapter, we will take a look at both environments, get you a login account on IBM Quantum Experience®, and install a local version of **Qiskit®**. We will also discuss the fast-moving environment that is open source Qiskit®, and how to keep your local environment up to date.

We will cover the following recipes:

- Creating your IBM Quantum Experience® account
- Installing Qiskit®
- Downloading the code samples
- Installing your API key and accessing your provider
- Keeping your Qiskit® environment up to date

So, let's get started. This chapter, and its contents, is pretty important as it provides you with the foundation on which you can start building your Qiskit® future. Do spend a moment or two setting this up, and then get going with the recipes in this book to get started with quantum programming on Qiskit®. To get you started quickly, you can also grab and run the sample recipe code that is provided with this book.

Technical requirements

The recipes that we will discuss in this chapter can be found here: `https://github.com/PacktPublishing/Quantum-Computing-in-Practice-with-Qiskit-and-IBM-Quantum-Experience/tree/master/Chapter01`.

You can run the recipes in this book in your local Qiskit® environment that you set up as a part of this chapter. You can also run most of them in the **Quantum Lab** environment of the online IBM Quantum Experience®. This is also true for the `c1_r1_version.py` recipe in this chapter, which lists the installed version of Qiskit® in the environment in which you run the recipe.

For information about how to download the recipes, see *Downloading the code samples*.

The local environment in which you choose to install Qiskit® must have **Python 3.5 or higher** installed (as of this book's writing). For detailed information about the most current requirements for Qiskit® installation, see the Qiskit® requirements page at `https://qiskit.org/documentation/install.html`.

IBM Quantum® recommends using the Anaconda distribution of Python (`https://www.anaconda.com/distribution/`), and to use virtual environments to keep your Qiskit® installation isolated from your usual Python environment.

New to virtual environments?

Virtual environments provide isolated Python environments that you can modify separately from each other. For example, you can create an isolated environment for your Qiskit® installation. You will then install Qiskit® only in that environment, and not touch the Python framework in the base environment which will then contain an untarnished version of Python.

As Qiskit® releases new versions of their packages, there is technically nothing stopping you from creating a new isolated environment for each updated version of Qiskit® to retain your old and stable version for your Qiskit® quantum programming, and a new environment where you can test updated versions of Qiskit®. You will find more on this in the *Keeping your Qiskit® environment up to date* recipe.

Creating your IBM Quantum Experience® account

Your key to exploring quantum computer programming with IBM is your *IBM Quantum Experience® account*. This free account gives you access to the online IBM Quantum Experience® interface, and the programming tools that are available there. An IBM Quantum Experience® account is not technically required to test out IBM Quantum Experience® or to install Qiskit® but is required to run your programs on the freely available IBM quantum computers, which are, after all, probably why you are reading this book in the first place.

Getting ready

To set up your IBM Quantum Experience® account, you can log in with an IBMid, or with one of the following:

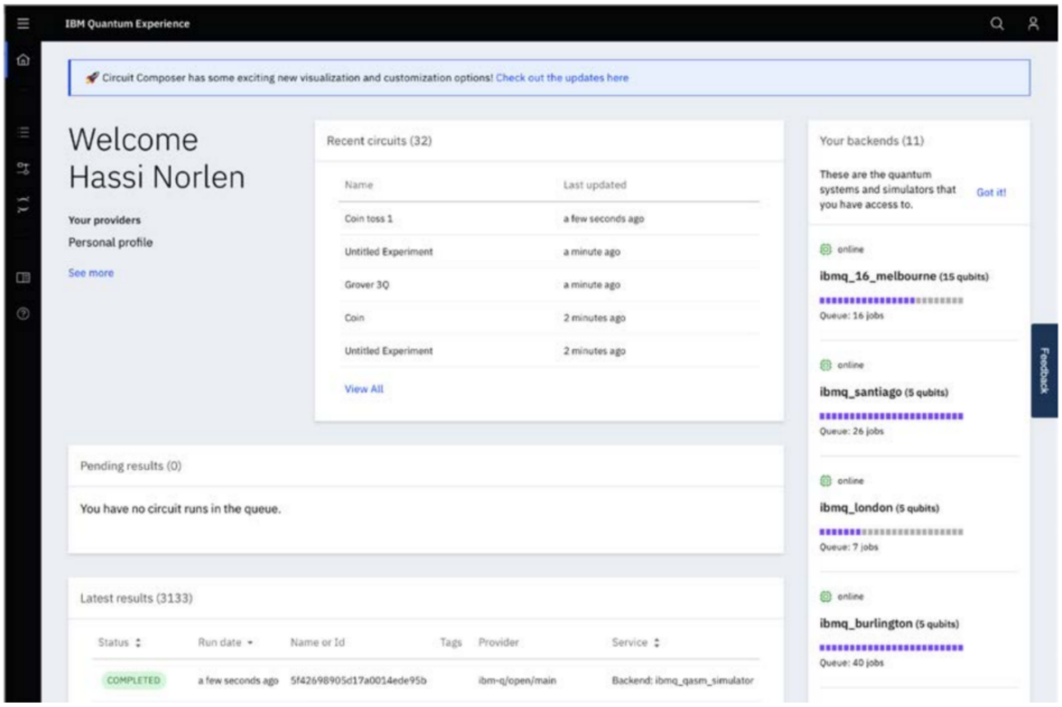
- A Google account
- A GitHub account
- A LinkedIn account
- A Twitter account
- An email address

How to do it...

1. In your browser (Google Chrome seems to work best), go to this link: `https://quantum-computing.ibm.com/login`.
2. Enter your IBMid credentials or select another login method.

   You can also skip the sign-in, which will give you access to IBM Quantum Experience® but with a limit of 3 qubits for your quantum circuits, and with simulator backends only.

3. Once you have logged in, you now have an activated IBM Quantum Experience® account, and will find yourself at the main dashboard:



🚀 Circuit Composer has some exciting new visualization and customization options! Check out the updates here

Welcome
Hassi Norlen

Your providers
Personal profile
See more

Recent circuits (32)

Name	Last updated
Coin toss 1	a few seconds ago
Untitled Experiment	a minute ago
Grover 3Q	a minute ago
Coin	2 minutes ago
Untitled Experiment	2 minutes ago

View All

Pending results (0)

You have no circuit runs in the queue.

Latest results (3133)

Status	Run date	Name or Id	Tags	Provider	Service
COMPLETED	a few seconds ago	5f42698905d17a0014ede95b		ibm-q/open/main	Backend: ibmq_qasm_simulator

Your backends (11)

These are the quantum systems and simulators that you have access to. Got it!

online
ibmq_16_melbourne (15 qubits)
Queue: 16 jobs

online
ibmq_santiago (5 qubits)
Queue: 26 jobs

online
ibmq_london (5 qubits)
Queue: 7 jobs

online
ibmq_burlington (5 qubits)
Queue: 40 jobs

Figure 1.1 – The IBM Quantum Experience® home page

4. From here, you have a couple of paths:

   Go to a composer to start building your quantum programs in a graphical user interface. Click the **Circuit composer** left-menu icon ( ) and then go to *Chapter 3, IBM Quantum Experience® – Quantum Drag and Drop*.

   If you want to start writing your quantum programs in Python without first installing a local Qiskit® instance, you can go to the Qiskit® notebooks to start working on your quantum programs in a Jupyter Notebook Python environment. Click on the **Quantum Lab** left-menu icon ( ), click **New Notebook**, and then go to *Chapter 4, Starting at the Ground Level with Terra*.

   If you want to continue down the Qiskit® path for which this book was written, you can now log out of IBM Quantum Experience®, and continue with installing Qiskit® on your local machine.

See also

- *IBM Quantum Experience is quantum on the cloud*: https://www.ibm.com/quantum-computing/technology/experience/.
- *Quantum computing: It's time to build a quantum community*: https://www.ibm.com/blogs/research/2016/05/quantum-computing-time-build-quantum-community/.

Installing Qiskit®

With your Python environment prepared and ready to go, and your IBM Quantum Experience® account set up, you can now use `pip` to install the Qiskit® Python extension. The process should take about 10 minutes or so, after which you can use your Python command line, or your favorite Anaconda Python interpreter to start writing your quantum programs.

Getting ready

This recipe provides information about the generic way to install Qiskit® and does not go into any detail regarding operating system differences or general installation troubleshooting.

For detailed information about the most current requirements for Qiskit® installation, see the Qiskit® requirements page at https://qiskit.org/documentation/install.html.

How to do it...

1. Create your Anaconda virtual environment:

   ```
   $ conda create -n environment_name python=3
   ```

   This will install a set of environment-specific packages.

2. Activate your virtual environment:

   ```
   $ conda activate environment_name
   ```

3. Verify that you are in your virtual environment.

   Your Command Prompt should now include the name of your environment; I used something like this for my own environment with the name `packt_qiskit`:

   ```
   (packt_qiskit) Hassis-Mac:~ hassi$
   ```

Nomenclature

In this chapter, I will print out the full prompt like this `(environment_name) … $` to remind you that you must execute the commands in the correct environment. In the rest of the book, I will assume that you are indeed in your Qiskit-enabled environment and just show the generic prompt: `$`.

4. If needed, do a `pip` update.

   To install Qiskit®, you must use `pip` as Qiskit® is not distributed as `conda` packages. The latest Qiskit® requires pip 19 or newer.

   If you have an older version of `pip`, you must upgrade using the following command:

   ```
   (environment_name) … $  pip  install  -U  pip
   ```

5. Install Qiskit®.

   So, with everything set up and prepared, we can now get on to the main course, installing the Qiskit® code in your environment. Here we go!

   ```
   (environment_name) … $  pip install qiskit
   ```

Failed wheel build

As part of the installation, you might see errors that the *wheel* failed to build. This error can be ignored.

6. Use Python to Verify that Qiskit® installed successfully.

   Open Python:

   ```
   (environment_name) … $  python3
   ```

   Import Qiskit®:

   ```
   >>> import qiskit
   ```

   This is a little bit exciting; we are going to use Qiskit® code for the first time. Granted, not exactly for programming a quantum computer, but at least to make sure that we are now ready to start programming your quantum programs.

   List the version details:

   ```
   >>> qiskit.__qiskit_version__
   ```

This should display the versions of the installed Qiskit® components:

```
{'qiskit-terra': '0.15.2', 'qiskit-aer': '0.6.1',
'qiskit-ignis': '0.4.0', 'qiskit-ibmq-provider': '0.9.0',
'qiskit-aqua': '0.7.5', 'qiskit': '0.21.0'}
```

Congratulations, your Qiskit® installation is complete; you are ready to go!

By using `pip install` from your virtual environment, you can install Qiskit® in just that environment, which will then stay isolated from the rest of your Python environment.

There's more...

Qiskit® also comes with some optional visualization dependencies to use visualizations across the Qiskit® components. You can install these with the following command:

```
(environment_name) … $ pip install qiskit[visualization]
```

Note

If you are using the zsh shell you must enclose the component in quotes:
pip install '`qiskit[visualization]`'

See also

For a quick introduction to Anaconda environments, see *Managing environments* in the Anaconda docs: `https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html`.

This book does not, in any way, act as a Qiskit® installation troubleshooting guide, and you might conceivably run into issues when installing it, depending on your local OS, versions, and more.

But fear not, help is on the way. Here are a couple of good and friendly channels on which to reach out for help:

- **Slack**: `https://qiskit.slack.com/`
- **Stack Exchange**: `https://quantumcomputing.stackexchange.com/questions/tagged/qiskit`

Downloading the code samples

The recipes in this book include short, and some not so short, sample programs that will lead you through your first steps in programming quantum computers. You can type in these programs directly from the instructions in the book if you want, but for convenience, you can also grab the sample code directly from the Packt Cookbook GitHub organization.

The Python samples are written for Python 3.5+ and the Qiskit® extension that you installed in your Python environment. The Python samples all have the extension: `.py`.

Getting ready

While you can type the recipes directly into your Python environment, or into Jupyter Notebooks on IBM Quantum Experience® or on your local Anaconda environment, it is somewhat more efficient to download or use **Git** to clone the sample code to your local environment. The advantage of cloning is that you can later refresh your local files from the remote repository if any updates are made.

If you do not plan to use Git, but instead to download the recipes as a compressed file, continue on with *How to do it*.

To use Git to clone the sample code, you must first do the following:

1. Get a GitHub account. These are free and you can sign up for one at `https://github.com`.
2. Install Git in your local environment. For more information, see `https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`.
3. If you are a user interface person, you might also want to install GitHub Desktop, available here: `https://desktop.github.com/`.

How to do it...

You have several different options to download the sample recipes to your local machine.

For each, start by opening your web browser and then go to the *Quantum-Computing-in-Practice-with-Qiskit-and-IBM-Quantum-Experience* GitHub repository at `https://github.com/PacktPublishing/Quantum-Computing-in-Practice-with-Qiskit-and-IBM-Quantum-Experience`.

Downloading the repository as a compressed file

The easiest way to get the recipes is to just grab the sample files as a compressed directory and decompress it on your local machine:

1. At the preceding URL, click the **Clone or download** button and select **Download zip**.
2. Download the compressed file and select a location.
3. Decompress the file.

Cloning the repository using git

1. Click the **Clone or download** button and copy the URL.
2. Open your command line and navigate to the location where you want to clone the directory.
3. Enter the following command:

```
$ git clone https://github.com/PacktPublishing/Quantum-
Computing-in-Practice-with-Qiskit-and-IBM-Quantum-
Experience.git
```

The command should result in something like the following:

```
Cloning into 'Quantum-Computing-in-Practice-with-Qiskit-
and-IBM-Quantum-Experience'...
remote: Enumerating objects: 250, done.
remote: Counting objects: 100% (250/250), done.
remote: Compressing objects: 100% (195/195), done.
remote: Total 365 (delta 106), reused 183 (delta 54),
pack-reused 115
Receiving objects: 100% (365/365), 52.70 MiB | 5.42
MiB/s, done.
Resolving deltas: 100% (153/153), done.
```

Cloning the repository using GitHub Desktop

1. Click the **Clone or download** button and select **Open in desktop**.
2. In the GitHub Desktop dialog, select a directory to clone the repository to and click **OK**.

You can now browse the recipes in this cookbook. Each chapter includes one or more recipes. If you want, you can copy and paste the recipes directly into your Python environment, or into Jupyter Notebooks on IBM Quantum Experience® or on your local Anaconda environment.

Opening a recipe file

So far, you have done everything with the command line. So how about you grab the following Python program and run it from your favorite Python interpreters, such as **Anaconda Spyder** or **Jupyter Notebooks**?

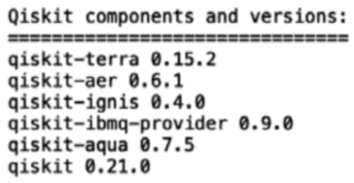
If you have downloaded the sample files, the recipe file is available in the following local directory:

```
<The folder where you downloaded the files>/https://github.
com/PacktPublishing/Quantum-Computing-in-Practice-with-
Qiskit-and-IBM-Quantum-Experience/blob/master/Chapter01/
ch1_r1_version.py
```

The `ch1_r1_version.py` code sample lists the version numbers of the Qiskit® components that we just installed:

```
# Import Qiskit
import qiskit

# Set versions variable to the current Qiskit versions
versions=qiskit.__qiskit_version__

# Print the version number for the Qiskit components

print("Qiskit components and versions:")
print("===============================")

for i in versions:
    print (i, versions[i])
```

When run, the preceding code should give an output similar to the following:



```
Qiskit components and versions:
===============================
qiskit-terra 0.15.2
qiskit-aer 0.6.1
qiskit-ignis 0.4.0
qiskit-ibmq-provider 0.9.0
qiskit-aqua 0.7.5
qiskit 0.21.0
```

Figure 1.2 – A list of the Qiskit® components and versions

The following sections cover how to run the script in the environments that we have available.

Python scripts in Spyder

In your local environment, you can now open the Python scripts in the Python interpreter of your choice; for example, Spyder, which is included with Anaconda:

> **Important**
>
> Be sure that you run your interpreter in the virtual environment in which you installed Qiskit®. Otherwise, it will not be able to find Qiskit®, and the program will not run correctly.

1. Open your Anaconda user interface.
2. Select your virtual environment.
3. Click the **Spyder** tile. If Spyder is not yet installed for your virtual environment, it will now install. This might take a while. Be patient!
4. In Spyder, open the Python script that is included with this chapter:

   ```
   <The folder where you downloaded the files>/https://
   github.com/PacktPublishing/Quantum-Computing-in-
   Practice-with-Qiskit-and-IBM-Quantum-Experience/blob/
   master/Chapter01/ch1_r1_version.py
   ```

5. Click **Run**. The script will now pull out the version numbers of the installed Qiskit® components. You can also open the Python scripts in a Jupyter notebook, for example, in the online IBM Quantum Experience® environment, but this takes a little extra work.

Jupyter Notebooks in Anaconda

1. Open your Anaconda user interface.
2. Select your virtual environment.
3. Click the **Jupyter Notebooks** tile. If Jupyter Notebooks is not yet installed for your virtual environment, it will now install.

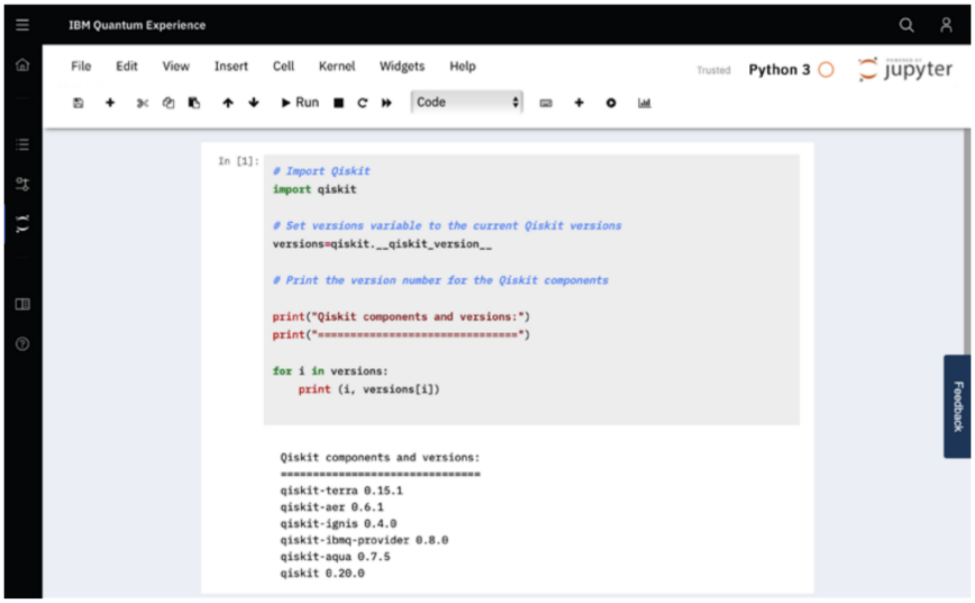
4. Your default web browser opens at your root directory. Browse to and open the following:

```
<The folder where you downloaded the files>/https://
github.com/PacktPublishing/Quantum-Computing-in-
Practice-with-Qiskit-and-IBM-Quantum-Experience/blob/
master/Chapter01/ch1_r1_version.py
```

5. The sample script opens in a Jupyter text editor. You can now see the code but not run it.
6. Go back to the Jupyter browser and click **New** | **Notebook**.
7. Copy and paste the Python script code into the new notebook. You can now click **Run** and see the code execute.

Jupyter Notebooks in IBM Quantum Experience®

1. To run the Python scripts in the online IBM Quantum Experience® notebooks, log in to IBM Quantum Experience® at `https://quantum-computing.ibm.com/login`.
2. On the main dashboard, click on the **Quantum Lab** left-menu icon ( ), then click **New Notebook** and follow the process we discussed *in the Jupyter Notebooks in Anaconda* section.:



```
# Import Qiskit
import qiskit

# Set versions variable to the current Qiskit versions
versions=qiskit.__qiskit_version__

# Print the version number for the Qiskit components

print("Qiskit components and versions:")
print("==============================")

for i in versions:
    print (i, versions[i])
```

```
Qiskit components and versions:
==============================
qiskit-terra 0.15.1
qiskit-aer 0.6.1
qiskit-ignis 0.4.0
qiskit-ibmq-provider 0.8.0
qiskit-aqua 0.7.5
qiskit 0.20.0
```

Figure 1.3 – Running your Qiskit® code on IBM Quantum Experience®

How it works...

The Qiskit®-based Python code that we will be going through in the chapters that follow can be run in any Python environment that meets the Qiskit® requirements, so you have free reins to pick the environment that suits you. And with that environment, you can also freely pick your favorite tools to run the programs.

For this book, I have tested running the code in both the **Spyder** editor that comes as standard with **Anaconda** and with the **Jupyter Notebook** environments on both IBM Quantum Experience® and Anaconda.

Installing your API key and accessing your provider

Now that you have installed Qiskit®, you can immediately start creating your quantum programs and run these on local simulators. If, however, at some point, you want to run your quantum code on actual IBM Quantum® hardware, you must install your own unique API key locally.

API keys on IBM Quantum Experience®

If you are running your Qiskit® programs in the IBM Quantum Experience® notebook environment, your API key is automatically registered.

Getting ready

Before you can install your API key, you must first have an IBM Quantum Experience® account. If you have not yet created one, go back and do it (see the *Creating your IBM Quantum Experience® account* section).

How to do it...

Let's take a look at how to install the API key locally:

1. Log in to IBM Quantum Experience® at `https://quantum-computing.ibm.com/login`.
2. On the IBM Quantum Experience® dashboard, find your user icon in the upper-right corner, click it, and select **My account**.
3. On the account page, find the **Qiskit in local environment** section, and click **Copy token**.

4. You can now paste your token in a temporary location or keep it in the clipboard.
5. On your local machine, access your Qiskit® environment. We have done this one, but here's a repeat of the process if you are using Anaconda.
6. Activate your virtual environment:

```
$ conda activate environment_name
```

7. Open Python:

```
$(environment_name) … $  python3
```

Verify that the Python header displays and that you are running the correct version of Python:

```
Python 3.7.6 (default, Jan  8 2020, 13:42:34)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc.
on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

8. Get the required `IBMQ` class:

```
>>> from qiskit import IBMQ
```

9. Install your API token locally:

```
>>> IBMQ.save_account('MY_API_TOKEN')
```

Here, instead of `MY_API_TOKEN`, paste in the API token that you just copied from IBM Quantum Experience®: Keep the single quotes as they are required for the command.

10. Load your account.

Now that the token is in place, let's verify that all is in order and that your account has the correct privileges:

```
>>> IBMQ.load_account()
```

This should display the following output:

```
<AccountProvider for IBMQ(hub='ibm-q', group='open',
project='main')>
```

This is the provider information for your account, with `hub`, `group`, and `project`.

How it works...

The main class that you import for this exercise is `IBMQ`, which is a toolbox for working with the quantum hardware and software that is provided by IBM in the cloud.

In this chapter, we used `save.account()` to store your account locally. As we go forward, in the recipes where we will access the IBM Quantum® machines, we will use the `IBMQ.load_account()` and `IBMQ.get_provider()` classes in your quantum programs to make sure that you have the correct access.

Updating your API key

If for some reason, you need to create a new API token on IBM Quantum Experience® and update your locally saved token, you can use the following command:

```
>>> IBMQ.save_account('NEW_API_TOKEN',
overwrite=True)
```

There's more...

In the code that follows in the recipes in this cookbook, we will set a `provider` variable to hold the provider information for your account by using the following command:

```
>>> provider = IBMQ.get_provider()
```

We can then use the `provider` information when selecting the IBM Quantum® computer, or backend, to run your quantum programs on. In the following example, we select a quantum computer that is called **IBM Q 5 Yorktown** (`ibmqx2`) as our backend. The internal reference for this machine is `ibmqx2`:

```
>>> backend = provider.get_backend('ibmqx2')
```

Keeping your Qiskit® environment up to date

Qiskit® is an open source programming environment that is in *continuous flux*. Over the course of writing this book, I have passed through many minor and major version updates of the software.

It is generally a good idea to stay updated with the latest version, but with some updates, components of the code might change behavior. It is always a good idea to have a good look at the release notes for each new version. Sometimes changes are introduced that will change the way your code behaves. In those cases, you might want to hold off on upgrading until you have verified that your code still works as expected.

If you are using Anaconda environments, then you can maintain more than one environment at different Qiskit® levels, to have a fallback environment in case an upgraded Qiskit® version breaks your code.

Qiskit® moves fast

The IBM Quantum Experience® Notebook environment always runs the latest version of Qiskit®, and it might be a good idea to test drive your code in that environment before you upgrade your local environment.

You can also subscribe to notification updates, to find out when a new release has been offered:

1. Log in to IBM Quantum Experience® at `https://quantum-computing.ibm.com/login`.
2. On the IBM Quantum Experience® dashboard, find your user icon in the upper-right corner, click it, and select **My account**.
3. On the account page, under **Notification** settings, set **Updates and new feature announcements** to **On**.

Getting ready

Before you begin, verify which version of Qiskit® you are running for each of your environments (if you have more than one).

For each environment, launch Python, either from the command line, from an IDE such as Spyder, or as a Jupyter notebook, then execute the following code:

```
>>> import qiskit
>>> qiskit.__qiskit_version__
```

If you have an old version of Qiskit® installed, the preceding code might result in the following output:

```
{'qiskit-terra': '0.9.0', 'qiskit-aer': '0.3.0', 'qiskit-ibmq-
provider': '0.3.0', 'qiskit-aqua': '0.6.0', 'qiskit': '0.12.0'}
```

You can then go to the Qiskit® release notes to find out if there is a more up-to-date version available: `https://qiskit.org/documentation/release_notes.html`

This is a lot of steps just to make sure. The whole process is automated in Python. To go down that path, go to the next section.

How to do it...

1. Activate your virtual environment:

```
$ conda activate environment_name
```

2. Run the following command to check for outdated `pip` packages for your virtual environment:

```
(environment_name) … $  pip list --outdated
```

3. This will return a list of all your `pip` packages that are currently outdated and list the available versions:

```
Example:
Package                  Version  Latest   Type
------------------------ -------- -------- -----
…
qiskit                   0.19.6   0.21.0   sdist
qiskit-aer               0.5.2    0.6.1    wheel
qiskit-aqua              0.7.3    0.7.5    wheel
qiskit-ibmq-provider     0.7.2    0.9.0    wheel
qiskit-ignis             0.3.3    0.4.0    wheel
qiskit-terra             0.14.2   0.15.1   wheel
…
```

4. It is then a breeze to update Qiskit® using `pip`:

```
(environment_name) … $  pip install qiskit --upgrade
```

5. From the command line, verify that Qiskit® is installed:

```
(environment_name)… $ pip show qiskit
```

This will result in an output similar to the following:

```
Name: qiskit
Version: 0.21.0
Summary: Software for developing quantum computing
programs
Home-page: https://github.com/Qiskit/qiskit
Author: Qiskit Development Team
Author-email: qiskit@us.ibm.com
License: Apache 2.0
Location: /Users/hassi/opt/anaconda3/envs/packt_qiskit/
lib/python3.7/site-packages
Requires: qiskit-aer, qiskit-terra, qiskit-aqua, qiskit-
```

```
ignis, qiskit-ibmq-provider
Required-by:
…
```

6. Verify that Qiskit® is integrated with Python in your isolated environment.

   Open Python:

   ```
   (environment_name)… $ python3
   ```

   Import Qiskit®:

   ```
   >>> import qiskit
   ```

   List the version details:

   ```
   >>> qiskit.__qiskit_version__
   ```

   This should display the versions of the installed Qiskit® components:

   ```
   {'qiskit-terra': '0.15.2', 'qiskit-aer': '0.6.1',
   'qiskit-ignis': '0.4.0', 'qiskit-ibmq-provider': '0.9.0',
   'qiskit-aqua': '0.7.5', 'qiskit': '0.21.0'}
   ```

Congratulations, your Qiskit® upgrade worked; you are now running the latest code!

How it works…

Depending on how you consume this book, you might be looking over this process as part of your first read-through, and no upgrades were available. If so, go ahead and bookmark this recipe, and come back to it at a later time when there has been a Qiskit® upgrade.

The pip tool will manage your upgrade for you for each virtual environment. As I mentioned before, it might be a good idea to do a staged upgrade of your environments if you have more than one.

For example, you can upgrade one environment and test run your quantum programs in that environment to make sure that the new version did not break anything in your code.

OK, with this you should be reasonably set with one or more Qiskit® environments on which to run your quantum programs. If you feel ready for it, you can now take the plunge and go directly to *Chapter 4*, *Starting at the Ground Level with Terra*, to start writing quantum programs in Python using Qiskit®. If you are up for some prep work to get a feel for what programming a quantum computer is all about, start with *Chapter 2*, *Quantum Computing and Qubits with Python*, to get an introduction to qubits and gates, or *Chapter 3*, *IBM Quantum Experience® – Quantum Drag and Drop*, to get a visual feel for quantum programs by using the IBM Quantum Experience® drag-and-drop programming interface.

No matter which path you take, don't worry, we'll let Python do the hard work. Again, have fun!

2
Quantum Computing and Qubits with Python

Quantum computing is a fairly new and fairly old field at the same time. The ideas and concepts used to achieve quantum computing (such as quantum mechanical superposition and entanglement) have been around for almost a century and the field of quantum information science was founded almost 40 years ago. Early explorers, such as Peter Shor and Lov Grover, produced quantum computing algorithms (Shor's algorithm and Grover's algorithm) that are now starting to become as well known as foundational physics concepts such as $E=mc^2$. For details, see the references at the end of the chapter.

At the same time, real quantum computers that utilize these effects are a relatively recent invention. The requirements for building one were outlined by DiVincenzo in the 1990, and IBM opened up its IBM Quantum Experience® and Qiskit® in 2016, effectively the first time anyone outside of a research lab could start exploring this nascent field for real.

So, what is the difference between classical computing and quantum computing? One way to start exploring is by taking a look at the basic computational elements used by each—the classical bits and the quantum qubits.

In this chapter, we will contrast bits and qubits, play with some generic linear algebra to explore them in more detail, and contrast deterministic (classical) computation and probabilistic (quantum) computation. We will even take a quick look at some basic Qiskit® presentation methods to visualize a qubit.

In this chapter, we will cover the following recipes:

- Comparing a bit and a qubit
- Visualizing a qubit in Python
- A quick introduction to quantum gates

Technical requirements

The recipes that we discuss in this chapter can be found here: `https://github.com/PacktPublishing/Quantum-Computing-in-Practice-with-Qiskit-and-IBM-Quantum-Experience/tree/master/Chapter02`.

For more information about how to get the recipe sample code, refer to the *Downloading the code samples* section in *Chapter 1*, *Preparing Your Environment*.

Comparing a bit and a qubit

So, let's start with the obvious—or perhaps, not so obvious—notion that most people who read this book know what a bit is.

An intuitive feeling that we have says that a bit is something that is either **zero** (**0**) or **one** (**1**). By putting many bits together, you can create bytes as well as arbitrary large binary numbers, and with those, build the most amazing computer programs, encode digital images, encrypt your love letters and bank transactions, and more.

In a classical computer, a bit is realized by using low or high voltages over the transistors that make up the logic board, typically something such as 0 V and 5 V. In a hard drive, the bit might be a region magnetized in a certain way to represent 0 and the other way for 1, and so on.

In books about quantum computing, the important point to drive home is that a classical bit can only be a 0 or a 1; it can never be anything else. In the computer example, you can imagine a box with an input and an output, where the box represents the program that you are running. With a classical computer (and I use the term classical here to indicate a binary computer that is not a quantum computer), the input is a string of bits, the output is another string of bits, and the box is a bunch of bits being manipulated, massaged, and organized to generate that output with some kind of algorithm. An important thing, again, to emphasize is that while in the box, the bits are still bits, always 0s or 1s, and nothing else.

A qubit, as we will discover in this chapter, is something quite different. Let's go explore.

Getting ready

As recipes go, this one isn't really that much to brag about. It is just a quick Python and NumPy implementation that defines a bit as a 2x1 matrix, or a vector representing 0 or 1. We also introduce the Dirac notation of $|0\rangle$, $|1\rangle$, and $a|0\rangle + b|1\rangle$ to represent our qubits. We then calculate the probability of getting various results when measuring the bits and qubits.

The Python file for the following recipe can be downloaded from here: `https://github.com/PacktPublishing/Quantum-Computing-in-Practice-with-Qiskit-and-IBM-Quantum-Experience/blob/master/Chapter02/ch2_r1_bits_qubits.py`.

How to do it...

1. Let's start by importing `numpy` and `math`, which we will need to do the calculations:

```
import numpy as np
from math import sqrt, pow
```

2. Create and print the bit and qubit vectors for 0, 1, $|0\rangle$, $|1\rangle$, and $a|0\rangle + b|1\rangle$ as `[1,0]`, `[0,1]`, `[1,0]`, `[0,1]`, and `[a,b]`, respectively:

```
# Define the qubit parameters for superposition
a = sqrt(1/2)
b = sqrt(1/2)
if round(pow(a,2)+pow(b,2),0)!=1:
    print("Your qubit parameters are not normalized.
        \nResetting to basic superposition")
    a = sqrt(1/2)
    b = sqrt(1/2)
bits = {"bit = 0":np.array([1,0]),
    "bit = 1":np.array([0,1]),
    "|0\u27E9":np.array([1,0]),
    "|1\u27E9":np.array([0,1]),
    "a|0\u27E9+b|1\u27E9":np.array([a,b])}
# Print the vectors
for b in bits:
  print(b, ": ", bits[b].round(3))
print ("\n")
```

Notice the Unicode entries here: `\u27E9`. We use this instead of just > to create the nice-looking Dirac qubit rendering |0⟩ in the output.

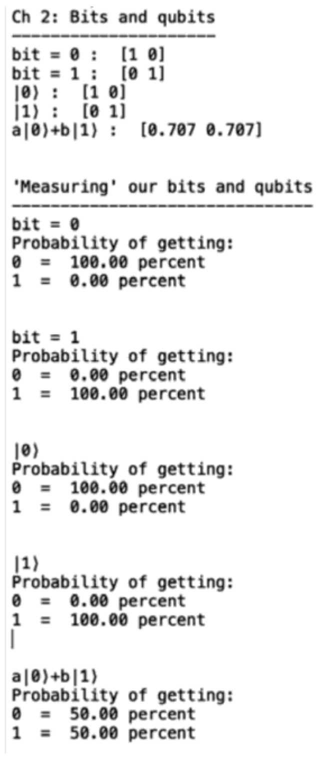
You must provide the correct a and b parameters

Note that the parameter verification code checks whether the values for `a` and `b` are *normalized*. If not, then `a` and `b` are reset to a simple 50/50 superposition by setting $a = \frac{1}{\sqrt{2}}$ and $b = \frac{1}{\sqrt{2}}$.

3. Measure the qubits by creating a measurement dictionary, and then calculate the probability of getting `0` and `1` from the bit vectors we created:

```
print("'Measuring' our bits and qubits")
print("-------------------------------")
prob={}
for b in bits:
    print(b)
    print("Probability of getting:")
    for dig in range(len(bits[b])):
        prob[b]=pow(bits[b][dig],2)
        print(dig, " = ", '%.2f'%(prob[b]*100), percent")
    print ("\n")
```

The preceding code should give the following output:



```
Ch 2: Bits and qubits
---------------------
bit = 0 :  [1 0]
bit = 1 :  [0 1]
|0⟩ :  [1 0]
|1⟩ :  [0 1]
a|0⟩+b|1⟩ :  [0.707 0.707]


'Measuring' our bits and qubits
-------------------------------
bit = 0
Probability of getting:
0  =  100.00 percent
1  =  0.00 percent


bit = 1
Probability of getting:
0  =  0.00 percent
1  =  100.00 percent


|0⟩
Probability of getting:
0  =  100.00 percent
1  =  0.00 percent


|1⟩
Probability of getting:
0  =  0.00 percent
1  =  100.00 percent


a|0⟩+b|1⟩
Probability of getting:
0  =  50.00 percent
1  =  50.00 percent
```

Figure 2.1 – Simulating bits and qubits with NumPy

Now we know what the probabilities of getting the values of `0` or `1` are when measuring the bits and qubits. For some of these (0, 1, $|0\rangle$, and $|1\rangle$) the outcome is what we expected, 0 or 100%; the bit or qubit is either 0 or 1 and nothing else. For one ($a|0\rangle + b|1\rangle$), which is a qubit that is in a superposition of 0 and 1, the probability of getting either is 50%. This is an outcome that can never happen for a classical bit, only for a qubit. We will explain why in the next section.

How it works...

What we have seen in this recipe is that the probability of reading a classical bit will always be *100%*, either 0 or 1; there are no other options. But for a qubit that can be expressed as $a|0\rangle + b|1\rangle$, the probability of a 0 or 1 is proportional to $|a|^2 + |b|^2 = 1$. For pure $|0\rangle$ and $|1\rangle$ states, *a* or *b* is always 1, and the probability of measuring each is 100%. But for the qubit that we labeled $a|0\rangle + b|1\rangle$, a and b are both $\frac{1}{\sqrt{2}}$, giving a probability of 50% for 0 or 1

$(|\frac{1}{\sqrt{2}}|^2 + |\frac{1}{\sqrt{2}}|^2 = 0.5 + 0.5 = 1)$.

Measuring a bit and a qubit

The word **measure** means two slightly different things in classical computing and quantum computing. In classical computing, you can measure your bits at any time without seriously disturbing the calculations that you are doing. For a quantum computer, measuring is a more definite act that results in your qubit reverting from a bit that behaves quantum-mechanically to a bit that behaves classically. After you measure a qubit, you are done. You can do no further quantum actions on that qubit.

Due to the quantum mechanical nature of a qubit, we can describe it as a vector similar to the vector we use for a bit. To clarify that, when we are talking about qubits, we do not just use 0 and 1 as labels, but rather the Dirac *ket* notation, $|0\rangle$ and $|1\rangle$, indicating that these are state vectors in a vector space.

We can write out the state vector, $|\psi\rangle$ (psi), of a qubit like this:

- $|\psi\rangle = |0\rangle$ for a qubit in the *ground state*, representing 0
- $|\psi\rangle = |1\rangle$ for a qubit in the *excited state*, representing 1

Here, we have used **ground state** and **excited state** as one way of categorizing qubits. This is appropriate as the Josephson junctions that the IBM Quantum® qubits use are quantum systems with two energy levels. Depending on the underlying physical system, qubits can also be based on other two-level quantum systems, such as electron spin (up or down) or photon polarization (horizontal or vertical).

So far, nothing is intuitively much different from classical bits, as each represents just the value 0 or 1. But now we add a complexity: a qubit can also be a superposition of the two states, $|0\rangle$ and $|1\rangle$.

$|\psi\rangle = a|0\rangle + b|1\rangle$, where *a* and *b* are complex numbers. These numbers are normalized so that $|a|^2 + |b|^2 = 1$, which geometrically means that the resulting vector, $|\psi\rangle$, has a length of 1. This is important!

Going back to the simplest cases, these can be described as follows:

- $|\psi\rangle = |0\rangle = a|0\rangle + b|1\rangle = 1|0\rangle + 0|1\rangle$ for a qubit in the ground state. In this case, a=1 and b=0.
- $|\psi\rangle = |1\rangle = a|0\rangle + b|1\rangle = 0|0\rangle + 1|1\rangle$ for a qubit in the excited state. In this case, a=0 and b=1.

So far, so good, but now we add the quantum twist: superposition. The following qubit state vector is also supported:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Just to check that we are still normalized, in this case,

$$|a|^2 + |b|^2 = \left|\frac{1}{\sqrt{2}}\right|^2 + \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2} + \frac{1}{2} = 1.$$

But what does this state vector mean?

The qubit is set up in a state where it is exactly halfway between $|0\rangle$ and $|1\rangle$; it exists in a superposition of the two basic states. It is behaving quantumly.

Important Note

The quantum superposition state of a qubit can only be sustained while we are doing calculations on the quantum computer. The same is true for an actual particle in nature, such as a photon that behaves quantum-mechanically. For example, the polarization of the photon can be described as a superposition of horizontal and vertical orientations while the photon is in flight, but when you add a polarizer in its path, you will measure it as either horizontal or vertical, and nothing else.

Going back to the computer-as-a-box example, for a quantum computer we have a similar image, a string of bits as input and another string of bits as output. The difference comes inside the box where the qubits can exist in superposition while we are doing our calculations.

As soon as we measure the qubits to get that string of output bits however, the qubits must decide, quantum-mechanically, if they are a $|0\rangle$ or a $|1\rangle$, and here is where those *a* and *b* parameters come in.

The $|a|^2 + |b|^2 = 1$ formula not only states that the vector is normalized to the length 1, but it also describes the probability of getting the $|0\rangle$ and $|1\rangle$ outputs. The probability of getting $|0\rangle$ is $|a|^2$, and $|1\rangle$ is $|b|^2$. This is the core of the difference between a quantum computer and a classical computer. The quantum computer is probabilistic—you cannot know in advance what the end result will be, only the probability of getting a certain result—whereas the classical computer is deterministic—you can always, at least in theory, predict what the answer will be.

About probabilistic computing

People often get a little confused about quantum computers and probabilistic outcomes and visualize the whole quantum programming concept as qubits spinning randomly and uncontrollably in all different states at the same time. This is not a true picture; each qubit is initialized in a specific known state, $|\psi\rangle$, and then acted upon by quantum gate manipulations. Each manipulation is strictly deterministic; there is nothing random. At each stage in a quantum state evolution, we know exactly what our qubit is doing, expressed as an $a|0\rangle + b|1\rangle$ superposition. It is only at the end, when we measure and force the qubit to be either 0 or 1, that the probabilistic nature shows with the probability of measuring 0 or 1 set by the *a* and *b* parameters ($|a|^2 + |b|^2 = 1$).

See also

For more information about qubits and how to interpret them, refer to the following excellent books:

- *Dancing with Qubits, How quantum computing works and how it can change the world*, Robert S. Sutor, Packt Publishing Ltd., 2019, *Chapter 7*, *One Qubit*
- *Quantum Computation and Quantum Information*, Isaac L. Chuang; Michael A. Nielsen, Cambridge University Press, 2010, *Chapter 1.2*, *Quantum bits*
- *Quantum Mechanics: The theoretical minimum*, Leonard Susskind & Art Friedman, Basic Books, 2015, *Lecture 1: Systems and experiments*
- *Shor, I'll do it*, Scott Aaronson's blog, `https://www.scottaaronson.com/blog/?p=208`
- *What's a Quantum Phone Book?*, Lov Grover, Lucent Technologies, `https://web.archive.org/web/20140201230754/http://www.bell-labs.com/user/feature/archives/lkgrover/`
- *The Physical Implementation of Quantum Computation*, David P. DiVincenzo, IBM, `https://arxiv.org/abs/quant-ph/0002077`

Visualizing a qubit in Python

In this recipe, we will use generic Python with NumPy to create a vector and visual representation of a bit and show how it can be in only two states, 0 and 1. We will also introduce our first, smallish foray into the Qiskit® world by showing how a qubit can not only be in the unique 0 and 1 states but also in a superposition of these states. The way to do this is to take the vector form of the qubit and project it on the so-called **Bloch sphere**, for which there is a Qiskit® method. Let's get to work!

In the preceding recipe, we defined our qubits with the help of two complex parameters—*a* and *b*. This meant that our qubits could take values other than the 0 and 1 of a classical bit. But it is hard to visualize a qubit halfway between 0 and 1, even if you know *a* and *b*.

However, with a little mathematical trickery, it turns out that you can also describe a qubit using two angles—**theta** (θ) and **phi** (φ)—and visualize the qubit on a Bloch sphere. You can think of the θ and φ angles much as the latitude and longitude of the earth. On the Bloch sphere, we can project any possible value that the qubit can take.

The equation for the transformation is as follows:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle$$

Here, we use the formula we saw before:

$$|\psi\rangle = a|0\rangle + b|1\rangle$$

a and *b* are, respectively, as follows:

$$a = \cos\left(\frac{\theta}{2}\right)$$

$$b = e^{i\varphi}\sin\left(\frac{\theta}{2}\right)$$

I'll leave the deeper details and math to you for further exploration:

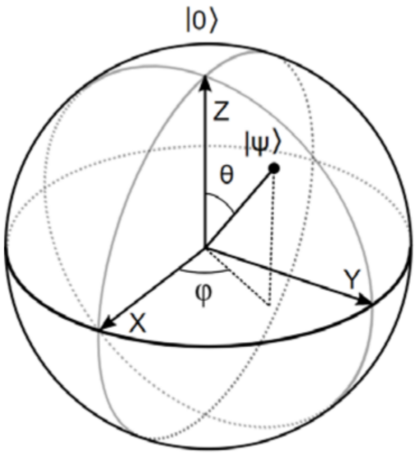


Figure 2.2 – The Bloch sphere.

The poor classical bits cannot do much on a Bloch sphere as they can exist at the North and South poles only, representing the binary values 0 and 1. We will include them just for comparison.

The reason 0 points up and 1 points down has peculiar and historical reasons. The qubit vector representation of $|0\rangle$ is $\begin{bmatrix}1\\0\end{bmatrix}$, or up, and $|1\rangle$ is $\begin{bmatrix}0\\1\end{bmatrix}$, or down, which is intuitively not what you expect. You would think that 1, or a more exciting qubit, would be a vector pointing upward, but this is not the case; it points down. So, I will do the same with the poor classical bits as well: 0 means up and 1 means down.

The latitude, the distance to the poles from a cut straight through the Bloch sphere, corresponds to the numerical values of *a* and *b*, with $\theta = 0$ pointing straight up for $|0\rangle$ (a=1, b=0), $\theta = \pi$ pointing straight down for $|1\rangle\,(a = 0, b = 1)$, and $\theta = \frac{\pi}{2}$ points to the equator for the basic superposition where $a = b = \frac{1}{\sqrt{2}}$.

So, what we are adding to the equation here is the *phase* of the qubit. The φ angle cannot be directly measured and has no impact on the outcome of our initial quantum circuits. Later on in the book, in *Chapter 9*, *Grover's Search Algorithm*, we will see that you can use the phase angle to great advantage in certain algorithms. But we are getting ahead of ourselves.

Getting ready

The Python file for the following recipe can be downloaded from here: `https://github.com/PacktPublishing/Quantum-Computing-in-Practice-with-Qiskit-and-IBM-Quantum-Experience/blob/master/Chapter02/ch2_r2_visualize_bits_qubits.py`.

How to do it...

For this exercise, we will use the θ and φ angles as latitude and longitude coordinates on the Bloch sphere. We will code the 0, 1, $|0\rangle$, $|1\rangle$, and $\frac{(|0\rangle + |1\rangle)}{\sqrt{2}}$ states with the corresponding angles. As we can set these angles to any latitude and longitude value we want, we can put the qubit state vector wherever we want on the Bloch sphere:

1. Import the classes and methods that we need, including `numpy` and `plot_bloch_vector` from Qiskit®. We also need to use `cmath` to do some calculations on imaginary numbers:

```
import numpy as np
import cmath
from math import pi, sin, cos
from qiskit.visualization import plot_bloch_vector
```

2. Create the qubits:

```
# Superposition with zero phase
angles={"theta": pi/2, "phi":0}
# Self defined qubit
#angles["theta"]=float(input("Theta:\n"))
#angles["phi"]=float(input("Phi:\n"))
# Set up the bit and qubit vectors
bits = {"bit = 0":{"theta": 0, "phi":0},
    "bit = 1":{"theta": pi, "phi":0},
    "|0\u27E9":{"theta": 0, "phi":0},
    "|1\u27E9":{"theta": pi, "phi":0},
    "a|0\u27E9+b|1\u27E9":angles}
```

From the code sample, you can see that we are using the theta angle only for now, with theta = 0 meaning that we point straight up and theta = π meaning straight down for our basic bits and qubits: 0, 1, $|0\rangle$, and $|1\rangle$. Theta $\frac{\pi}{2}$ takes us halfway, to the equator, and we use that for the superposition qubit, $a|0\rangle + b|1\rangle$.

3. Print the bits and qubits on the Bloch sphere.

The Bloch sphere method takes a three-dimensional vector as input, but we have to build the vector first. We can use the following formula to calculate the *X*, *Y*, and *Z* parameters to use with `plot_bloch_vector` and display the bits and qubits as Bloch sphere representations:

$$\text{bloch} = (\cos\varphi\sin\theta\,, \sin\varphi\sin\theta\,, \cos\theta)$$

This is how it would look in our vector notation:

$$\text{bloch} = \begin{bmatrix} \cos\varphi\sin\theta \\ \sin\varphi\sin\theta \\ \cos\theta \end{bmatrix}$$

And this is how we set this up in Python:

```
bloch=[cos(bits[bit]["phi"])*sin(bits[bit]
    ["theta"]),sin(bits[bit]["phi"])*sin(bits[bit]
    ["theta"]),cos(bits[bit]["theta"])]
```

We now cycle through the bits dictionary to display the Bloch sphere view of the bits and qubits, as well as the state vectors that correspond to them:

$$|\psi\rangle = a|0\rangle + b|1\rangle = \begin{bmatrix} a \\ b \end{bmatrix}$$

The state vector is calculated using the equation we saw previously:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle$$

What we see now is that *a* and *b* can actually turn into complex values, just as defined.

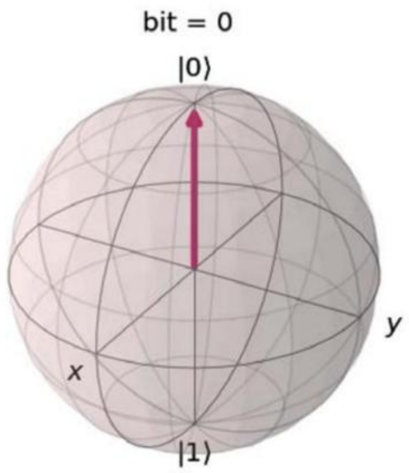
Here's the code:

```
for bit in bits:
     bloch=[cos(bits[bit]["phi"])*sin(bits[bit]
        ["theta"]),sin(bits[bit]["phi"])*sin(bits[bit]
        ["theta"]),cos(bits[bit]["theta"])]
    display(plot_bloch_vector(bloch, title=bit))
    # Build the state vector
    a = cos(bits[bit]["theta"]/2)
    b = cmath.exp(bits[bit]["phi"]*1j)*sin(bits[bit]
        ["theta"]/2)
    state_vector = [a * complex(1, 0), b * complex(1, 0)]
    print("State vector:", np.around(state_vector,
        decimals = 3))
```

4. The sample code should give an output similar to the following examples.

   First, we show the classical bits, 0 and 1:

```
Ch 2: Bloch sphere visualization of bits and qubits
---------------------------------------------------
```



```
State vector: [1.+0.j 0.+0.j]
```

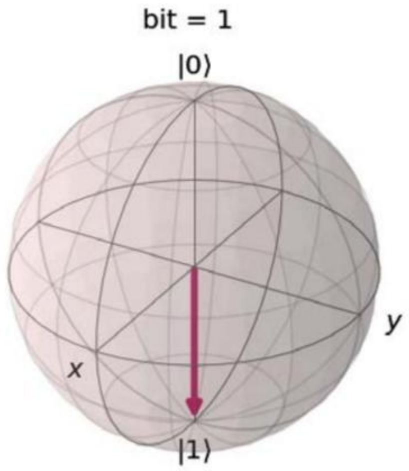
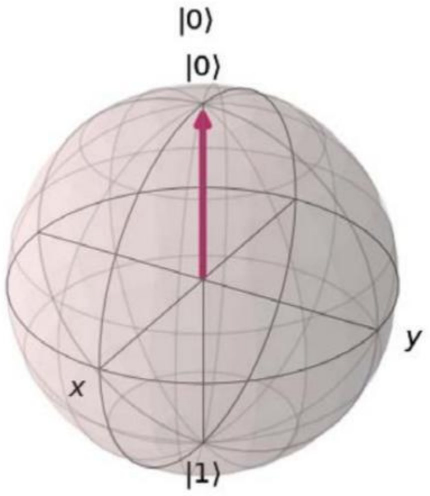
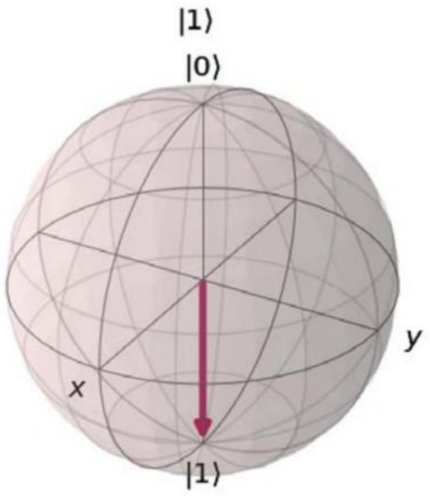


Figure 2.3 – Bloch sphere visualization of classical bits

5. Then, we show the quantum bits, or qubits, $|0\rangle$ and $|1\rangle$:



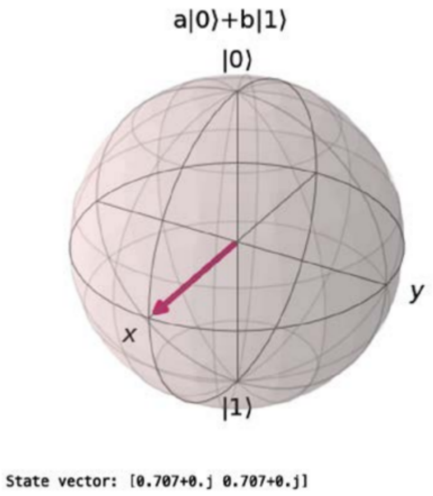
```
State vector: [1.+0.j 0.+0.j]
```



```
State vector: [0.+0.j 1.+0.j]
```

Figure 2.4 – Bloch sphere visualization of qubits

6. Finally, we show a qubit in superposition, a mix of $|0\rangle$ and $|1\rangle$:



```
State vector: [0.707+0.j 0.707+0.j]
```

Figure 2.5 – Bloch sphere visualization of a qubit in superposition

So, there is very little that is earth-shattering with the simple 0, 1, $|0\rangle$, and $|1\rangle$ displays. They simply point up and down to the north and south pole of the Bloch sphere as appropriate. If we check what the value of the bit or qubit is by measuring it, we will get 0 or 1 with 100% certainty.

The qubit in superposition, calculated as $\frac{|0\rangle + |1\rangle}{\sqrt{2}}$, on the other hand, points to the equator. From the equator, it is an equally long distance to either pole, thus a 50/50 chance of getting a 0 or a 1.

In the code, we include the following few lines, which define the `angles` variable that sets θ and φ for the $a|0\rangle + b|1\rangle$ qubit:

```
# Superposition with zero phase
angles={"theta": pi/2, "phi":0}
```

There's more...

We mentioned earlier that we weren't going to touch on the phase (φ) angle, at least not initially. But we can visualize what it does for our qubits. Remember that we can directly describe *a* and *b* using the angles θ and φ.

To test this out, you can uncomment the lines that define the angles in the sample code:

```
# Self-defined qubit
angles["theta"]=float(input("Theta:\n"))
angles["phi"]=float(input("Phi:\n"))
```

You can now define what your third qubit looks like by manipulating the θ and φ values. Let's test what we can do by running the script again and plugging in some angles.

For example, we can try the following:

$$\theta = \frac{\pi}{2} \approx 1.570$$

$$\varphi = \frac{\pi}{8} \approx 0.392$$

You should see the final Bloch sphere look something like the following:

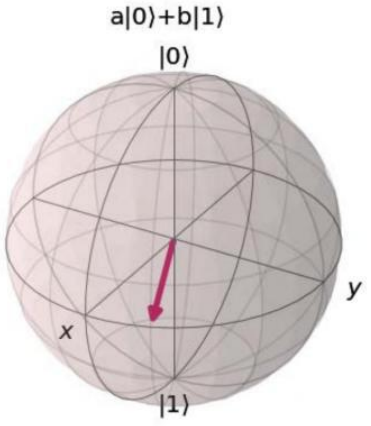


Figure 2.6 – The qubit state vector rotated by $\frac{\pi}{8}$

Note how the state vector is still on the equator with $\theta = \frac{\pi}{2}$ but now at a $\frac{\pi}{8}$ angle to the *x* axis. You can also take a look at the state vector: [0.707+0.j 0.653+0.271j].

We have now stepped away from the Bloch sphere prime meridian and out into the complex plane, and added a phase angle, which is represented by an imaginary state vector component along the *y* axis: $|\psi\rangle = 0.707|0\rangle + (0.653 + 0.271i)|1\rangle$

Let's go on a trip

Go ahead and experiment with different θ and φ angles to get other *a* and *b* entries and see where you end up. No need to include 10+ decimals for these rough estimates, two or three decimals will do just fine. Try plotting your hometown on the Bloch sphere. Remember that the script wants the input in radians and that theta starts at the North Pole, not at the equator. For example, the coordinates for Greenwich Observatory in England are 51.4779° N, 0.0015° W, which translates into: $\theta = 0.6723, \varphi = 0.0003$.

Here's Qiskit® and a globe displaying the same coordinates:

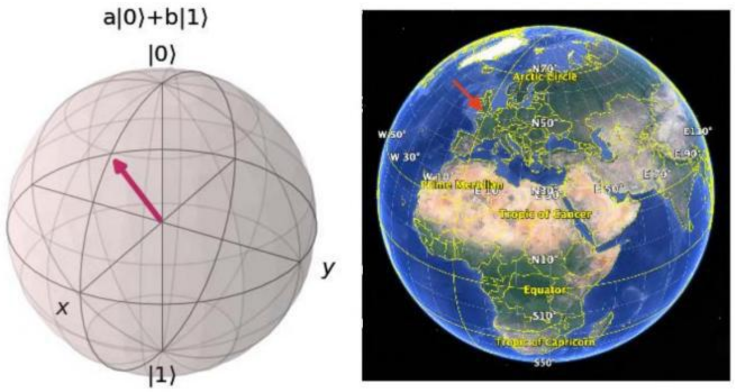


Figure 2.7 – Greenwich quantumly and on a globe

See also

Quantum Computation and Quantum Information, Isaac L. Chuang; Michael A. Nielsen, Cambridge University Press, 2010, *Chapter 1.2, Quantum bits*, and *Chapter 4.2, Single qubit operations.*

A quick introduction to quantum gates

Now that we have sorted out the difference between bits and qubits, and have also understood how to visualize the qubit as a Bloch sphere, we know all that there is to know about qubits, correct? Well, not quite. A qubit, or for that matter, hundreds or thousands of qubits, is not the only thing you need to make a quantum computer! You need to perform logical operations on and with the qubits. For this, just like a classical computer, we need logical gates.

I will not go into any great detail on how logical gates work, but suffice to say that a quantum gate, operates on the input of one or more qubits and outputs a result.

In this recipe, we will work our way through the mathematical interpretation of few quantum gates by using matrix multiplication of single- and multi-qubit gates. Don't worry, we will not dig deep, just a little to scratch the surface. You will find a deeper look quantum gates in *Chapter 6*, *Understanding the Qiskit Gate Library*.

Again, we will not be building any actual Qiskit quantum circuits just yet. We are still using more or less plain Python with some NumPy matrix manipulations to prove our points.

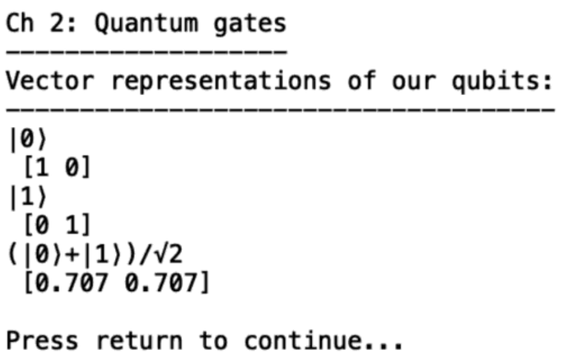
Getting ready

The Python file for the following recipe can be downloaded from here: `https://github.com/PacktPublishing/Quantum-Computing-in-Practice-with-Qiskit-and-IBM-Quantum-Experience/blob/master/Chapter02/ch2_r3_qubit_gates.py`.

How to do it...

This recipe will create vector and matrix representations of qubits and gates, and use simple algebra to illustrate the behavior of the qubits as gates are applied to them:

1. In your Python environment, run `ch2_r3_qubit_gates.py` and respond to the **Press return to continue** prompts to move along the program.
2. First, we see the vector representations of three qubit states: $|0\rangle$, $|1\rangle$, and $\frac{|0\rangle + |1\rangle}{\sqrt{2}}$:

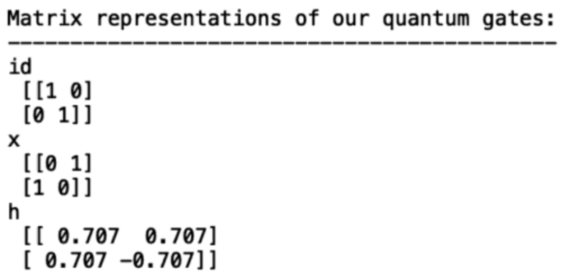


```
Ch 2: Quantum gates
-------------------
Vector representations of our qubits:
-------------------------------------
|0⟩
 [1 0]
|1⟩
 [0 1]
(|0⟩+|1⟩)/√2
 [0.707 0.707]

Press return to continue...
```

Figure 2.8 – Qubits as vectors

3. Next, we display the matrix representation of a couple of gates.

   We will use the `Id` (does nothing), `X` (flips the qubit), and `H` (creates a superposition) gates:

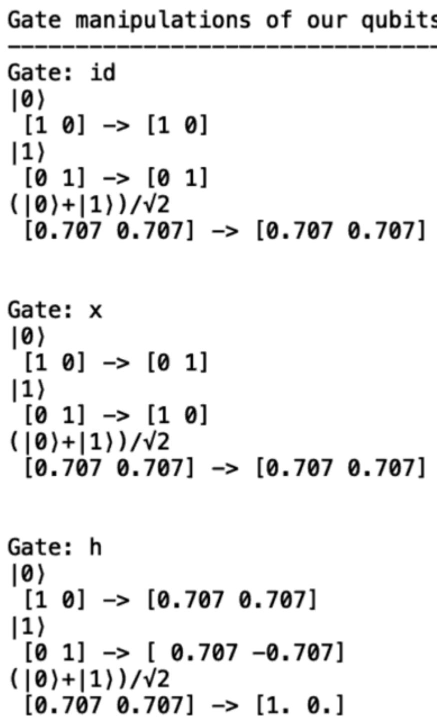


```
Matrix representations of our quantum gates:
--------------------------------------------
id
 [[1 0]
 [0 1]]
x
 [[0 1]
 [1 0]]
h
 [[ 0.707  0.707]
 [ 0.707 -0.707]]
```

Figure 2.9 – Single-qubit gates as matrices

4. The final step for our single-qubit setup is to see how each gate manipulates the qubits.

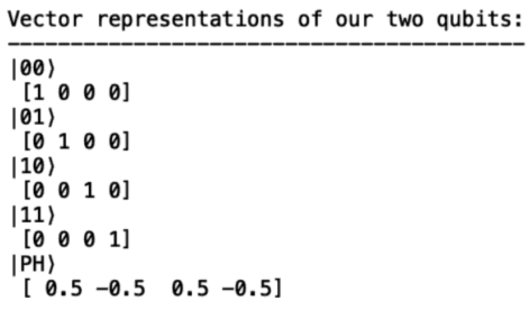
   This is done using matrix multiplication of the qubit vector and the gate matrix:



```
Gate manipulations of our qubits:
---------------------------------
Gate: id
|0⟩
 [1 0] -> [1 0]
|1⟩
 [0 1] -> [0 1]
(|0⟩+|1⟩)/√2
 [0.707 0.707] -> [0.707 0.707]


Gate: x
|0⟩
 [1 0] -> [0 1]
|1⟩
 [0 1] -> [1 0]
(|0⟩+|1⟩)/√2
 [0.707 0.707] -> [0.707 0.707]


Gate: h
|0⟩
 [1 0] -> [0.707 0.707]
|1⟩
 [0 1] -> [ 0.707 -0.707]
(|0⟩+|1⟩)/√2
 [0.707 0.707] -> [1. 0.]
```

Figure 2.10 – Gates acting on qubits

5. With the single qubits done, we now move on to working with combinations of two qubit states: |00⟩, |01⟩, |10⟩, and |11⟩:

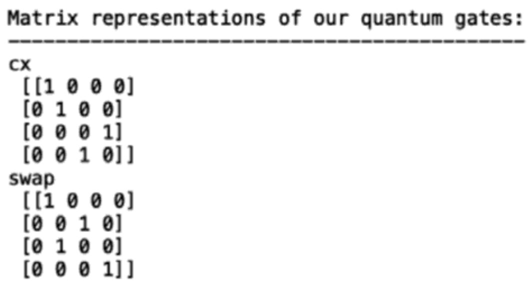


```
Vector representations of our two qubits:
-----------------------------------------
|00⟩
 [1 0 0 0]
|01⟩
 [0 1 0 0]
|10⟩
 [0 0 1 0]
|11⟩
 [0 0 0 1]
|PH⟩
 [ 0.5 -0.5  0.5 -0.5]
```

Figure 2.11 – Two qubits as vectors

6. Like for the single qubits, we now show the matrix representations of the two-qubit quantum gates.

   Here, we use `CX` (controlled NOT, flips one qubit if the other is 1) and `swap` (swaps the values of the two qubits):

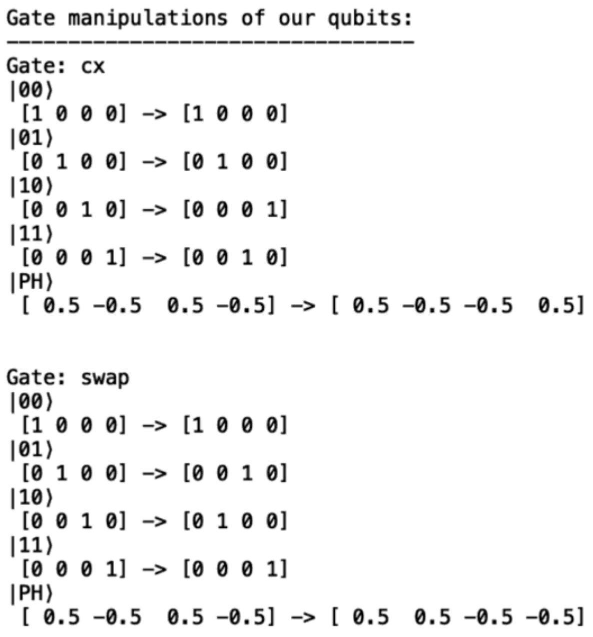


```
Matrix representations of our quantum gates:
--------------------------------------------
cx
 [[1 0 0 0]
 [0 1 0 0]
 [0 0 0 1]
 [0 0 1 0]]
swap
 [[1 0 0 0]
 [0 0 1 0]
 [0 1 0 0]
 [0 0 0 1]]
```

Figure 2.12 – Two-qubit gates as matrices

7. Finally, let's see gate manipulations of our multi-qubit states.

   Again, we have a matrix multiplication of the qubits vector and the gate matrix:



```
Gate manipulations of our qubits:
----------------------------------
Gate: cx
|00⟩
 [1 0 0 0] -> [1 0 0 0]
|01⟩
 [0 1 0 0] -> [0 1 0 0]
|10⟩
 [0 0 1 0] -> [0 0 0 1]
|11⟩
 [0 0 0 1] -> [0 0 1 0]
|PH⟩
 [ 0.5 -0.5  0.5 -0.5] -> [ 0.5 -0.5 -0.5  0.5]


Gate: swap
|00⟩
 [1 0 0 0] -> [1 0 0 0]
|01⟩
 [0 1 0 0] -> [0 0 1 0]
|10⟩
 [0 0 1 0] -> [0 1 0 0]
|11⟩
 [0 0 0 1] -> [0 0 0 1]
|PH⟩
 [ 0.5 -0.5  0.5 -0.5] -> [ 0.5  0.5 -0.5 -0.5]
```

Figure 2.13 – Multi-qubit gates acting on two qubits

That's it… we have now witnessed Python-generated linear algebra that describes how our qubits are defined and how they behave when gates are applied to them.

How it is works...

The previous section contained a lot of printed information with very little explanation of how we got those results. Let's dig into the sample code to see how the output is generated:

Tip

The numbered steps that follow correspond to the same numbered steps in the preceding *How to do it…* section. Refer back to those steps to see the result of the code samples that follow.

1. Let's start by importing the math tools we need:

```
import numpy as np
from math import sqrt
```

2. Set up the basic vectors for our qubits.

A qubit set to the value 0 is labeled $|0\rangle$ in the Dirac ket notation and is mathematically represented by the $\begin{bmatrix}1\\0\end{bmatrix}$ vector, if it is set to 1 as $|1\rangle$, or the $\begin{bmatrix}0\\1\end{bmatrix}$ vector. So far, so good, still only 0 and 1. As we have seen, the real magic comes when you have a qubit set to a superposition value represented by a vector pointing to the equator of the Bloch sphere, or anywhere except the poles—for example, $\frac{|0\rangle + |1\rangle}{\sqrt{2}}$, which would be represented by the following vector:

$$\begin{bmatrix}\frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}}\end{bmatrix}$$

The generic way of writing a qubit

Remember, the generic way of writing a qubit in superposition is $|\psi\rangle = a|0\rangle + b|1\rangle$, with *a* and *b* as complex numbers and $|a|^2 + |b|^2 = 1$.

This is how we create the $|0\rangle$ qubit vector using NumPy:

```
np.array([1,0])
```

Here's how we create a dictionary of our qubits in the sample:

```
qubits = {"|0\u27E9":np.array([1,0]),
    "|1\u27E9":np.array([0,1]),
    "(|0\u27E9+|1\u27E9)/\u221a2":1/sqrt(2)*np.
    array([1,1])}
for q in qubits:
  print(q, "\n", qubits[q].round(3))
```

3. Set up the basic matrices for our quantum gates.

   For qubits, any single-qubit gate can be represented by a 2x2 matrix like this: $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$.

   For single qubits, the math that we have implemented is a matrix operation that corresponds to the truth table for the two operations, *ID* and *NOT* (or *X*, as the quantum gate is called):

$$\text{ID: } \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$
$$\text{X: } \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

   But here, we also add another example, the *H* (or Hadamard) gate that does something completely new:

$$\text{H: } \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

   When we run $|0\rangle$ through an H gate, you get the superposition result:

$$0.707|0\rangle + 0.707|1\rangle$$

   The same is true when you run $|1\rangle$ through the gate, but with the sign flipped for the $|1\rangle$ component:

$$0.707|0\rangle - 0.707|1\rangle$$

   Finally, when you run a qubit in the generic 50/50 superposition ($|\psi\rangle = 0.707|0\rangle + 0.707|1\rangle$) through the H gate, you get one of the base qubits back:

$$|0\rangle \text{ or } |1\rangle$$

   This is our first tangible demonstration of another aspect of quantum gates that we'll touch on in the *There's more…* section. They are reversible. This means that no information is lost as you apply a gate to your qubit. You can always run the gate backward and end up where you started by applying its inverse.

   Here's how you create the X gate as a NumPy matrix:

```
np.array([[0, 1], [1, 0]])
```

Here's how we create a dictionary of our gates in the sample:

```
gates ={"id":np.array([[1, 0], [0, 1]]),
    "x":np.array([[0, 1], [1, 0]]),
    "h":1/sqrt(2)*np.array([[1, 1], [1, -1]])}
for g in gates:
  print(g, "\n", gates[g].round(3))
```

4. Now, let's use NumPy to apply the defined gates on our qubits.

   The application of a gate on a qubit can be expressed as a vector multiplication of the qubit and the gate. Here's the NumPy matrix dot multiplication for an X gate on the $|0\rangle$ qubit:

   ```
   np.dot(np.array([[0, 1], [1, 0]]), np.array([1,0]))
   ```

   In our sample, we step our way through the two dictionaries that we created, applying the matrix multiplication to each gate/qubit combination:

   ```
   for g in gates:
       print("Gate:",g)
       for q in qubits:
           print(q,"\n",qubits[q].round(3),"->",
               np.dot(gates[g],qubits[q]).round(3))
   ```

   Here, we see the expected behavior of the gates on our qubits: the ID gate does nothing, the X gate flips the qubit, and the H gate creates or uncreates a superposition.

   If you want to experiment a little, you can take a look at the vector representations of the various quantum gates that we show in *Chapter 6, Understanding the Qiskit® Gate Library*, and see whether you can add these gates to the **gates** dictionary.

   In this first example, we took a look at how to build our single qubits and gates as vectors and matrices, and how to run our qubits through the gates by using vector multiplication. Now let's do the same with two qubits…

5. Set up our multi-qubit vectors.

   First, we expand our Dirac-noted qubit combinations: $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$.

These represent, respectively, both qubits 0, first qubit 1 and second 0, first qubit 0 and second 1, and both qubits 1. Here, we are using the **backward Qiskit® notation** for our qubits, starting with the first qubit (q_0) as the **Least Significant Bit** (**LSB**) in the vector notation, like this: $|q_1\ q_0\rangle$.

The vector representation of these are, respectively, as follows:

$$\begin{bmatrix}1\\0\\0\\0\end{bmatrix},\begin{bmatrix}0\\1\\0\\0\end{bmatrix},\begin{bmatrix}0\\0\\1\\0\end{bmatrix},\text{ and }\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

We already know how to build our qubits as 2x1 NumPy arrays, so let's extend that to 4x1 vectors. With NumPy, this is how we create, for example, the $|00\rangle$ qubit vector:

```
np.array([1,0,0,0])
```

In the sample code, we set up a dictionary with the multi-qubit arrays:

```
twoqubits = {"|00\u27E9":np.array([1,0,0,0]),
    "|01\u27E9":np.array([0,1,0,0]),
    "|10\u27E9":np.array([0,0,1,0]),
    "|11\u27E9":np.array([0,0,0,1]),
    "|PH\u27E9":np.array([0.5,-0.5,0.5,-0.5])}
for b in twoqubits:
  print(b, "\n", twoqubits[b])
```

6. Set up our multi-qubit gate matrices.

The two-qubit quantum gates are represented by 4x4 matrices, such as the **controlled-NOT** (**CX**) gate, which flips the first qubit (q_0) if the controlling second qubit (q_1) is set to 1:

$$\text{CX:}\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$$

Gate matrices like these, where one qubit acts as the control and the other as controlled, differ somewhat depending on which qubit you select as the control. If the CX gate points the other way, with the first qubit (q_0) as the controlling qubit, the matrix will look like this instead:

$$\text{CX} = \begin{bmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{bmatrix}$$

Here's how we build the gates:

```
twogates ={"cx":np.array([[1, 0, 0, 0], [0, 1, 0, 0],
    [0, 0, 0, 1], [0, 0, 1, 0]]),
    "swap":np.array([[1, 0, 0, 0], [0, 0, 1, 0],
    [0, 1, 0, 0], [0, 0, 0, 1]])}
```

Here's a NumPy matrix dot multiplication example for a CX gate on the $|11\rangle$ qubit:

```
np.dot(np.array([[1, 0, 0, 0], [0, 1, 0, 0],
    [0, 0, 0, 1], [0, 0, 1, 0]]), np.array([0,0,0,1]))
```

Here's the sample code:

```
twogates ={"cx":np.array([[1, 0, 0, 0], [0, 1, 0, 0],
    [0, 0, 0, 1], [0, 0, 1, 0]]),
    "swap":np.array([[1, 0, 0, 0], [0, 0, 1, 0],
    [0, 1, 0, 0], [0, 0, 0, 1]])}
for g in twogates:
  print(g, "\n", twogates[g].round())
print ("\n")
```

7. Then, we'll apply the gates to our bits and see the results:

```
for g in twogates:
    print("Gate:",g)
    for b in twoqubits:
        print(b,"\n",twoqubits[b],"->",
            np.dot(twogates[g],twoqubits[b]))
    print("\n")
```

The main takeaway with the multi-qubit matrix manipulations is that the output is a vector of the same dimensions as the input vector; no information is lost.

There's more...

One other aspect of quantum gates that is generally not true of classical gates is that they are reversible. If you run the gate backward, you end up with the input states of your qubits, and no information is lost. The final recipe in this chapter illustrates this.

The sample code

The sample file for the following recipe can be downloaded from here: `https://github.com/PacktPublishing/Quantum-Computing-in-Practice-with-Qiskit-and-IBM-Quantum-Experience/blob/master/Chapter02/ch2_r4_reversible_gates.py`:

1. Let's start by importing all we need:

```
import numpy as np
from math import sqrt
```

2. Set up the basic qubit vectors and gate matrices. When printing out the gates, we compare the gate matrix with its complex conjugate. If these are the same, the gate and its inverse are identical:

```
qubits = {"|0\u232A":np.array([1,0]),
    "|1\u232A":np.array([0,1]),
    "(|0\u232A+|1\u232A)/\u221a2":1/sqrt(2)*np.
    array([1,1])}
for q in qubits:
  print(q, "\n", qubits[q])
print ("\n")
gates ={"id":np.array([[1, 0], [0, 1]]),
    "x":np.array([[0, 1], [1, 0]]),
    "y":np.array([[0, -1.j], [1.j, 0]]),
    "z":np.array([[1, 0], [0, -1]]),
    "h":1/sqrt(2)*np.array([[1, 1], [1, -1]]),
    "s":np.array([[1, 0], [0, 1j]])}
diff=""
for g in gates:
  print(g, "\n", gates[g].round(3))
  if gates[g].all==np.matrix.conjugate(gates[g]).all:
      diff="(Same as original)"
  else:
      diff="(Complex numbers conjugated)"
  print("Inverted",g, diff, "\n",
    np.matrix.conjugate(gates[g]).round(3))
print ("\n")
```

3. Demonstrate that the basic quantum gates are reversible by applying the gate then its complex conjugate, and then comparing the outcome with the input. For the quantum gates, which are reversible, this will bring the qubit back to the starting state:

```
for g in gates:
    input("Press enter...")
    print("Gate:",g)
    print("-------")
    for q in qubits:
        print ("\nOriginal qubit: ",q,"\n",
            qubits[q].round(3))
        print ("Qubit after",g,"gate: \n",
            np.dot(gates[g],qubits[q]).round(3))
        print ("Qubit after inverted",g,"gate.","\n",
            np.dot(np.dot(gates[g],qubits[q]),
            np.matrix.conjugate(gates[g])).round(3))
    print("\n")
```

Running the sample

When you run this `ch2_r4_reversible_gates.py` script, it will do the following:

1. Like before, create and print out vector and matrix representations of our qubits and quantum gates.

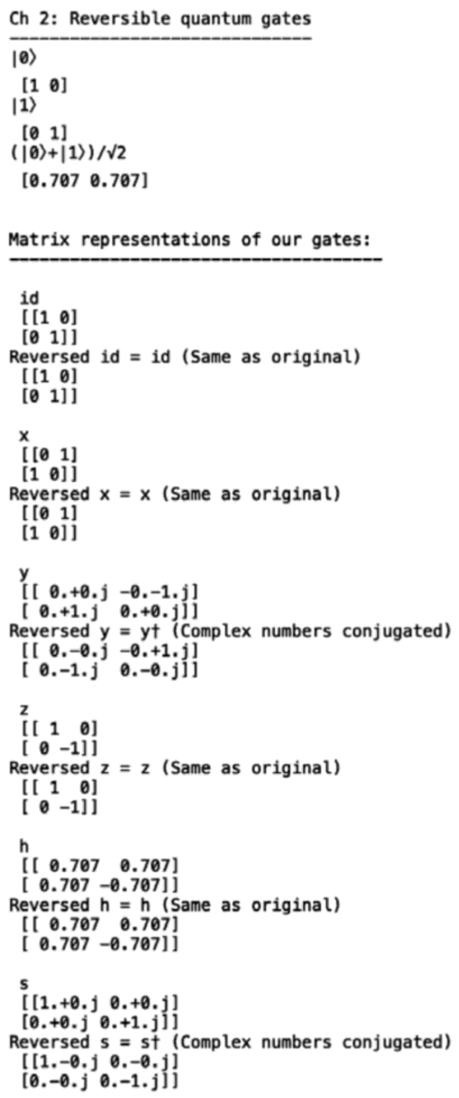
   This time, we add three new gates:

$$Y: \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

$$Z: \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$S: \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

   Here, *Y* and *Z* perform π rotation around the corresponding axes, in essence acting as NOT gates along the *y* and *z* axes on the Bloch sphere. The S gate adds a new functionality to the gates, $\pi/2$ rotation around the *z* axis. We will return to these gates in more detail in *Chapter 6*, *Understanding the Qiskit® Gate Library*:



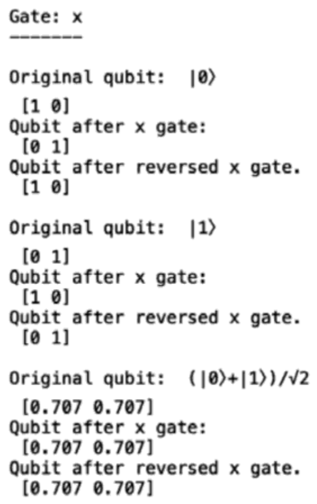
```
Ch 2: Reversible quantum gates
------------------------------
|0⟩
 [1 0]
|1⟩
 [0 1]
(|0⟩+|1⟩)/√2
 [0.707 0.707]


Matrix representations of our gates:
-------------------------------------

 id
 [[1 0]
 [0 1]]
Reversed id = id (Same as original)
 [[1 0]
 [0 1]]

 x
 [[0 1]
 [1 0]]
Reversed x = x (Same as original)
 [[0 1]
 [1 0]]

 y
 [[ 0.+0.j -0.-1.j]
 [ 0.+1.j  0.+0.j]]
Reversed y = y† (Complex numbers conjugated)
 [[ 0.-0.j -0.+1.j]
 [ 0.-1.j  0.-0.j]]

 z
 [[ 1  0]
 [ 0 -1]]
Reversed z = z (Same as original)
 [[ 1  0]
 [ 0 -1]]

 h
 [[ 0.707  0.707]
 [ 0.707 -0.707]]
Reversed h = h (Same as original)
 [[ 0.707  0.707]
 [ 0.707 -0.707]]

 s
 [[1.+0.j 0.+0.j]
 [0.+0.j 0.+1.j]]
Reversed s = s† (Complex numbers conjugated)
 [[1.-0.j 0.-0.j]
 [0.-0.j 0.-1.j]]
```

Figure 2.14 – Quantum gates and their inverses

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part, so for gates with only real numbers in their matrices, the complex conjugate does nothing, and the gate is its own reverse.

2. Then, for each of our qubits, we apply each gate and then its reverse gate and show that we end up with the same qubit state as we started with.

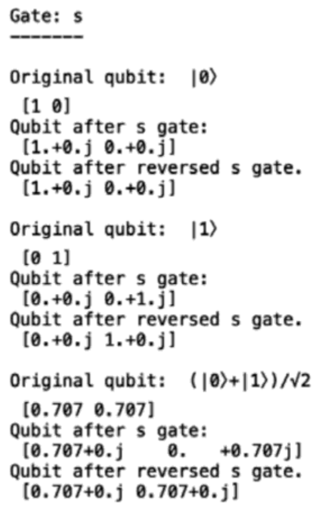
   The examples that follow are for the X and S gates:



```
Gate: x
-------

Original qubit:  |0⟩
 [1 0]
Qubit after x gate:
 [0 1]
Qubit after reversed x gate.
 [1 0]

Original qubit:  |1⟩
 [0 1]
Qubit after x gate:
 [1 0]
Qubit after reversed x gate.
 [0 1]

Original qubit:  (|0⟩+|1⟩)/√2
 [0.707 0.707]
Qubit after x gate:
 [0.707 0.707]
Qubit after reversed x gate.
 [0.707 0.707]
```

Figure 2.15 – The effects of the X gate and reversed X gate on three qubit states

The reversed X gate is simply itself, and applying it twice to a qubit brings back the original qubit state:



```
Gate: s
-------

Original qubit:  |0⟩
 [1 0]
Qubit after s gate:
 [1.+0.j 0.+0.j]
Qubit after reversed s gate.
 [1.+0.j 0.+0.j]

Original qubit:  |1⟩
 [0 1]
Qubit after s gate:
 [0.+0.j 0.+1.j]
Qubit after reversed s gate.
 [0.+0.j 1.+0.j]

Original qubit:  (|0⟩+|1⟩)/√2
 [0.707 0.707]
Qubit after s gate:
 [0.707+0.j    0.   +0.707j]
Qubit after reversed s gate.
 [0.707+0.j 0.707+0.j]
```

Figure 2.16 – The effects of the S and reversed S gate ($\mathbf{S}^\dagger$) on three qubit states

The reverse of the S gate is called the $S^\dagger$ gate, where $S^\dagger$ is the complex conjugation of S. Applying S followed by $S^\dagger$ brings back the original qubit state.

See also

- *Quantum Computation and Quantum Information*, Isaac L. Chuang; Michael A. Nielsen, Cambridge University Press, 2010, *Chapter 4.2, Single qubit operations*, and *Chapter 4.3, Controlled operations*.
- *The Feynman Lectures on Physics*, Feynman, Richard P.; Leighton, Robert B.; Sands, Matthew, 1965, Addison-Wesley. Take a look at the online version, and the chapter on amplitudes and vectors, for more about Dirac notation: `https://www.feynmanlectures.caltech.edu/III_08.html#Ch8-S1`.
- For a quick interactive look at a single qubit Bloch sphere representation, take a look at the **grok bloch** application by Qiskit Advocate James Weaver: `https://github.com/JavaFXpert/grok-bloch`.
- You can install and run it from your own Python environment, or run it online here: `https://javafxpert.github.io/grok-bloch/`.
- The application supports the simple X and H gates that we have tested so far, as well as additional gates that we will be touching on in the following chapters, such as Y, Z, Rx, Ry, Rz, and more. For a deeper dive into the quantum gates that are available with Qiskit®, refer to *Chapter 6, Understanding the Qiskit® Gate Library*.

3
IBM Quantum Experience® – Quantum Drag and Drop

Something pretty amazing happened in the cloud in early 2016; a new type of computer opened its arms to the world—a programmable quantum computer.

In this chapter, we will talk briefly about the early history of IBM Quantum Experience®, how to get there, and how to open a user account. We will take a look at the drag-and-drop user interface for programming the IBM quantum computers (Circuit Composer).

Also, we will take a quick peek at how you can move back and forth between IBM Quantum Experience® and Qiskit® by using the underlying OpenQASM coding.

In this chapter, we will cover the following recipes:

- Introducing IBM Quantum Experience®
- Building quantum scores with Circuit Composer

- Tossing a quantum coin
- Moving between worlds

We will not stay long here; just long enough to scratch the surface, present a quantum circuit we will play with later in *Chapter 4*, *Starting at the Ground Level with Terra*, and get a feel for the plethora of gates that are available to use.

Technical requirements

The quantum programs that we will discuss in this chapter can be found here: `https://github.com/PacktPublishing/Quantum-Computing-in-Practice-with-Qiskit-and-IBM-Quantum-Experience/tree/master/Chapter03`.

If you haven't already, get yourself an IBM Quantum Experience® account. For information, see *Creating your IBM Quantum Experience® account* in *Chapter 1*, *Preparing Your Environment*.

Introducing IBM Quantum Experience®

IBM Quantum Experience® is an open platform available for someone to start their quantum computing journey. In it, you have free access to a number of IBM quantum computers, ranging in size from a single qubit to 15 qubits (at the time of writing), as well as a 32-qubit simulator that runs on IBM POWER9™ hardware. That's a lot of power at your fingertips.

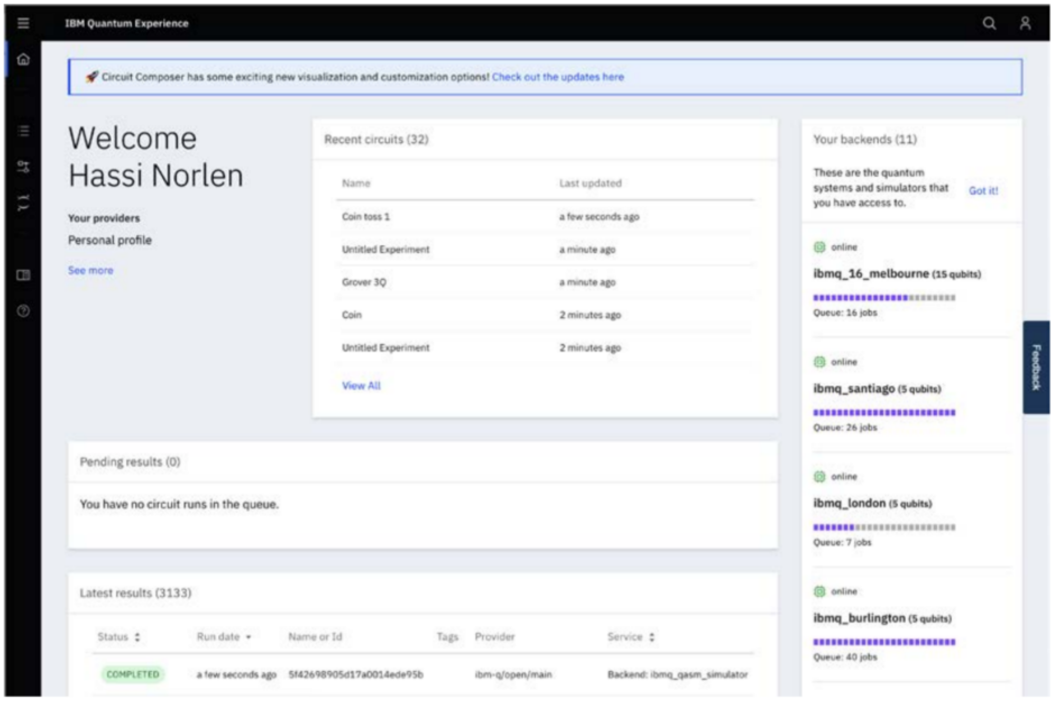
IBM Quantum Experience® opened its doors in May 2016, in a world-first announcement that the public would now have access to actual quantum computing hardware in the cloud. Since then, several other companies have announced similar initiatives and opened up for cloud quantum computing, initially on *simulators*. Notable among this crowd are Google, Microsoft, Rigetti, Qutech, and more. As of this book's writing, IBM gives free access to both hardware and software quantum computing through its IBM Quantum Experience®, we will focus on that platform.

From your web browser, go to the following URL, and log in with your IBM Quantum Experience® account: `https://quantum-computing.ibm.com/`.

You are now on the main IBM Quantum Experience® landing page from which you can access all the quantum experience tools.

From here, you will see the following:

- In the right pane, we have backends that are available to you. Clicking on each brings up a data page with the access status, provider access, chip structure and error rate data, the number of qubits, a list of basis gates, and more.
- In the center area, you find your workbench. There's a list of recent circuits, currently running experiments, and your previous experiment results; when you first drop in here, it will be quite empty. From this area, you can also manage your user profile, configure notifications, get your API key, and more.
- To the left, we have the main tools and help resources. These are described in more detail in the next section's *How to do it…*.



Circuit Composer has some exciting new visualization and customization options! Check out the updates here

Welcome
Hassi Norlen

Your providers
Personal profile
See more

Recent circuits (32)

Name	Last updated
Coin toss 1	a few seconds ago
Untitled Experiment	a minute ago
Grover 3Q	a minute ago
Coin	2 minutes ago
Untitled Experiment	2 minutes ago

View All

Your backends (11)

These are the quantum systems and simulators that you have access to. Got it!

online
ibmq_16_melbourne (15 qubits)
Queue: 16 jobs

online
ibmq_santiago (5 qubits)
Queue: 26 jobs

online
ibmq_london (5 qubits)
Queue: 7 jobs

online
ibmq_burlington (5 qubits)
Queue: 40 jobs

Pending results (0)

You have no circuit runs in the queue.

Latest results (3133)

Status	Run date	Name or Id	Tags	Provider	Service
COMPLETED	a few seconds ago	5f42698905d17a0014ede95b		ibm-q/open/main	Backend: ibmq_qasm_simulator

Figure 3.1 – The IBM Quantum Experience® home page

Now that we have successfully logged in and looked around, let's take a look at the quantum computing programming tools at our disposal. From the main menu on the right, you can access the following pages:

- **Results**
- **Circuit Composer**
- **Quantum Lab**

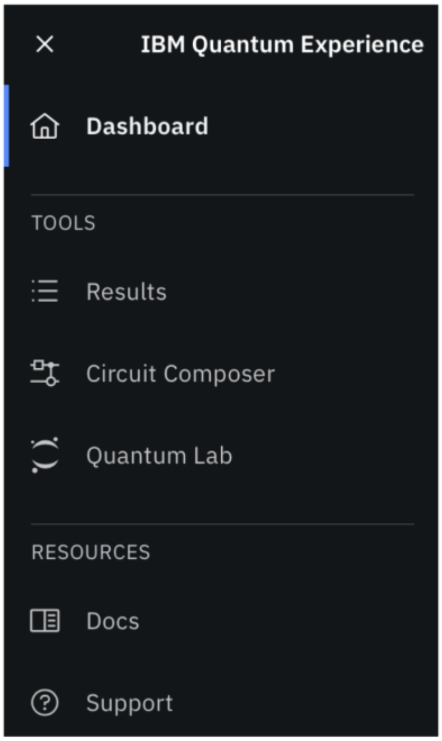
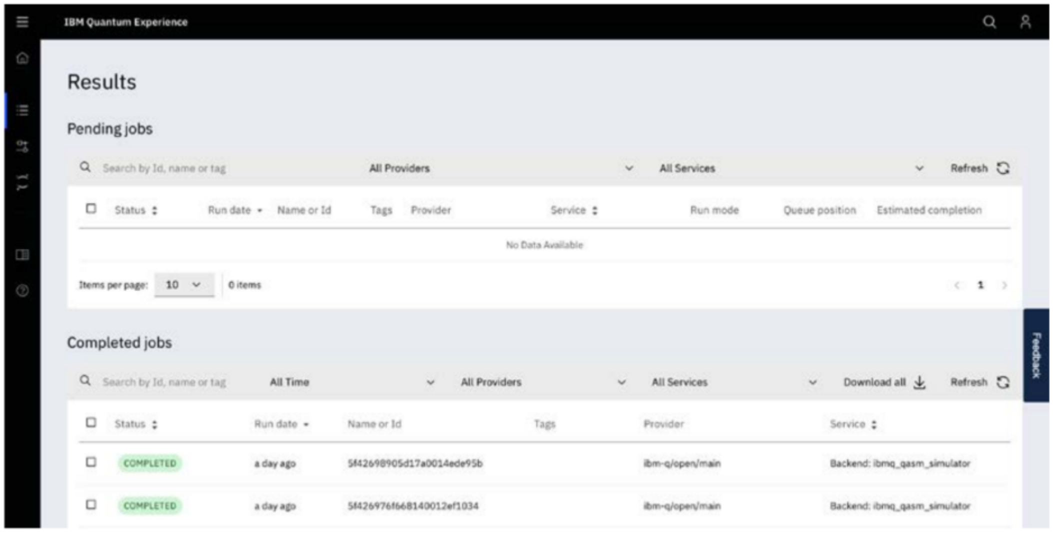


Figure 3.2 – The IBM Quantum Experience® programming tools

Let's look at each of those pages now.

Results

The **Results** section of IBM Quantum Experience® is just a long list of your pending jobs and your previously run quantum computing programs. You can search, sort, and filter by variables such as execution time, services (backends), and more:



Results

Pending jobs

Status	Run date	Name or Id	Tags	Provider	Service	Run mode	Queue position	Estimated completion

No Data Available

Items per page: 10 | 0 items

Completed jobs

Status	Run date	Name or Id	Tags	Provider	Service
COMPLETED	a day ago	5f42698905d17a0014ede95b		ibm-q/open/main	Backend: ibmq_qasm_simulator
COMPLETED	a day ago	5f426976f668140012ef1034		ibm-q/open/main	Backend: ibmq_qasm_simulator

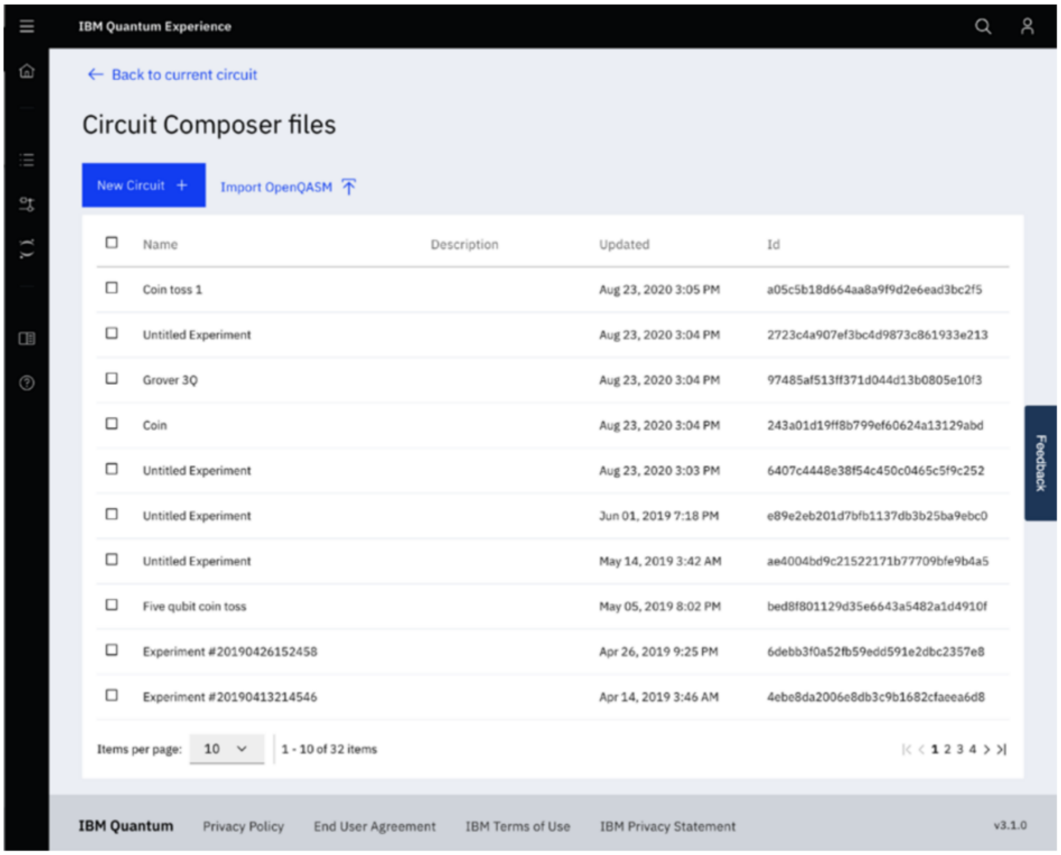
Figure 3.3 – The Results page

The **Results** pane in IBM Quantum Experience® includes not only the jobs that are run from the **Circuit Composer** but also all jobs that you run on IBM Quantum® backends from your local Qiskit® with the same ID.

Each job includes not only the results of your job but also other data such as how long the job stayed in each stage of processing, how long it took to run, the status of the job, the **transpiled** circuit diagram, and the OpenQASM code for the circuit.

Circuit Composer

The Circuit Composer is the main tool for working with your quantum scores (which is what IBM Quantum Experience® calls quantum programs built using the Circuit Composer tool). We will go through it in detail in the recipes in this chapter, but I will provide you with a quick overview of its components here:



Circuit Composer files

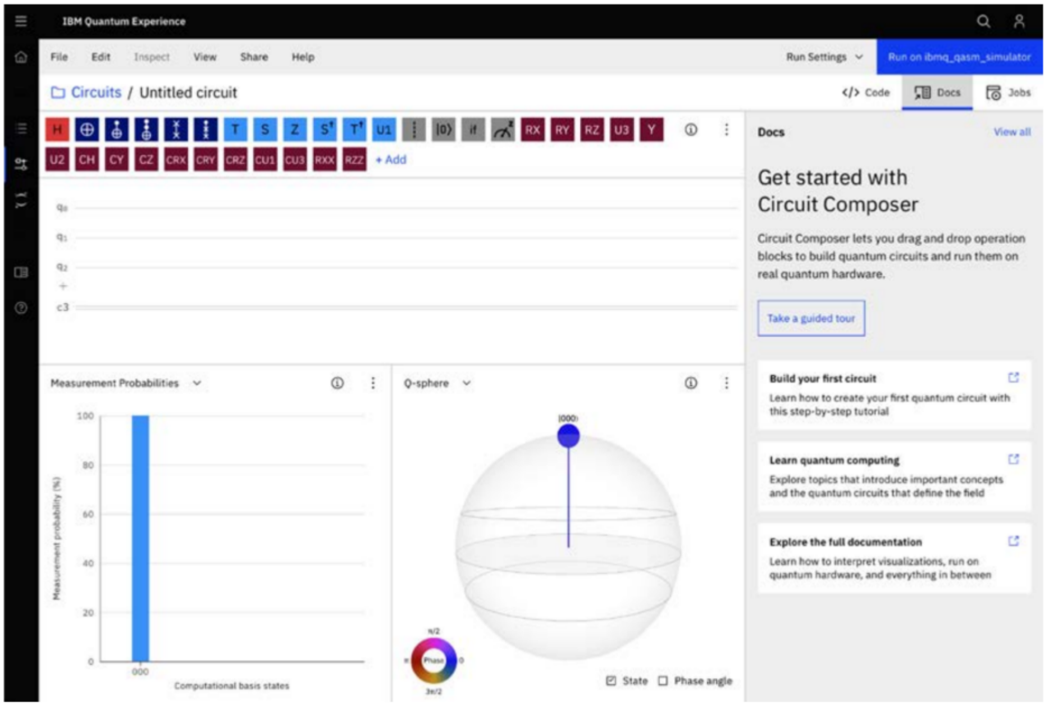
Name	Description	Updated	Id
Coin toss 1		Aug 23, 2020 3:05 PM	a05c5b18d664aa8a9f9d2e6ead3bc2f5
Untitled Experiment		Aug 23, 2020 3:04 PM	2723c4a907ef3bc4d9873c861933e213
Grover 3Q		Aug 23, 2020 3:04 PM	97485af513ff371d044d13b0805e10f3
Coin		Aug 23, 2020 3:04 PM	243a01d19ff8b799ef60624a13129abd
Untitled Experiment		Aug 23, 2020 3:03 PM	6407c4448e38f54c450c0465c5f9c252
Untitled Experiment		Jun 01, 2019 7:18 PM	e89e2eb201d7bfb1137db3b25ba9ebc0
Untitled Experiment		May 14, 2019 3:42 AM	ae4004bd9c21522171b77709bfe9b4a5
Five qubit coin toss		May 05, 2019 8:02 PM	bed8f801129d35e6643a5482a1d4910f
Experiment #20190426152458		Apr 26, 2019 9:25 PM	6debb3f0a52fb59edd591e2dbc2357e8
Experiment #20190413214546		Apr 14, 2019 3:46 AM	4ebe8da2006e8db3c9b1682cfaeea6d8

Items per page: 10 | 1 - 10 of 32 items

Figure 3.4 – The Circuit Composer files page

Just like the **Results** pane has a list of jobs, the **Circuit Composer files** pane has a list of your *circuits*. From here, you can open and run all circuits that you have created using the Circuit Composer.

You can also click **New Circuit** to start from scratch, which opens Circuit Composer on an untitled circuit:


IBM Quantum Experience
File Edit Inspect View Share Help
Run Settings
Run on ibmq_qasm_simulator
Circuits / Untitled circuit
Code Docs Jobs
H T S Z S† T† U1 |0⟩ if RX RY RZ U3 Y
U2 CH CY CZ CRX CRY CRZ CU1 CU3 RXX RZZ + Add
q0 q1 q2 c3
Measurement Probabilities
Measurement probability (%)
100 80 60 40 20 0
000
Computational basis states
Q-sphere
|000⟩
π/2 π 0 3π/2 Phase
State Phase angle
Docs View all
Take a guided tour

Get started with Circuit Composer

Circuit Composer lets you drag and drop operation blocks to build quantum circuits and run them on real quantum hardware.

Build your first circuit
Learn how to create your first quantum circuit with this step-by-step tutorial

Learn quantum computing
Explore topics that introduce important concepts and the quantum circuits that define the field

Explore the full documentation
Learn how to interpret visualizations, run on quantum hardware, and everything in between

Figure 3.5 – A blank circuit in Circuit Composer

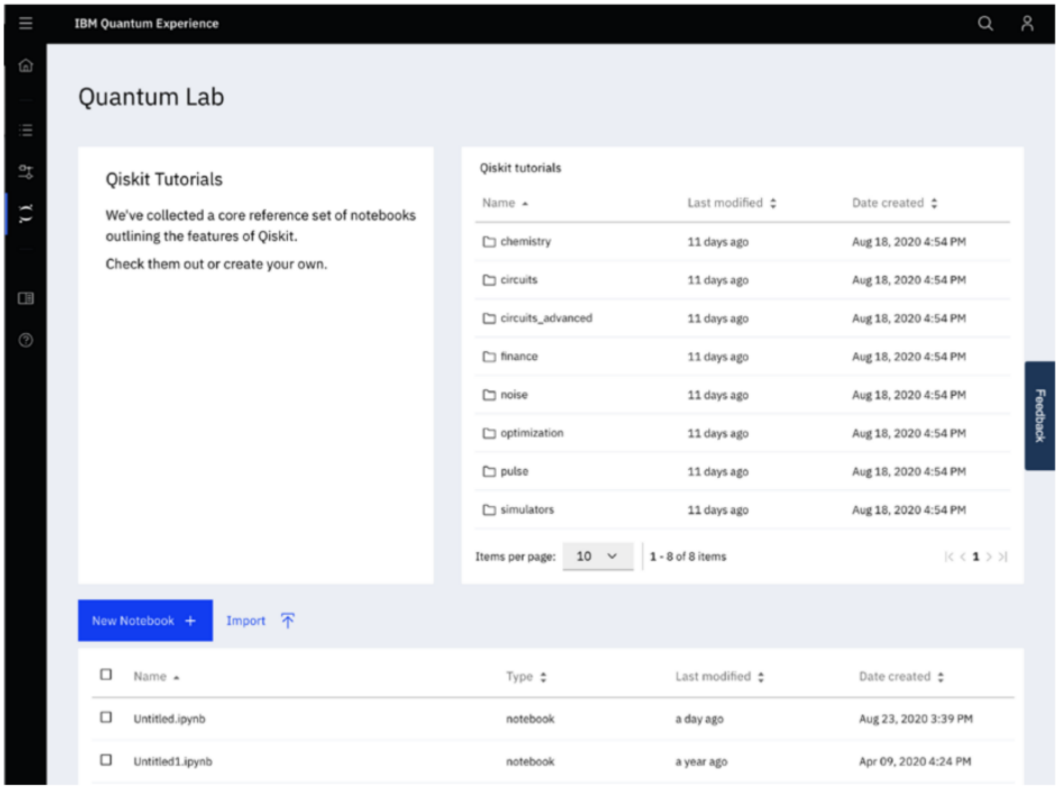
No Qiskit® overlap

The Circuit Composer window does not (in contrast to the **Results** pane) contain any of the circuits that you have run from a local Qiskit® environment. Only the quantum scores that you have created in IBM Quantum Experience® are available here. If you want to see your Qiskit® circuits here, you must import them as OpenQASM code. See the *Moving between worlds* recipe at the end of this chapter.

Once you have opened or created a circuit, a set of new tools open up to help you build your quantum score. These are covered in the next recipe, *Building quantum scores with Circuit Composer*.

Quantum Lab

The third toolkit is a collection of Jupyter Notebook tutorials put together by the Qiskit® team. You can access them all from the **Qiskit tutorials** tile. You can also create your Jupyter Notebooks from this pane, and these will show up in this window much like the circuits in the previous one:



Qiskit Tutorials

We've collected a core reference set of notebooks outlining the features of Qiskit.

Check them out or create your own.

Qiskit tutorials

Name	Last modified	Date created
chemistry	11 days ago	Aug 18, 2020 4:54 PM
circuits	11 days ago	Aug 18, 2020 4:54 PM
circuits_advanced	11 days ago	Aug 18, 2020 4:54 PM
finance	11 days ago	Aug 18, 2020 4:54 PM
noise	11 days ago	Aug 18, 2020 4:54 PM
optimization	11 days ago	Aug 18, 2020 4:54 PM
pulse	11 days ago	Aug 18, 2020 4:54 PM
simulators	11 days ago	Aug 18, 2020 4:54 PM

Items per page: 10 | 1 - 8 of 8 items | 1

Name	Type	Last modified	Date created
Untitled.ipynb	notebook	a day ago	Aug 23, 2020 3:39 PM
Untitled1.ipynb	notebook	a year ago	Apr 09, 2020 4:24 PM

Figure 3.6 – Quantum Lab

Running Python programs in notebooks

You can also use the Jupyter Notebook environment to run the quantum computing Python sample scripts that we include in this book. Take a look at the *Downloading the code samples* recipe in *Chapter 1*, *Preparing Your Environment*, for a quick reminder.

In addition to the tools that you will use to code your quantum programs, IBM Quantum Experience® also includes some extended help in the form of two additional pages:

- **Docs**: This page includes a collection of getting started tutorials, a more extensive set of instructions for Circuit Composer, algorithms, and more. This is a good starting point to explore IBM Quantum Experience® when you are done working your way through this book.
- **Support**: As IBM Quantum Experience® is built on Qiskit®, the support resources are tailored directly for this type of experience via a Slack workspace and Stack Exchange tags for IBM Quantum Experience® (`ibm-q-experience`) and Qiskit® (`qiskit`). These social environments are vibrant and responsive, and you can bounce around questions, ideas, and more in a give-and-take manner. Questions are not long left unanswered by knowledgeable members and moderators!

Building quantum scores with Circuit Composer

This recipe will walk you through the basic steps of creating a quantum score in IBM Quantum Experience®, to get a feel for how the composer works, how to build and modify a score, and finally how to analyze the score step by step using the **Inspect** feature.

Drag-and-drop programming

The recipes in this chapter will be run in the IBM Quantum Experience® web environment, using the drag-and-drop interface, which nicely visualizes what you are doing in an intuitive way.

How to do it...

Let's build ourselves a little quantum score:

1. From your web browser (Chrome seems to work best), go to the following URL, and then log in with your IBM Quantum Experience® account: `https://quantum-computing.ibm.com/`.
2. In the left pane, select **Circuit Composer**.

   This opens the composer to a blank **Untitled circuit**.

3. Optional: Set the number of qubits to play with.

   In the default setting, you will see three lines, much like a music score (hence the term quantum score). Each line represents one of your qubits, and the basic score is designed for a 5-qubit machine. As you will see in the *Comparing backends* recipe in *Chapter 5, Touring the IBM Quantum® Hardware with Qiskit®,* this is currently the most common setup for the free IBM quantum machines.

   For this example, we want to use only 1 qubit for clarity. If we use all five, the results that will be displayed will also include the results of the four we won't be using, which can be confusing.

   So, in the **Untitled circuit** tab that you just opened, hover over a qubit label ($\mathbf{q_0}$). The label shifts to a trash can icon. Use this icon to remove qubits until you have one left. Your quantum score now has only one line.

   Prepending that line is the label, $\mathbf{q_0}$, which is the name of your qubit:

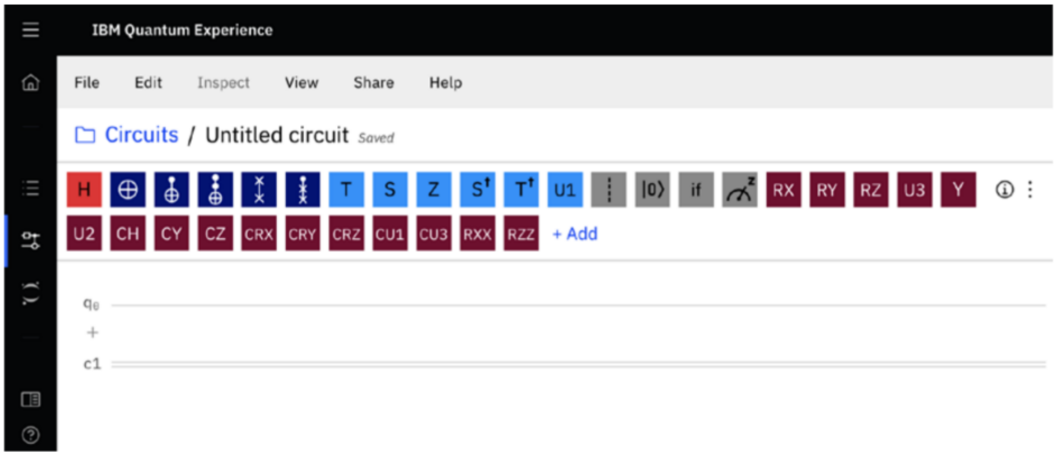


Figure 3.7 – A blank 1-qubit quantum score

4. Add a ⊕ gate to the score.
5. Now select and drag the ⊕ gate to the $\mathbf{q_0}$ line of your score.

Tip

As you will see further in *Chapter 4, Starting at the Ground Level with Terra*, in Qiskit®, the NOT gate is represented by an **X**.

You have now added an X, or NOT gate, which will flip the qubit from its initial set value 0 to 1:

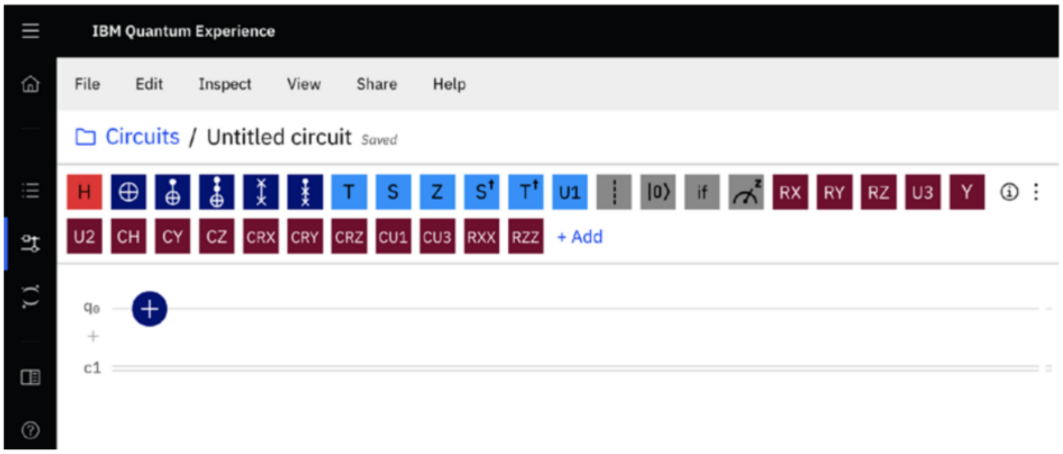


Figure 3.8 – NOT gate added

A glossary of operations

To get more information about the available instructions, click the (**i**) icon in the upper-right corner of Circuit Composer and select **Operations glossary** to open up an exhaustive guide to all instructions (gates, measurements, and more) that are available to you.

6. Now, add a measurement instruction to finish off your circuit.

The measurement instruction is required if you want to run your score and get a result. It measures the state of the $\mathbf{q_0}$ qubit and writes the result (0 or 1) to the classical register (**c1**) so that you can see the outcome of your experiment.

In multi-qubit circuits, there is no need to display all the classical registers as lines. Instead, they are represented by one line labeled with the number of classical registers that it represents; for example, **c5** for five classical registers:

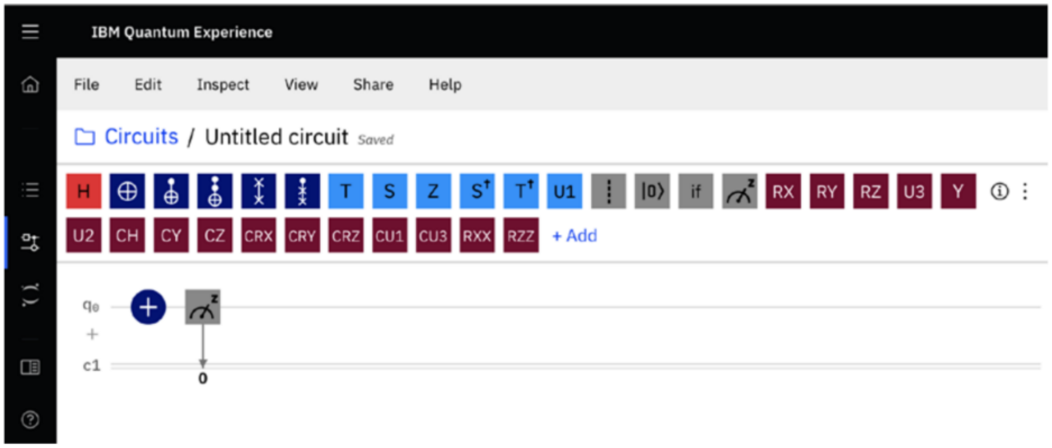


Figure 3.9 – Measurement instruction added

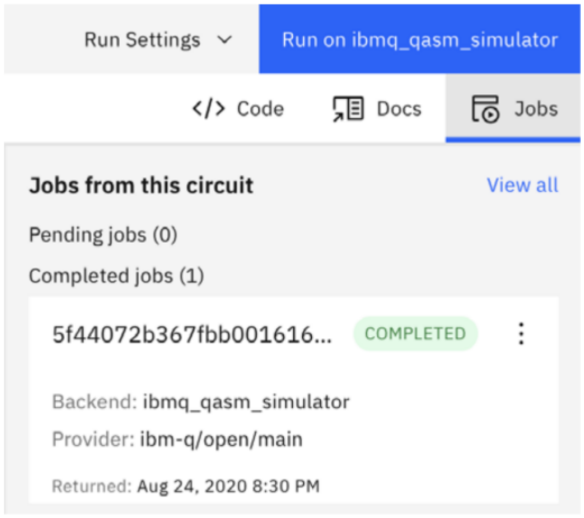
7. You can now run your circuit.

   Optionally, save the circuit by first selecting **Untitled circuit** and giving the experiment a good name.

   Click the **Run on ibmq_qasm_simulator** button.

8. Take a look at the results.

   To see the results of your job, click the **Jobs** icon right beneath the **Run** button. The results of your job are displayed:



Jobs from this circuit View all

Pending jobs (0)

Completed jobs (1)

5f44072b367fbb001616... COMPLETED

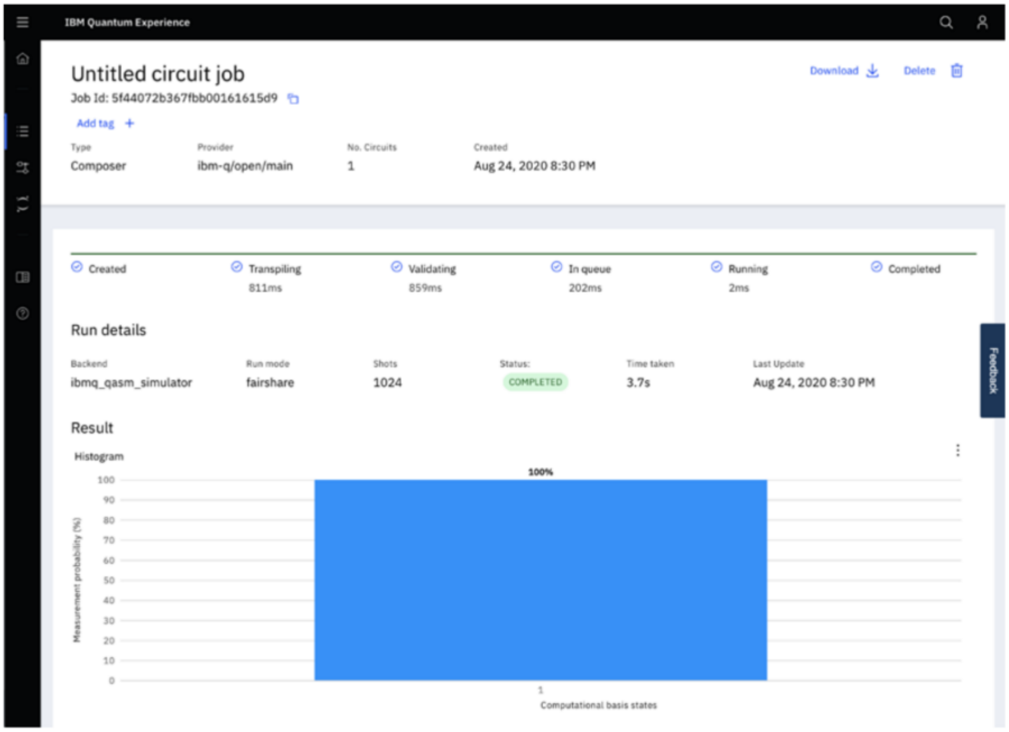
Backend: ibmq_qasm_simulator

Provider: ibm-q/open/main

Returned: Aug 24, 2020 8:30 PM

Figure 3.10 – The job results box

Clicking the job results box opens the **Result** page, and displays the final result of the job you just ran. In this case, we got a result of **1**, with **100%** certainty:



Untitled circuit job

Job Id: 5f44072b367fbb00161615d9

Type	Provider	No. Circuits	Created
Composer	ibm-q/open/main	1	Aug 24, 2020 8:30 PM

Created · Transpiling 811ms · Validating 859ms · In queue 202ms · Running 2ms · Completed

Run details

Backend	Run mode	Shots	Status:	Time taken	Last Update
ibmq_qasm_simulator	fairshare	1024	COMPLETED	3.7s	Aug 24, 2020 8:30 PM

Result

Histogram

Figure 3.11 – Job result: 1 with 100% certainty

9. Now, go ahead and play a little.

   Drag some more quantum instructions into your score willy-nilly, and adjust the number of qubits up and down. You are now building complex quantum circuits, but not necessarily working quantum programs or algorithms. This would be like soldering on random gates to your classical computer or cooking by randomly adding ingredients to your pot. You would get some kind of result, but probably nothing useful or edible. But it is fun!

   Here's an example – see if you can recreate it and then inspect it to see what it does (if anything):

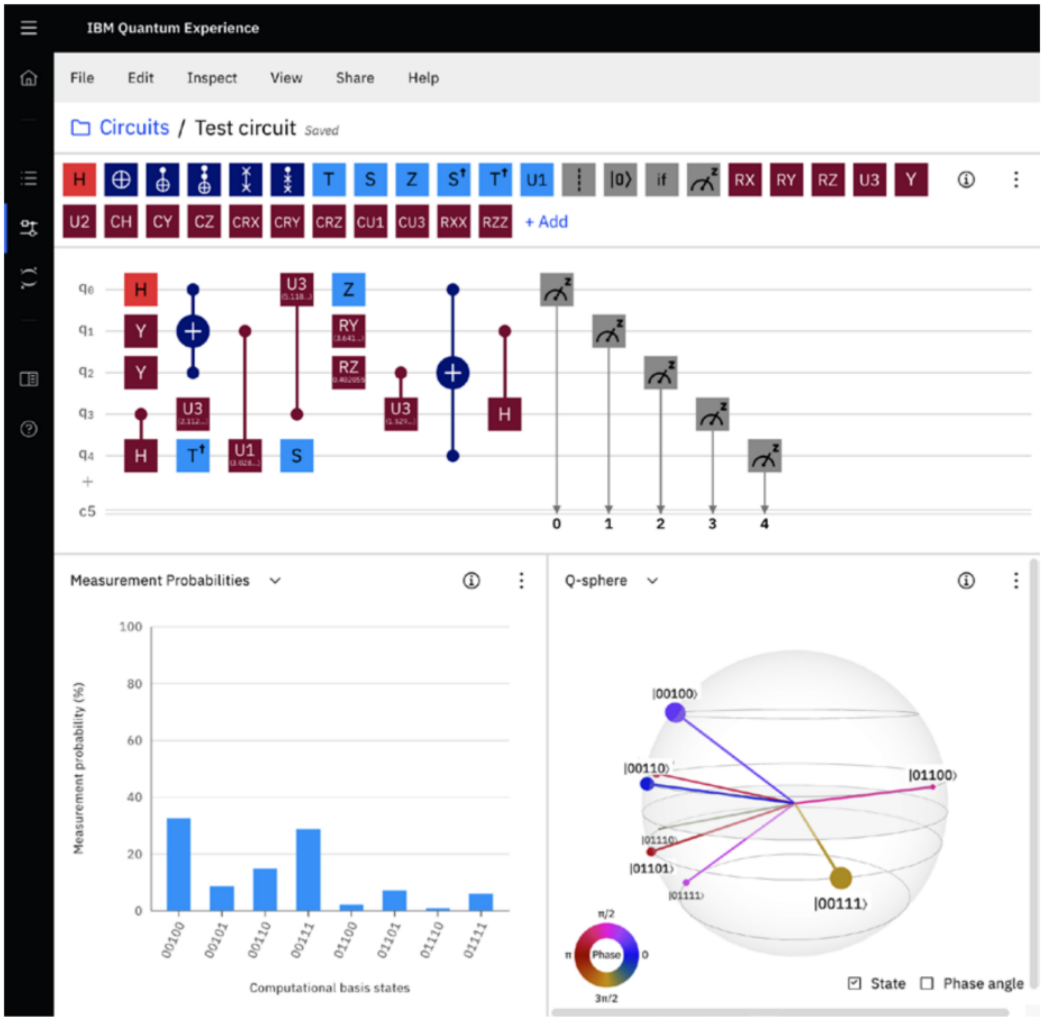


Figure 3.12 – Randomly dragged and dropped circuit