

A hands-on approach

LEARN
Quantum
Computing
with
Python and Q#

Sarah Kaiser
Chris Granade



MEAP



MANNING

WOW! eBook
www.wowebook.org



MEAP Edition
Manning Early Access Program
Learn Quantum Computing with Python and Q#
A Hands-on approach
Version 5

Copyright 2020 Manning Publications

For more information on this and other Manning titles go to
manning.com

©Manning Publications Co. To comment go to [liveBook](#)

WOW! eBook
www.wowebook.org

welcome

Thank you for joining the MEAP for *Learn Quantum Computing with Python and Q#: A Hands-on approach* we hope you'll enjoy getting started with quantum computing!

Quantum computing has recently been gaining more interest as the research field is maturing and large tech companies are investing heavily in it. With all this hype, it is both interesting and important to learn what we could do with these new devices as well as what will likely be out of scope. Our goal with this book is to both introduce you to this exciting new field, as well as Microsoft's domain-specific programming language for quantum computers: Q#.

When we were learning quantum computing ourselves, it was very exciting but also a bit scary and intimidating. Looking back, it doesn't have to be that way, as a lot of what makes topics like quantum computing confusing is the way in which they are presented, not the content itself. With this book, there is an opportunity both for us as authors and for you as readers to learn without this obstruction, using modern programming languages like Python and Q# to help along the way.

We've written the book to be accessible to developers, rather than the "textbook" style common to most other quantum computing books. If you have done some programming before and are familiar with matrices, vectors, and some basic operations involving matrices, such as what you might use in computer graphics or machine learning, you should be good to go! We are importantly *not* assuming any prior knowledge of quantum mechanics or physics, we will help you learn what you need along the way.

By the end of the book, you should be able to:

- Understand what a quantum computer is and why it is such a rapidly expanding field
- Write quantum programs in Q#, Microsoft's new quantum programming language
- Predict what kinds of problems might be suitable to solve with a quantum computer

- Program and use quantum simulators that can be used to run quantum algorithms on classical computers today

We are excited for you to start your own journey through quantum computing by joining this MEAP, and we look forward to hearing from you in the [liveBook Discussion Forum](#) about what you find along your way!

—Sarah Kaiser and Chris Granade

brief contents

PART 1: GETTING STARTED WITH QUANTUM

- 1 Introducing Quantum Computing*
- 2 Qubits: The Building Blocks*
- 3 Sharing Secrets With Quantum Key Distribution*
- 4 Nonlocal Games: Working With Multiple Qubits*
- 5 Teleportation and Entanglement: Moving Quantum Data Around*

PART 2: PROGRAMMING QUANTUM ALGORITHMS IN Q#

- 6 Changing the odds: An introduction to Q#*
- 7 What is a Quantum Algorithm?*
- 8 Quantum sensing: It's not just a phase*

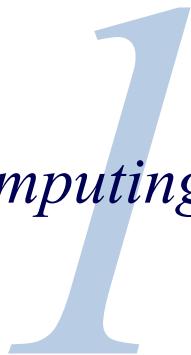
PART 3: APPLIED QUANTUM COMPUTING

- 9 Solving Chemistry Problems with Quantum Computers*
- 10 Searching Databases With Quantum Computers*
- 11 Arithmetic With Quantum Computers*

APPENDICES

- A Installing Required Software*

Introducing Quantum Computing



This chapter covers:

- Why people are excited about quantum computing,
- What a quantum computer is,
- What a quantum computer is and is not capable of, and
- How a quantum computer relates to classical programming.

Quantum computing has been an increasingly popular research field and source of hype over the last few years. There seem to be news articles daily discussing new breakthroughs and developments in quantum computing research, promising that we can solve any number of different problems faster and with lower energy costs. Quantum computing can make an impact across society, making it an exciting time to get involved and learn how to program quantum computers and apply quantum resources to solve problems that matter.

In all of the buzz about the advantages quantum computing offers, however, it is easy to lose sight of the real scope of the advantages. We have some interesting historical precedent for what can happen when promises about a technology outpace reality. In the 1970s, machine learning and artificial intelligence suffered from dramatically reduced funding, as the hype and excitement around AI outstripped its results; this would later be called the "AI winter." Similarly, Internet companies faced the same danger trying to overcome the dot-com bust.

One way forward is by critically understanding what the promise offered by quantum computing is, how quantum computers work, and what they can do, and what is not in

scope for quantum computing. Also it's just really cool to learn about an entirely new computing model! To develop that understanding, as you read this book you'll learn how quantum computers work by programming simulations that you can run on your laptop today. These simulations will show many of the essential elements of what we expect real commercial quantum programming to be like, while useful commercial hardware is coming online.

1.1 Who This Book is For

This book is intended for people who are interested in quantum computing and have had little to no experience with quantum mechanics, but some programming background. As we learn to write quantum simulators in Python and quantum programs in Q#, Microsoft's specialized language for quantum computing, we'll use traditional programming ideas and techniques to help us out. A general understanding of programming concepts like loops, functions, and variable assignments will be helpful. Similarly, we will be using some mathematical concepts from linear algebra such as vectors and matrices, to help us describe the quantum concepts — if you're familiar with computer graphics or machine learning, many of the concepts we need here will be similar. We'll use Python to review the most important mathematical concepts along the way, but familiarity with linear algebra will be helpful.

1.2 Who This Book is Not For

Quantum computing is a wondrous and fascinating new field that offers new ways of thinking about computation, and new tools for solving difficult problems — this book can help you get your start in quantum computing, so that you can continue to explore and learn. That said, this book isn't a textbook, and isn't intended to prepare you for quantum computing research all on its own. As with classical algorithms, developing new quantum algorithms is a mathematical art as much as anything else; while we touch on the math in this book and use it to explain algorithms, there's a variety of different textbooks available that can help you build on the ideas that we cover here.

If you're already familiar with quantum computing, and want to go further into the physics or mathematics, we suggest one of the following resources:

1.2.1 Textbooks and other resources for learning further

- *Quantum Computing: A Gentle Introduction* by Eleanor G. Rieffel and Wolfgang H. Polak (ISBN-13: 9780262526678)
- *Quantum Computing since Democritus* by Scott Aaronson (ISBN-13: 9780521199568)
- *Quantum Computation and Quantum Information* by Michael A. Nielsen and Isaac L. Chuang (ISBN-13: 9781107002173)
- *Quantum Processes Systems, and Information* by Benjamin Schumacher and

Michael Westmoreland (ISBN-13: 978S0521875349)

1.3 How this book is organized

The goal of this text is to enable you to start exploring and using the practical tools we have now for quantum computing. The text of this book is broken up into three main parts that build on each other.

- Part I will gently introduce the concepts needed to describe *qubits*, the fundamental unit of a quantum computer. This Part will describe how to simulate qubits in Python, making it easy to write simple quantum programs.
- Part II will describe how to use the Quantum Development Kit and the Q# programming language to compose qubits, and to run quantum algorithms that perform differently than any known classical algorithms.
- In Part III, we will apply the tools and methods from the previous two Parts to learn how quantum computers might be applied to real-world problems such as simulating chemical properties.

DEEP DIVE: It's OK to snorkel!

Quantum computing is a richly interdisciplinary area of study, bringing together ideas from programming, physics, mathematics, engineering, and computer science. From time to time throughout the book, we'll take a moment to point to how quantum computing draws on ideas from these other fields to put the concepts we're learning about into that richer context.

While these asides are meant to spark curiosity and further exploration, they are by nature tangential. You'll get everything you need to enjoy quantum programming in Python and Q# from this book whether or not you plunge into these deep dives. Taking a deep dive can be fun and enlightening, but if deep dives aren't your thing, that's OK too; it's perfectly fine to snorkel.

1.4 Why does quantum computing matter?

Computing technology advances at a truly stunning pace. Three decades ago, the 80486 processor would allow users to execute 50 MIPS (million instructions per second), but today, small computers like the Raspberry Pi can reach 5,000 MIPS, while desktop processors can easily reach 50,000 to 300,000 MIPS. If you have an exceptionally difficult computational problem you'd like to solve, a very reasonable strategy is to simply wait for the next generation of processors to make your life easier, your videos stream faster, and your games more colorful.

For many problems that we care about, however, we're not so lucky. We might hope that getting a CPU that's twice as fast lets us solve problems twice as big, but just as with so much in life, more is different. Suppose we want to sort a list of 10 million numbers and find that it takes about 1 second. If we later want to sort a list of 1 billion numbers in one second, we'll need a CPU that's 130 times faster, not just 100 times. Some problems make this even worse: for some problems in graphics, going from 10

million to 1 billion points would take 13,000 times longer.

Problems as widely varied as routing traffic in a city and predicting chemical reactions get more difficult *much* more quickly still. If quantum computing were simply a computer that runs 1,000 times as fast, we would barely make a dent in the daunting computational challenges that we want to solve. Thankfully, quantum computers are much more interesting than that. In fact, we expect that quantum computers will likely be much *slower* than classical computers, but that the resources required to solve many problems *scales* differently, such that if we look at the right kinds of problems we can break through "more is different." At the same time, quantum computers aren't a magic bullet, in that some problems will remain hard. For example, while it is likely that quantum computers can help us immensely with predicting chemical reactions, it is possible that they won't be of much help with other difficult problems.

Investigating exactly which problems we can obtain such an advantage in and developing quantum algorithms to do so has been a large focus of quantum computing research. Understanding where we can find advantages by using quantum computers has recently become a significant focus for quantum software development in industry as well. Software platforms such as the Quantum Development Kit make it easier to develop software for solving problems on quantum computers, and in turn to assess how easy different problems are to solve using quantum resources.

Up until this point, it has been very hard to assess quantum approaches in this way, as doing so required extensive mathematical skill to write out quantum algorithms and to understand all of the subtleties of quantum mechanics. Now, industry has started developing software packages and even new languages and frameworks to help connect developers to quantum computing. By leveraging the entire Quantum Development Kit, we can abstract away most of the mathematical complexities of quantum computing and help people get to actually *understanding* and *using* quantum computers. The tools and techniques taught in this book allow developers to explore and understand what writing programs for this new hardware platform will be like.

Put differently, quantum computing is not going away, so understanding what scale of what problems we can solve with them matters quite a lot indeed! There are already many governments and CEOs that are convinced that quantum computing will be the next big thing in computing. Some people care about quantum attacks on cryptography, some want their own quantum computer next year. Independent of the whether or not a quantum "revolution" happens, quantum computing has and will factor heavily into decisions about how to develop computing resources over the next several decades.

1.4.1 Decisions that are strongly impacted by quantum computing

- What assumptions are reasonable in information security?
- What skills are useful in degree programs?

- How to evaluate the market for computing solutions?

For those of us working in tech or related fields, we increasingly must make or provide input into these decisions. We have a responsibility to understand what quantum computing is, and perhaps more importantly still, what it is not. That way, we will be best prepared to step up and contribute to these new efforts and decisions.

All that aside, another reason that quantum computing is such a fascinating topic is that it is both similar to and very different from classical computing. Understanding both the similarities and differences between classical and quantum computing helps us to understand what is fundamental about computing in general. Both classical and quantum computation arise from different descriptions of physical laws such that understanding computation can help us understand the universe itself in a new way.

What's absolutely critical, though, is that there is no one right or even best reason to be interested in quantum computing. Whether you're reading this because you want to have a nice nine-to-five job programing quantum computers or because you want to develop the skills you need to contribute to quantum computing research, you'll learn something interesting to you along the way.

Further reading

If you are interested in exploring the more philosophical or foundational implications of quantum computing, much of this is covered in computational complexity.

Some good resources along these lines are:

- The Complexity Zoo (complexityzoo.uwaterloo.ca/Complexity_Zoo),
- *Complexity Theory: A modern approach* (ISBN-13: 9780521424264), or
- *Quantum Computing Since Democritus* (ISBN-13: 9780521199568).

1.5 What Can Quantum Computers Do?

As quantum programmers we would like to know:

If we have a particular problem, how do we know it makes sense to solve it with a quantum computer?

We are still learning about the exact extent of what quantum computers are capable of, and thus we dont have any concrete rules to answer this question yet. So far, we have found some examples of problems where quantum computers offer significant advantages over the best known classical approaches. In each case, the quantum algorithms that have been found to solve these problems exploit quantum effects to achieve the advantages, sometimes referred to as a quantum advantage.

1.5.1 Some useful quantum algorithms

- Grover's algorithm (Chapter 10): searches a list of N items in \sqrt{N} steps.
- Shor's algorithm (Chapter 11): quickly factors large integers, such as those used by cryptography to protect private data.

Though we'll see several more in this book, both Grover's and Shor's are good examples of how quantum algorithms work: each uses quantum effects to separate correct answers to computational problems from invalid solutions. One way to realize a quantum advantage is to find ways of using quantum effects to separate correct and incorrect solutions to classical problems.

What are quantum advantages?

Grover's and Shor's algorithms illustrate two distinct kinds of quantum advantage. Factoring integers might be easier classically than we suspect, as we haven't been able to prove that factoring is difficult. The evidence that we use to conjecture that factoring is hard classically is largely derived from experience, in that many people have tried very hard to factor integers quickly, but haven't succeeded. On the other hand, Grover's algorithm is provably faster than any classical algorithm, but uses a fundamentally different kind of input.

Other quantum advantages might derive from problems in which we must simulate quantum dynamics. Quantum effects such as interference are quite useful in simulating other quantum effects.

Finding a *provable* advantage for a practical problem is an active area of research in quantum computing. That said, quantum computers can be a powerful resource for solving problems even if we can't necessarily prove that there will never be a better classical algorithm. After all, Shor's algorithm already challenges the assumptions underlying large swaths of classical information security—a mathematical proof is in that sense made necessary only by the fact that we haven't yet built a quantum computer in practice.

Quantum computers also offer significant benefits to simulating properties of quantum systems, opening up applications to quantum chemistry and materials science. For instance, quantum computers could make it much easier to learn about the ground state energies of chemical systems. These ground state energies then provide insight into reaction rates, electronic configurations, thermodynamic properties, and other properties of immense interest in chemistry.

Along the way to developing these applications, we have also seen significant advantages in spin-off technologies such as quantum key distribution and quantum metrology, as we will see in the next few chapters. In learning to control and understand quantum devices for the purpose of computing, we also have learned valuable techniques for imaging, parameter estimation, security, and more. While these are not applications for quantum computing in a strict sense, they go a long way to showing the values of *thinking* in terms of quantum computation.

Of course, new applications of quantum computers are much easier to discover when we have a concrete understanding of how quantum algorithms work and how to build

new algorithms from basic principles. From that perspective, quantum programming is a great resource to learn how to open up entirely new applications.

1.6 What is a Quantum Computer?

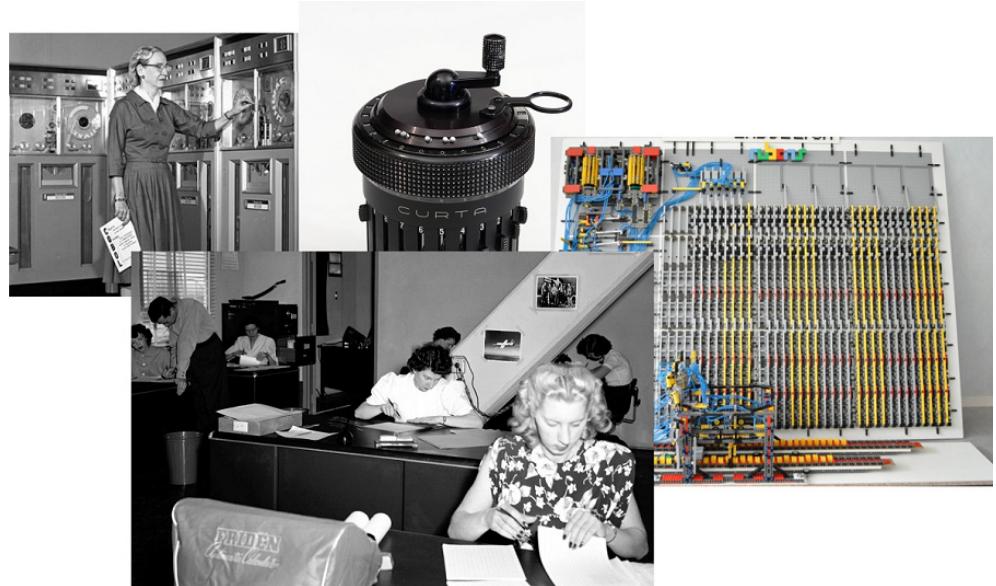
Let's talk a bit about *what* actually makes up a quantum computer. To facilitate this, let's briefly talk about what the term "computer" means. In a very broad sense, a computer is a device that takes data as input and does some sort of operations on that data.

Definition 1.1: Computer

A computer is a device that takes data as input and does some sort of operations on that data.

There are many examples of what we have called a computer, see [1.1](#) for some examples.

Figure 1.1. Several examples of different kinds of computers, including a mainframe operated by Rear Admiral Hopper, a room of humans working to solve flight calculations, a mechanical calculator, and a LEGO-based Turing machine. Each computer can be described by the same mathematical model as computers like cell phones, laptops, and servers.



When we say the word "computer" in conversation, though, we tend to mean something more specific. Often, we think of a *computer* as an electronic device like the one we are currently writing this book on (or that you might be using to read this book!). For any resource up to this point that we have made a computer out of, we can

model it with classical physics—that is, in terms of Newton's laws of motion, Newtonian gravity, and electromagnetism.

Following this perspective, we will refer to computers that are described using classical physics as *classical computers*. This will help us tell apart the kinds of computers we're used to (e.g.: laptops, phones, bread machines, houses, cars, and pacemakers) from the computers that we're learning about in this book.

Specifically, in this book, we'll be learning about quantum computers. By the way we have formulated the definition for classical computers, if we just replace the term *classical physics* with *quantum physics* we have a suitable definition for what a quantum computer is!

Definition 1.2: Quantum Computer

A quantum computer is a device that takes data as input and does some sort of operations on that data, which requires the use of quantum physics to describe this process.

The distinction between classical and quantum computers is precisely that between classical and quantum physics. We will get into this more later in the book, but the primary difference is one of scale: our everyday experience is largely with objects that are large enough and hot enough that even though quantum effects still exist, they don't do much on average. Quantum physics still remains true, even at the scale of everyday objects like coffee mugs, bags of flour, and baseball bats, but we can do a very good job of describing how these objects interact using physical laws like Newton's laws of motion.

DEEP DIVE: What happened to relativity?

Quantum physics applies to objects that are very small and very cold or well-isolated. Similarly, another branch of physics called *relativity* describes objects that are large enough for gravity to play an important role, or that are moving very fast—near the speed of light. We have already discussed classical physics, which could also be said to describe things that are **neither** quantum mechanical nor relativistic. So far we have primarily been comparing classical and quantum physics, raising the question: why aren't we concerned about relativity? Many computers use relativistic effects to perform computation; indeed, global positioning satellites depend critically on relativity.

That said, all of the computation that is implemented using relativistic effects can also be described using purely classical models of computing such as Turing machines. By contrast, quantum computation cannot be described as faster classical computation, but requires a different mathematical model. There has not yet been a proposal for a "gravitic computer" that uses relativity to realize a different model of computation, so we're safe to set relativity aside in this book.

Quantum computing is the art of using small and well-isolated devices to usefully transform our data in ways that cannot be described in terms of classical physics alone. We will see in the next chapter, for instance, that we can generate random numbers on

a quantum device by using the idea of *rotating* between different states. One way to build quantum devices is to use small classical computers such as digital signal processors (DSPs) to control properties of exotic materials.

NOTE
Physics and quantum computing

The exotic materials used to build quantum computers have names that can sound intimidating, like "superconductors" and "topological insulators." We can take solace, though, from how we learn to understand and use classical computers. Modern processors are built using materials like *semiconductors*, but we can program classical computers without knowing what a semiconductor is. Similarly, the physics behind how we can use superconductors and topological insulators to build quantum computers is both a fascinating subject, but it's not required for us to learn how to program and use quantum devices.

Quantum operations are applied by sending in small amounts of microwave power and amplifying very small signals coming out of the quantum device.

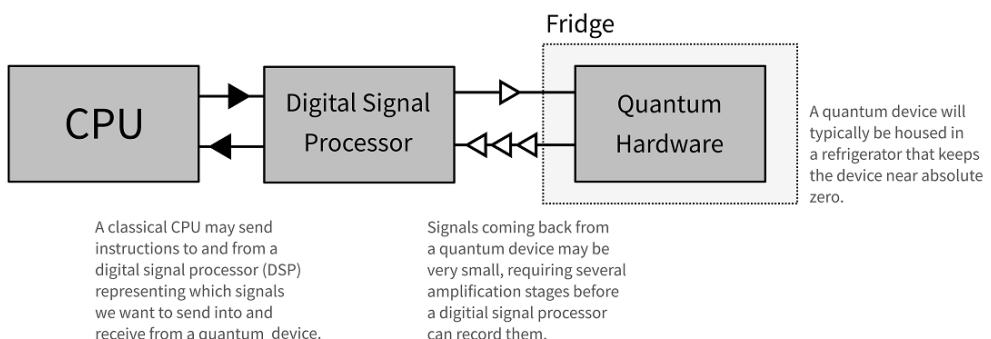
Other qubit devices may differ in the details of how they are controlled, but what remains consistent is that all quantum devices are controlled from and read out by classical computers and control electronics of some kind. After all, we are ultimately interested in classical data, and so there must eventually be an interface with the classical world.

NOTE

For most quantum devices, we need to keep them very cold and very well isolated, since quantum devices can be very susceptible to noise.

By applying quantum operations using embedded classical hardware, we can manipulate and transform quantum data. The power of quantum computing then comes from carefully choosing which operations to apply in order to implement a useful transformation that solves a problem of interest.

Figure 1.2. An example of how a quantum device might interact with a classical computer through the use of a digital signal processor (DSP). The DSP sends low-power signals into the quantum device, and amplifies very low-power signals coming back to the device.



1.6.1 How will we use quantum computers?

It is important to understand both the potential and the limitations of quantum computers, especially given the hype surrounding quantum computation. Many of the misunderstandings underlying this hype stem from extrapolating analogies beyond where they make any sense — all analogies have their limits, and quantum computing is no different in that regard.

TIP

If you've ever seen descriptions of new results in quantum computing that read like "we can teleport cats that are in two places at once using the power of infinitely many parallel universes all working together to cure cancer," then you've seen the danger of extrapolating too far from where analogies are useful.

Indeed, any analogy or metaphor used to explain quantum concepts will be wrong if you dig deep enough. Simulating how a quantum program acts in practice can be a great way to help test and refine the understanding provided by analogies. Nonetheless, we will still leverage analogies in this book, as they can be quite helpful in providing intuition for how quantum computation works.

One especially common point of confusion regarding quantum computing is the way in which users will leverage quantum computers. We as a society now have a particular understanding of what a device called a *computer* does. A computer is something that you can use to run web applications, write documents, and run simulations to name a few common uses. In fact, classical computers are present in every aspect of our lives, making it easy to take computers for granted. We don't always even notice what is and isn't a computer. Cory Doctorow made this observation by noting that "your car is a computer you sit inside of" (www.youtube.com/watch?v=iaf3Sl2r3jE).

Quantum computers, however, are likely to be much more special-purpose. Just as not all computation runs on graphical processing units (GPUs) or field-programmable gate arrays (FPGAs), we expect quantum computers to be somewhat pointless for some tasks.

IMPORTANT

Programming a quantum computer comes along with some restrictions, so classical computers will be preferable in cases where there's no particular quantum advantage to be found.

Classical computing will still be around and will be the main way we communicate and interact with each other as well as our quantum hardware. Even to get the classical computing resource to interface with the quantum devices we will also need in most cases a digital to analogue signal processor as shown in [1.2](#).

Moreover, quantum physics describes things at very small scales (both size and energy) that are well-isolated from their surroundings. This puts some hard limitations to what environments we can run a quantum computer in. One possible solution is to keep our quantum devices in cryogenic fridges, often near absolute 0 K (-459.67 °F, or

-273.15 °C). While this is not a problem at all to achieve in a data center, maintaining a dilution refrigerator isn't really something that makes sense on a desktop, much less in a laptop or a cell phone. For this reason, it's very likely that quantum computers will, at least for quite a while after they first become commercially available, be used through the cloud.

Using quantum computers as a cloud service resembles other advances in specialized computing hardware. By centralizing exotic computing resources in data centers, it's possible to explore computing models that are difficult for all but the largest users to deploy on-premises. Just as high-speed and high-availability Internet connections have made cloud computing accessible for large numbers of users, you will be able to use quantum computers from the comfort of your favorite WiFi-blanketed beach, coffee shop, or even from a train as you watch majestic mountain ranges off in the distance.

Exotic cloud computing resources

- Specialized gaming hardware (PlayStation Now, Xbox One).
- Extremely low-latency high-performance computing (e.g. Infiniband) clusters for scientific problems.
- Massive GPU clusters.
- Reprogrammable hardware (e.g. Catapult/Brainwave).
- Tensor processing unit (TPU) clusters.
- High-permanence high-latency archival storage (e.g. Amazon Glacier).

1.6.2 What can't quantum computers do?

Like other forms of specialized computing hardware, quantum computers won't be good at everything. For some problems, classical computers will simply be better suited to the task. In developing applications for quantum devices, it's helpful to note what tasks or problems are out of scope for quantum computing.

The short version is that we don't have any hard-and-fast rules to quickly decide between which tasks are best run on classical computers, and which tasks can take advantage of quantum computers. For example, the storage and bandwidth requirements for Big Data-style applications are very difficult to map onto quantum devices, where you may only have a relatively small quantum system. Current quantum computers can only record inputs of no more than a few dozen bits, a limitation that will become more relevant as quantum devices are used for more demanding tasks. Although we expect to eventually be able to build much larger quantum systems than we can now, classical computers will likely always be preferable for problems which require large amounts of input/output to solve.

Similarly, machine learning applications that depend heavily on random access to large sets of classical inputs are conceptually difficult to solve with quantum computing. That said, there *may* be other machine learning applications exist that map much more

naturally onto quantum computation. Research efforts to find the best ways to apply quantum resources to solve machine learning tasks are still ongoing. In general, problems that have small input and output data sizes, but large amounts of computation to get from input to output are good candidates for quantum computers.

In light of these challenges, it might be tempting to conclude that quantum computers *always* excel at tasks which have small inputs and outputs, but have very intense computation between the two. Notions like "quantum parallelism" are popular in media, and quantum computers are sometimes even described as using parallel universes to compute.

NOTE

The concept of "parallel universes" is a great example of an analogy that can help make quantum concepts understandable, but that can lead to nonsense when taken to its extreme. It can be sometimes helpful to think of the different parts of a quantum computation as being in different universes that can't affect each other, but this description makes it harder to think about some of the effects we will learn in this book, such as interference. When taken too far, the "parallel universes analogy" also lends itself to thinking of quantum computing in ways that are closer to a particularly pulpy and fun episode of a sci-fi show like Star Trek than to reality.

What this fails to communicate, however, is that it isn't always obvious how to use quantum effects to extract useful answers from a quantum device, even if it appears to contain the desired output. For instance, one way to factor an integer classically is to list each *potential* factor, and to check if it's actually a factor or not.

Factoring N classically

- Let $i = 2$.
- Check if the remainder of N/i is zero.
 - If so, return that i factors N .
 - If not, increment i and loop.

We can speed this classical algorithm up by using a large number of different classical computers, one for each potential factor that we want to try. That is, this problem can be easily parallelized. A quantum computer can try each potential factor within the same device, but as it turns out, this isn't *yet* enough to factor integers faster than the classical approach above. If you run this on a quantum computer, the output will be one of the potential factors chosen at random. The actual correct factors will occur with probability about $1/\sqrt{N}$, which is no better than the classical algorithm above.

As we'll see in Chapter 11, though, we can improve this by using other quantum effects, however, to factor integers with a quantum computer faster than the best-known classical factoring algorithms. Much of the heavy lifting done by Shor's algorithm is to make sure that the probability of measuring a correct factor at the end is much larger than measuring an incorrect factor. Cancelling out incorrect answers in this way is where much of the art of quantum programming comes in; it's not easy or

even possible to do for all problems we might want to solve.

To understand more concretely what quantum computers can and can't do, and how to do cool things with quantum computers despite these challenges, it's helpful to take a more concrete approach. Thus, let's consider what a quantum program even **is**, so that we can start writing our own.

1.7 **What is a Program?**

Throughout this book, we will often find it useful to explain a quantum concept by first re-examining the analogous classical concept. In particular, let's take a step back and examine what a classical program is.

Definition 1.3: Program

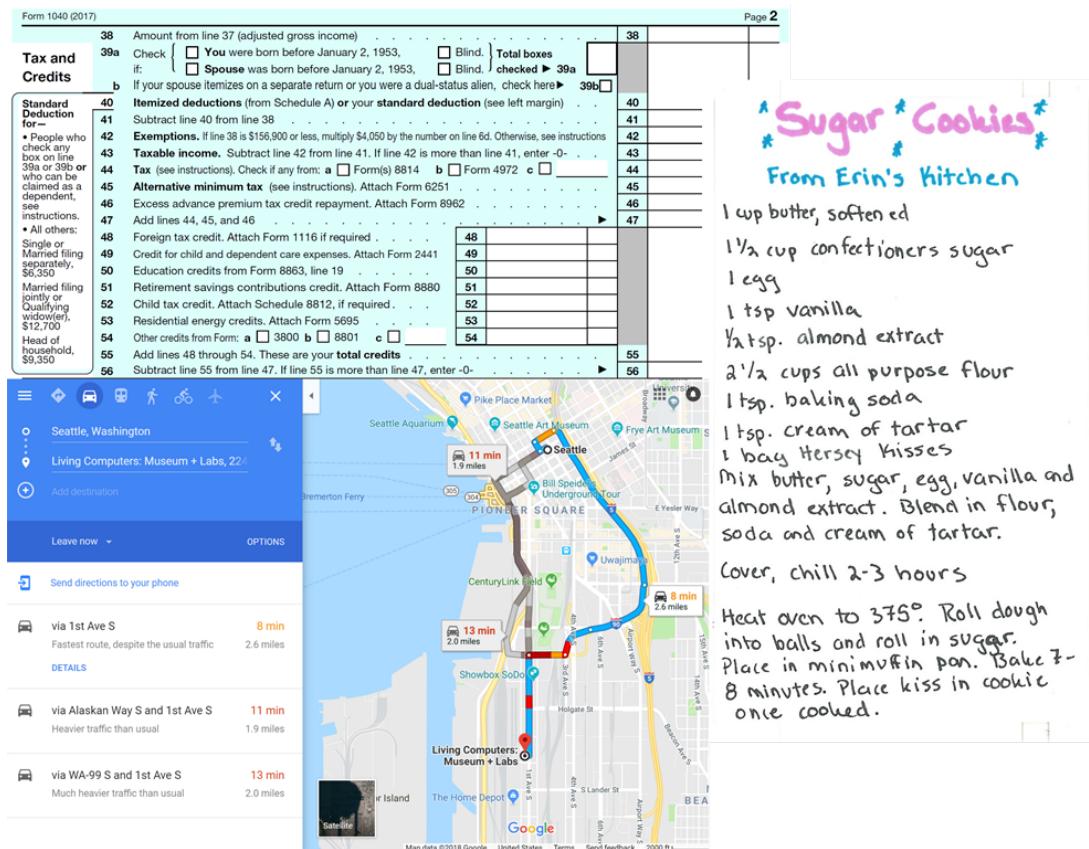
A program is a sequence of instructions that can be interpreted by a classical computer to perform a desired task.

Examples of classical programs

- Tax forms
- Map directions
- Recipes
- Python scripts

We can write classical programs to break down a wide variety of different tasks for interpretation by all sorts of different computers.

Figure 1.3. Examples of classical programs. Tax forms, map directions, and recipes are all examples in which a sequence of instructions is interpreted by a classical computer such as a person.



Let's take a look at what a simple "hello, world" program might look like in Python:

```
>>> def hello():
...     print("Hello, world!")
...
>>> hello()
Hello, world!
```

At its most basic, this program can be thought of as a sequence of instructions given to the Python *interpreter*, which then executes each instruction in turn to accomplish some effect — in this case, printing a message to the screen.

We can make this way of thinking more formal by using the `dis` module provided with Python to *disassemble* `hello()` into a sequence of instructions:

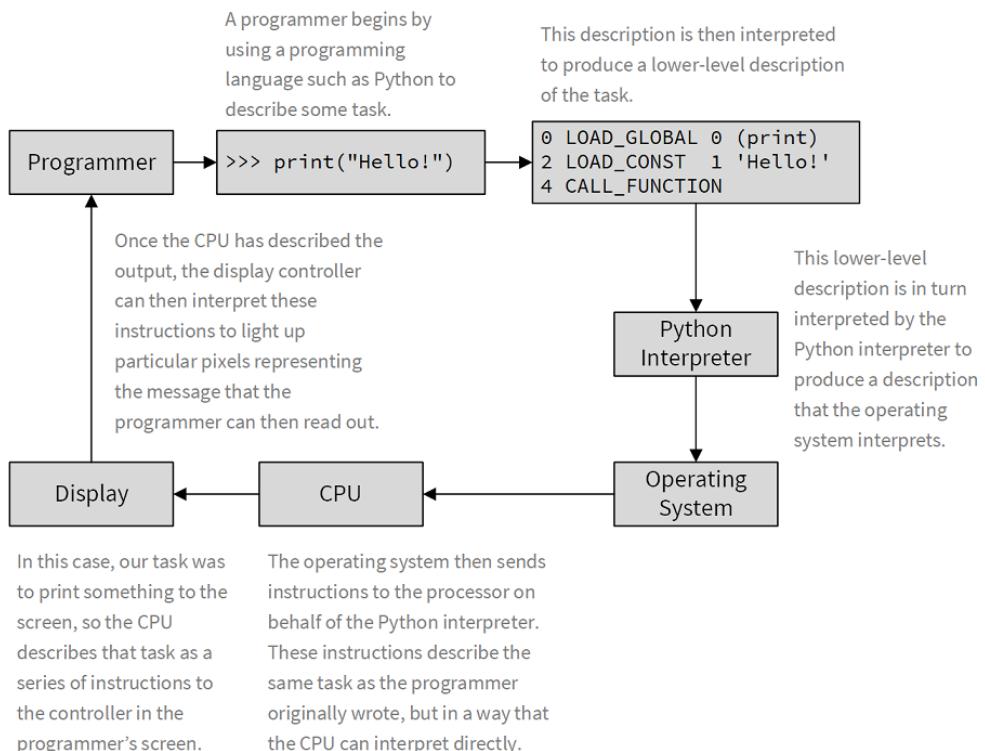
```
>>> import dis
>>> dis.dis(hello)
 2           0 LOAD_GLOBAL               0 (print)
 2 LOAD_CONST    1 ('Hello, world!')
 4 CALL_FUNCTION          1
 6 POP_TOP
 8 LOAD_CONST    0 (None)
10 RETURN_VALUE
```

NOTE

You may get different output on your system, depending on what version of Python you're using.

Each line consists of a single instruction that is passed to the Python virtual machine; for instance, the LOAD_GLOBAL instruction is used to look up the definition of the print function. The print function is then called by the CALL_FUNCTION instruction. The Python code that we wrote above was *compiled* by the interpreter to produce this sequence of instructions. In turn, the Python virtual machine executes our program by calling instructions provided by the operating system and the CPU.

Figure 1.4. An example of how a classical computing task is repeatedly described and interpreted.



At each level, we have a *description* of some task that is then *interpreted* by some other program or piece of hardware to accomplish some goal. This constant interplay between description and interpretation motivates calling Python, C, and other such programming tools *languages*, emphasizing that programming is ultimately an act of communication.

In the example of using Python to print "Hello, world!" we are effectively communicating with Guido von Rossum, the founding designer of the Python language. Guido then effectively communicates on our behalf with the designers of the operating system that we are using. These designers in turn communicate on our behalf with Intel, AMD, ARM, or whomever has designed the CPU that we are using, and so forth. As with any other use of language to communicate, our choice of programming language affects how we think and reason about programming. When we choose a programming language, the different features of that language and the syntax used to express those features mean that some ideas are more easily expressed than others.

1.7.1 What is a Quantum Program?

Like classical programs, quantum programs consist of sequences of instructions that are interpreted by classical computers to perform a particular task. The difference, however, is that in a quantum program, the task we wish to accomplish involves controlling a quantum system to perform a computation.

Figure 1.5. Writing a quantum program with the Quantum Development Kit and Visual Studio Code.

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Debug, Terminal, Help.
- Title Bar:** TeleportationSample.qs - Teleportation - Visual Studio Code.
- Left Sidebar (EXPLORER):**
 - OPEN EDITORS
 - TELEPORTATION
 - Program.cs
 - README.md
 - TeleportationSample.csproj
 - TeleportationSample.qs
- Code Editor:**

```

operation Teleport (msg : Qubit, there : Qubit) : Unit {
    using (register = Qubit[1]) {
        // Ask for an auxiliary qubit that we can use to prepare
        // for teleportation.
        let here = register[0];

        // Create some entanglement that we can use to send our message.
        H(here);
        CNOT(here, there);

        // Move our message into the entangled pair.
        CNOT(msg, here);
        H(msg);

        // Measure out the entanglement.
        if (M(msg) = One) {
            Z(there);
        }
    }
}
```
- Terminal:**

```
Chris@ ~\...\Quantum\...\Teleportation > dotnet run
Round 0: Sent True, got True.
Teleportation successful!!

Round 1: Sent False, got False.
Teleportation successful!!

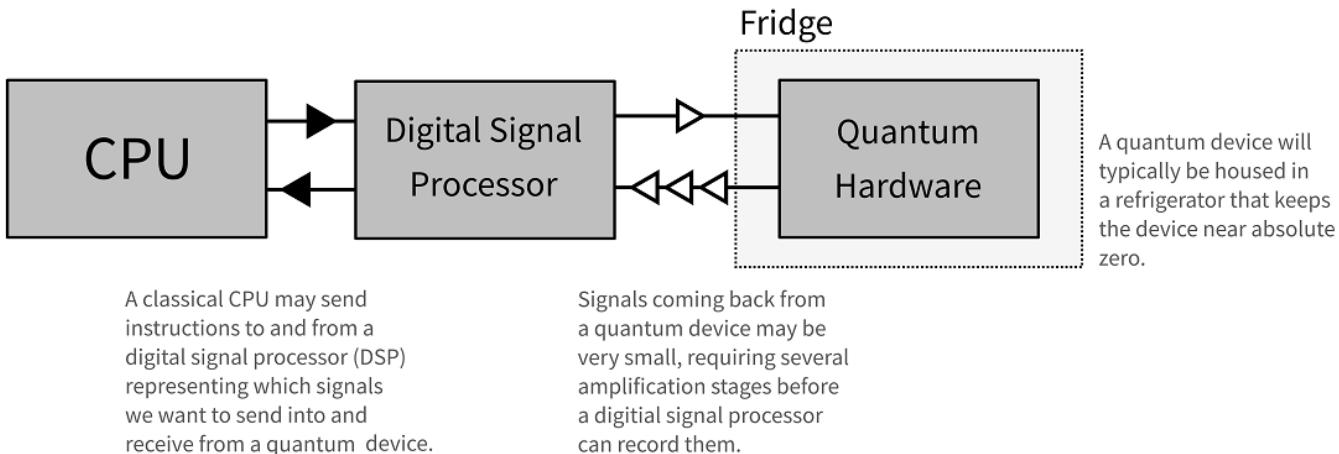
Round 2: Sent True, got True.
Teleportation successful!!

Round 3: Sent False, got False.
Teleportation successful!!

Round 4: Sent True, got True.
```
- Bottom Status Bar:**
 - master
 - 0 0 0
 - Teleportation
 - Ln 58, Col 20 Spaces: 4 UTF-8 CRLF Q# 😊

The instructions available to classical and quantum programs differ according to this difference in tasks. For instance, a classical program may describe a task such as loading some cat pictures from the Internet in terms of instructions to a networking stack, and eventually in terms of assembly instructions such as `mov` (move). By contrast, quantum languages like Q# allow programmers to express quantum tasks in terms of instructions like `M` (measure).

Figure 1.6. An example of how a quantum device might interact with a classical computer through the use of a digital signal processor (DSP). The DSP sends low-power signals into the quantum device, and amplifies very low-power signals coming back to the device.



When run using quantum hardware, these programs may instruct a digital signal processor such as that shown in [1.6](#) to send microwaves, radio waves, or lasers into a quantum device, and to amplify signals coming out of the device.

If we are to achieve a different ends, however, it makes sense for us to use a language that reflects what we wish to communicate! We have many different classical programming languages for just this reason, as it doesn't make sense to use only one of C, Python, JavaScript, Haskell, Bash, T-SQL, or any of a whole multitude of other languages. Each language focuses on a subset of tasks that arise within classical programming, allowing us to choose a language that lets us express how we would like to communicate that task to the next level of interpreters.

Quantum programming is thus distinct from classical programming almost entirely in terms of what tasks are given special precedence and attention. On the other hand, quantum programs are still interpreted by classical hardware such as digital signal processors, so a quantum programmer writes quantum programs using classical computers and development environments.

Throughout the rest of this book, we will see many examples of the kinds of tasks that a quantum program is faced with solving or at least addressing, and what kinds of classical tools we can use to make quantum programming easier.

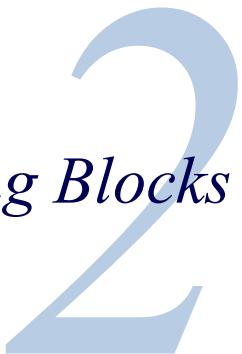
1.8 Summary

In this chapter you learned:

- Recognize the significance of quantum computing in modern society,

- Predict what kinds of problems a quantum computer may be good at solving,
- and recognize the similarities and differences between programming for a quantum computer vs. a classical computer.

Qubits: The Building Blocks



This chapter covers:

- Why random numbers are an important resource.
- What is a qubit?
- What are the basic operations we can perform on a qubit?
- How to program a quantum random number generator in Python.

In this chapter, we are going to start to get our feet wet with some quantum programming concepts. The main concept we will explore is the qubit, the quantum analogue of a classical bit. We use qubits as an abstraction or model to describe the new kinds of computing that are possible with quantum physics. To help learn about what qubits are and how we interact with them, we will use an example of how they are being used today: random number generation. While we can build up much more interesting devices from these qubits, the simple example of a *quantum random number generator* (QRNG) will be a good way to get familiar with the qubit!

2.1 Why do we need random numbers?

Humans like certainty. We like it when we press a key on our keyboard and it does the same thing every time. However, there are some contexts in which we *do* want randomness.

2.1.1 Things some humans like to use randomness for

- Playing games

- Simulating complex systems (e.g.: stock market)
- Picking secure secrets (e.g.: passwords and cryptographic keys)

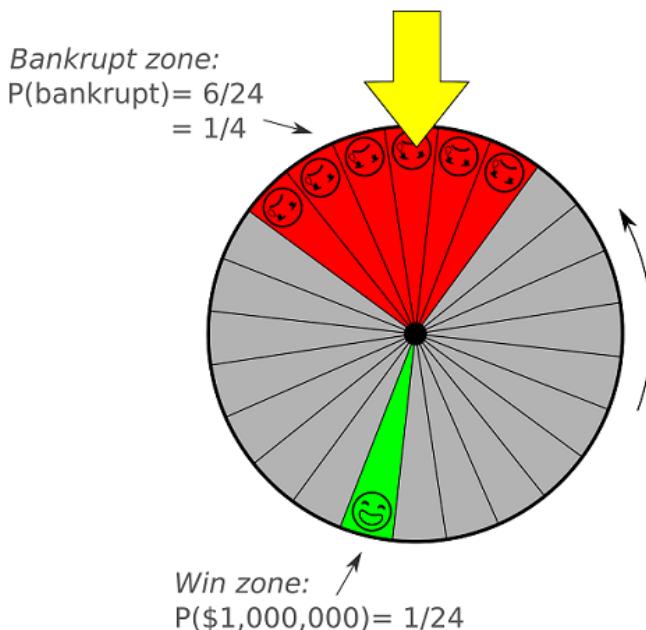
In all of these situations where we want randomness, we can describe the "chances" for each outcome. Being random events means that describing the chances is all we can say about the situation until the die is cast (or the coin is flipped or the password is re-used). When we describe the "chances" of each example, we say things like:

2.1.2 Statements about probability

- if I roll this die, then I will see a six **with probability** 1 out of 6,
- if I flip this coin, then I will see a heads **with probability** 1 out of 2.

We can also describe cases where the probabilities aren't the same for every outcome we could measure. In *Wheel of Fortune™* (2.1), the probability that if we spin the wheel then we will get a \$1,000,000 prize is much smaller than the probability that if we spin the wheel, then we will go bankrupt.

Figure 2.1. Probabilities of \$1,000,000 and Bankrupt in Wheel of Fortune™



Like in game shows, there are also many contexts in computer science where randomness is critical, especially when security is required. If we want to keep some information private, then cryptography lets us do so by combining our data with random numbers in different ways. If our random number generator isn't very good — that is to say if an attacker can predict what numbers we use to protect our private data — then cryptography doesn't help us very much. We can also imagine using a

poor random number generator to run a raffle or a lottery; an attacker who figures out how our random numbers are generated can take us straight to the bank.

NOTE**What are the odds**

You can lose a lot of money by using random numbers that your adversaries can predict. Just ask the producers of *Press Your Luck!*, a popular 1980s game show. A contestant found that he could predict where their new electronic "wheel" would land, letting him win more than \$250,000 in today's money. Read more at: priceconomics.com/the-man-who-got-no-whammies/.

As it turns out, quantum mechanics lets us build some really unique sources of randomness. If we build them right, the randomness of our results is guaranteed by *physics*, not an assumption about how long it would take for a computer to solve a difficult problem. This means that for a hacker or adversary they would have to break the laws of physics to break the security! Now this does not mean that you should use quantum random numbers for everything, humans are still the weakest link in security infrastructure :)

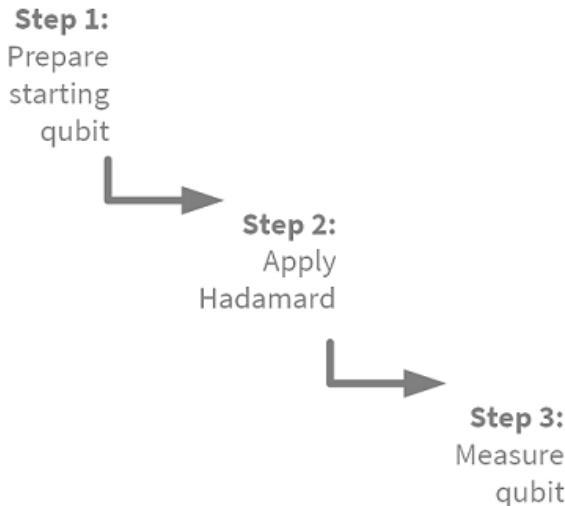
DEEP DIVE: Computational Security and Information Theoretic Security

Some ways of protecting private information rely on assumptions about what problems are easy or hard for an attacker to solve. For instance, the RSA algorithm is a commonly used encryption algorithm, and is based on the difficulty of finding prime factors for large numbers. RSA is used on the web and in other contexts to protect user data, under the assumption that adversaries can't easily factor very large numbers. So far, this has proven to be a rather good assumption, but it is entirely possible that a new factoring algorithm is discovered, undermining the security of RSA. New models of computation like quantum computing also change how reasonable or unreasonable it is to make computational assumptions like "factoring is hard." As we'll see in Chapter 9, a quantum algorithm known as *Shor's algorithm* allows for solving some kinds of cryptographic problems much faster than classical computers, challenging the assumptions that are commonly used to promise computational security.

By contrast, if an adversary can only ever randomly guess at secrets, even with very large amounts of computing power, then a security system provides much better guarantees about its ability to protect private information. Such systems are said to be *informationally secure*. In the next chapter, we will see that generating random numbers in a hard-to-predict fashion allows us to implement an informationally secure procedure called a one-time pad.

This gives us some confidence that we can use them for vital tasks, such as to protect private data, run lotteries, and to play Dungeons and Dragons™. Simulating how quantum random number generators work lets us learn many of the basic concepts underlying quantum mechanics, so let's jump right in and get started!

Figure 2.2. Quantum random number generator algorithm.



One great way to get started is to look at an example of a quantum program that generates random numbers. Let's call it a quantum random number generator or QRNG for short. Don't worry if the algorithm doesn't make a lot of sense right now, we'll explain the different pieces as we go through the rest of the chapter.

2.1.3 Quantum random number generator algorithm

- Ask the quantum device to allocate a qubit.
- Apply a Hadamard instruction to our qubit.
- Measure our qubit and return the result.

In the rest of the chapter, we'll develop a Python class `QuantumDevice` to let us write programs that implement algorithms like the one above. Once we have a `QuantumDevice` class, we'll be able to write out QRNG as a Python program similar to classical programs that you're used to.

NOTE

Please see Appendix A for instructions on how to setup Python on your device to run quantum programs.

Listing 2.1. qrng.py : A quantum program that generates random numbers. Note that this sample will not run until you have written the simulator in this chapter ☺

```
def qrng(device : QuantumDevice) -> bool:
    with device.using_qubit() as q:
        q.h()
    return q.measure()
```

1
2
3
4

- ➊ Our quantum programs are written just like the classical programs that you're used to. In this case, we're using Python, so our quantum program is a Python function `qrng` that implements a quantum random number generator.
- ➋ Quantum programs work by asking quantum computing hardware for *qubits*, quantum analogues of bits that we can use to perform computations.
- ➌ Once we have a qubit, we can issue *instructions* to that qubit. Similarly to assembly languages, these instructions are often denoted by short abbreviations; we'll see what `h()` stands for later on in this chapter.
- ➍ To get data back from our qubits, we can *measure* them. In this case, half of the time, our measurement will return `True`, and the other half of the time, we'll get back `False`.

That's it!

Four steps and we've just created our first quantum program. This QRNG returns true or false. In Python terms, this means that you get a 1 or a 0 each time you run `qrng`. It's not a very sophisticated random number generator but the number it returns is truly random.

To run the `qrng` program above, we'll need to give our function a `QuantumDevice` that can give us access to qubits and that implements the different instructions we can send to qubits. Though we need only one qubit, to start, we're going to build our own quantum computer simulator. *Existing hardware could be used for this modest task, but what we will look at later will be beyond the scope of available hardware.* It will run locally on a laptop or desktop and behave in the same way as actual quantum hardware. The simplest kind of device that we can consider is a simulator that runs locally on a laptop or desktop, and that behaves in the same way as actual quantum hardware. Throughout the rest of the chapter, we'll build up the different pieces we need to write our own simulator and to run `qrng`.

2.2 What are Classical Bits?

When learning about the concepts of quantum mechanics, it can often be helpful to step back and re-examine *classical* concepts in a way that makes it easier to make the connection to how that concept is expressed in quantum computing. With that in mind, let's step back and take another look at what *bits* are.

Suppose that you'd like to send your dear friend Eve an important message, such as ❤️. How can we represent our message in a way that it can be easily sent?

We might start by making a list of every letter and symbol that we could hope to use in writing down messages. Thankfully, the Unicode Consortium (unicode.org/) has already done this for us, and has assigned a *code* to a very wide variety of the

characters we use across the world to communicate with each other. For instance, I is assigned the code 0049, while ☺ is given the code A66E, ☹ is denoted by 2E0E, and ❤ by 1F496. These codes may not seem too helpful at first glance, but they're useful recipes for how to send each symbol as a message. If we know how to send two messages, let's call them 0 and 1, these recipes let us build up more complicated messages like ☺, ☹, and ❤ as sequences of 0 and 1 messages:

0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Now we can send whatever we want if we know how to send just two messages to Eve, a 0 and a 1 message. Using these recipes, our message of "❤" becomes 0001 1111 0100 1001 0110 or unicode 1F496.

TIP

Don't send 0001 1111 0100 1001 0100 by mistake, or Eve will get a ❤ from you!

We call each of the two messages "0" and "1" a **bit**.

NOTE

To distinguish bits from the quantum bits that we'll see throughout the rest of the book, we'll often emphasize that we're talking about *classical* bits.

When we use the word bit, we generally mean one of two things:

- Any physical system that can be completely described by answering one true/false question.
- The information stored in such a physical system.

For example, padlocks, light switches, transistors, left or right spin on a curveball, and wine in wine glasses can all be thought of as bits, as we can use all of them to send or record messages:

Table 2.1. Table Examples of bits

Label	Padlock	Light Switch	Transistor	Wine Glass	Baseball
0	Unlocked	Off	Low voltage	Has white wine	Rotating to the left
1	Locked	On	High voltage	Has red wine	Rotating to the right

These examples are all bits, because we can fully describe them to someone else by

answering a single true/false question. Put differently, each example lets us send either a 0 or a 1 message. Like all conceptual models, a "bit" has its limitations — how would we describe a rosé wine, for instance?

That said, a "bit" is a useful tool because we can describe ways of interacting with bits that are independent of how we actually build the bit.

2.2.1 What Can We Do With Classical Bits?

Now that we have a way of describing and sending classical information, what can we do to process and modify our information? We describe the ways that we can process information in terms of *operations*, which we define as the ways of describing how a model can be changed or acted upon.

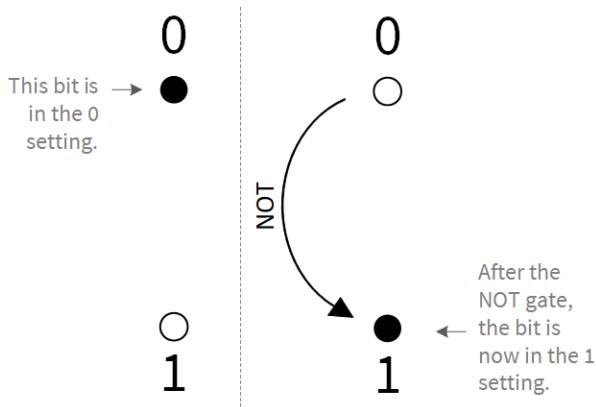
To visualize the NOT operation, let's imagine labeling two points as "0" and "1".

Figure 2.3. We depict a classical bit as a black dot in either the 0 or 1 position.



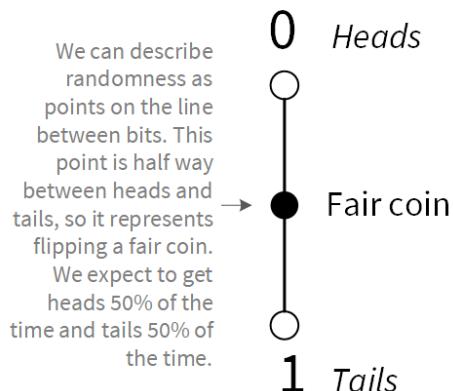
The NOT operation is then any transformation which turns "0" bits into "1" bits and vice versa. In classical storage devices like hard drives (and even floppy disks!), a NOT gate flips the magnetic field that stores our bit value. We can think of NOT as implementing a 180° rotation between the "0" and "1" points we drew above.

Figure 2.4. The classical NOT operation flips a 0 bit to a 1 bit and vice versa.



Visualizing classical bits this way also lets us extend our notion of bits slightly to include a way to describe *random* bits (which will be helpful later). If we have a *fair coin* (that is, a coin that lands heads "half" of the time and "tails" the other half), then it wouldn't be correct to call that coin a "0" or a "1". We only know what bit value our coin bit has if we set it with a particular side face up on a surface, or we can flip it for a random bit value. Every time we flip a coin, we know that eventually it will land and we will get a heads or tails. Whether it lands heads or tails is governed by a probability called the *bias* of the coin. We have to pick a side of the coin to describe the bias, which is easy to phrase as a question like this: What is the probability the coin will land heads? Thus a fair coin would have a bias of 50% because it lands with the value heads half of the time, which is mapped to the bit value 0 in [2.5](#).

Figure 2.5. Extending the concept of a bit to describe a coin, which has a probability of being found in either "0" or "1" each time it is flipped.



Using this visualization, we can take our previous two dots indicating the bit values "0" and "1" and connect them with a line on which we can plot the bias of our coin. It becomes easier to see that a NOT operation (which still works on our new probabilistic bit) doesn't do anything to a fair coin. If "0" and "1" occur with the same probability, then it doesn't matter if we rotate a "0" to a "1" and a "1" to a "0", we'll still wind up with "0" and "1" having the same probability.

What if our bias is not in the middle? If we know that someone is trying to cheat by using a weighted or modified coin that almost always lands on "heads", we could say the bias of the coin is 90% and could plot it on our line bit by drawing a point much closer to "0" than to "1".

Definition 2.1: State

We say that the point on a line where one would draw each classical bit is the *state* of that bit.

Let's consider a scenario:

Say I want to send you a bunch of bits stored using padlocks, what is the cheapest way I could do so?

One approach is to mail a box containing many padlocks that are either open or closed and hope that they arrive in the same state that I sent them in. On the other hand, we can both agree that all padlocks start off initially in the "0" (unlocked) state, and I can send you instructions on which padlocks to close. This way, you can go buy your own padlocks, and I only need to send a **description** of how to prepare those padlocks using classical NOT gates. Sending a piece of paper or even just an email is way cheaper than mailing a box of padlocks!

This illustrates a principle we will rely on throughout the book: **The state of a physical system can also be described in terms of instructions for how prepare that state.** Thus, the operations allowed on a physical system also define what states are possible.

Though it may sound completely trivial, there is one more thing that we can do with classical bits that will turn out to be critical to how we understand quantum computing: we can look at them. If I look at a padlock and conclude that "aha! that padlock is unlocked~!", then I can now think of my brain as a particularly squishy kind of bit. The "0" message is stored in my brain by my thinking "aha! that padlock is unlocked~!", while a "1" message might have been stored by my thinking "ah, well, that padlock is locked ☹." In effect, by looking at a classical bit, I have *copied* it into my brain. We say that the act of *measuring* the classical bit copies that bit.

More generally, modern life is built all around the ease with which we copy classical bits by looking at them. We copy classical bits with truly reckless abandon, measuring

many billions of classical bits every second that we copy data from our video game consoles to our TVs.

On the other hand, if a bit is stored as a coin, then the process of measuring involves flipping it. Now measuring doesn't quite copy the coin, as I might get a different measurement result the next time I flip. If I only have one measurement of a coin, I can't conclude what the probability of getting a heads or tails was. We didn't have this ambiguity with our padlock bits, because we knew the state of our padlocks was either "0" or "1". If I measured a padlock and found it to be in the "0" state, I would know that unless I did something to the padlock, it would always be in the "0" state.

The situation isn't precisely the same in quantum computing, as we'll see later on in the chapter. While measuring classical information is cheap enough that we complain about precisely how many billions of bits a \$5 cable can let us measure, we will have to be much more careful with how we approach quantum measurements.

2.2.2 Abstractions are our friend

Regardless of how we physically build a bit, we can thankfully represent them in the same way in both math and in code. For instance, Python provides the `bool` type (short for Boolean, in honor of the logician George Boole), which has two valid values: `True` and `False`. We can then represent transformations on bits such as NOT and OR as operations acting on `bool` variables. Importantly, we can specify a classical operation by describing how that operation transforms each possible input, which is often called a *truth table*.

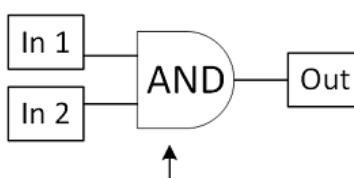
Definition 2.2: Truth table

A table describing the output of a classical operation for every possible combination of inputs is called that operation's *truth table*.

Figure 2.6. Truth table for the logical operation AND.

Truth tables are one way to show what happens to classical bits in functions or logical circuits.

Input	Output	
0 0	0	
0 1	0	
1 0	0	
1 1	1	



This logic gate generates the same output bits as the truth table to the left.

For example, we can find the truth table for the NAND (short for NOT-AND) operation in Python by iterating over combinations of `True` and `False`:

Listing 2.2. Using python to print out a truth table for NAND

```
>>> from itertools import product
>>> for inputs in product([False, True], repeat=2):
...     output = not (inputs[0] and inputs[1])
...     print(f"{inputs[0]}\t{inputs[1]}\t->\t{output}")
False  False    ->    True
False  True     ->    True
True   False    ->    True
True   True     ->    False
```

NOTE

Truth tables all the way down

Describing an operation as a truth table holds for more complicated operations as well; in principle even an operation like addition between two 64-bit integers can be written as a truth table. This isn't very practical, though, as a truth table for two 64-bit inputs would have $2^{128} \approx \times 10^{38}$ entries, and would take 10^{40} bits to write down. By comparison, recent estimates put the size of the entire Internet at closer to 10^{27} bits.

Much of the art of classical logic and hardware design is making *circuits* which can provide very compact representations of classical operations, rather than relying on potentially massive truth tables. In quantum computing we use the name *unitary operators* for similar truth tables for our quantum bits, which we will expand on as we go along.

In summary,

- Classical bits are physical systems which can be in one of two different *states*,
- Classical bits can be manipulated through *operations* to process information,
- The act of *measuring* a classical bit makes a copy of the information contained in the state.

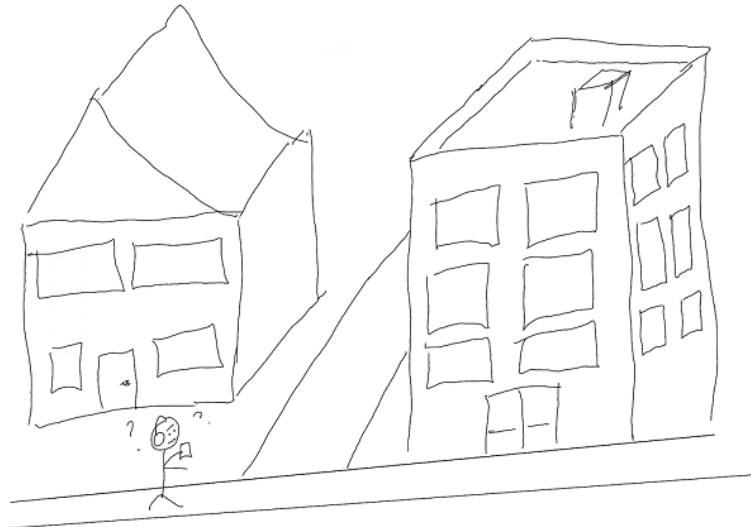
We're almost ready to get back to qubits, but we need a bit of math to help us out.

2.3 Approaching Vectors

One more thing we'll need before we can get to what qubits are is the concept of a *vector*.

Suppose a friend of yours is having people over to celebrate that they fixed their doorbell, and you'd very much like to find their house and celebrate the occasion with them. How can your friend help you find their home?

Figure 2.7. Looking for your friend's house party...



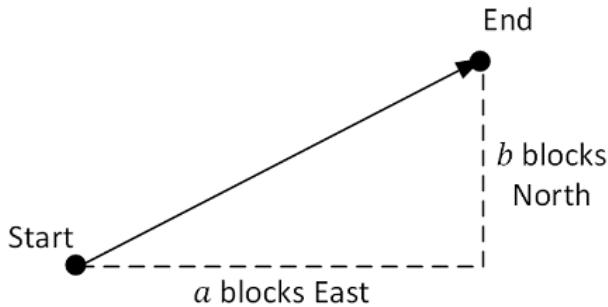
Vectors are a mathematical tool that can be used to represent a variety of different concepts—basically, anything that we can record by making an ordered list of numbers.

2.3.1 Examples of vectors

- Points on a map
- Colors of pixels in a display
- Damage elements in a computer game
- Velocity of an airplane
- Orientation of a gyroscope

For instance, if I'm lost in an unfamiliar city, someone can tell me where to go by giving me a vector that instructs me to take a blocks East and then b blocks North (we'll set aside the problem of routing around buildings). We write these instructions with the vector $[[a], [b]]$.

Figure 2.8. Vectors as coordinates.



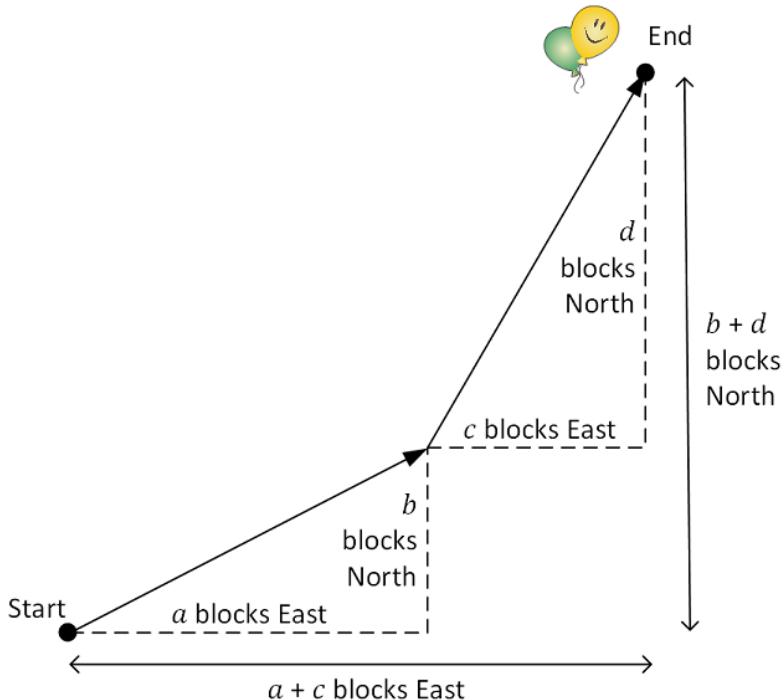
Like ordinary numbers, we can add different vectors together.

NOTE

An ordinary number is often called a *scalar* to distinguish it from a vector.

Using this way of thinking about vectors, we can think of this addition between vectors as being defined element-wise. That is, we interpret $[[a], [b]] + [[c], [d]]$ as being instructions to go a blocks East, b blocks North, c blocks East, then finally d blocks North. Since it doesn't matter what order we step in, this is equivalent to taking $a + c$ blocks East, then $b + d$ blocks North, and so we write that $[[a], [b]] + [[c], [d]]$ is $[[a + c], [b + d]]$.

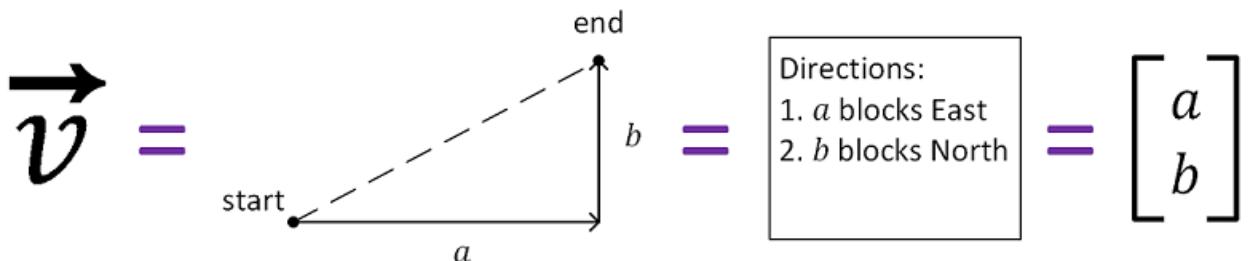
Figure 2.9. Adding vectors to find a party.



Definition 2.3: Vector

A vector \vec{v} in d dimensions can be written as a list of d numbers. For instance, $\vec{v} = [[2], [3]]$ is a vector in two dimensions.

Figure 2.10. Drawn vectors have the same information as a list of directions, or a column of numbers.



Similarly, we can multiply vectors by ordinary numbers to transform vectors. I may be lost not just in any city, for example, but a city that uses meters instead of the feet that I'm used to. To transform a vector given in meters to a vector in feet, I'll need to multiply each element of my vector by about 3.28. Let's do this using a Python library called *NumPy* to help us manage how we represent vectors in a computer.

TIP Full installation instructions are provided in Appendix A.

Listing 2.3. Representing vectors in Python with NumPy.

```
>>> import numpy as np
>>> directions_in_meters = np.array(
...     [[30], [50]])
>>> directions_in_feet = 3.28 * directions_in_meters
>>> directions_in_feet
array([[ 98.4],
       [164. ]])
```

1
2
3
4

- ➊ Vectors are a special case of NumPy *arrays*. We create arrays using the array function, passing a list of the rows in our vector. Each row is then a list of the columns —for vectors, we only ever have one column per row, but we'll have examples later where this isn't true.
- ➋ Let's start with an example of going 30 meters East, then 50 meters North.
- ➌ NumPy represents multiplication between scalars and vectors by the Python multiplication operator *.
- ➍ Printing out the result of multiplying, I see that I need to go 98.4 feet East, then 164 feet North.

This structure makes it easier to communicate directions. If we didn't use vectors, then each scalar would need its own direction, and it would be critical to keep the directions and scalars together.

2.4 Seeing the Matrix for Ourselves

As we'll see shortly, we can describe how qubits transform as we apply instructions to them in the same way that we describe transforming vectors, using a concept from linear algebra called a *matrix*. This is especially important as we consider transformations of vectors that are more complicated than adding or rescaling.

To see how to use matrices, let's return to the problem of finding the party — that doorbell isn't going to ring itself, after all! Up until now, we've simply assumed that the first component of each vector means East and the second means North, but someone could well have chosen another convention. Without a way of reconciling the transformation between these two conventions, I'll never find the party! Thankfully, not only will matrices help us model qubits later on in the chapter, they can help me find my way to my friends!

Happily, the transformation between listing North first and listing East first is simple to implement: we swap the coordinates $[[a], [b]]$ to obtain $[[b], [a]]$. Suppose that this swap is implemented by some function swap. Then, swap plays nicely with the vector addition that we saw above, in that $\text{swap}(v + w)$ is always the same as $\text{swap}(v) + \text{swap}(w)$. Similarly, if we stretch a vector and then swap (that is, scalar multiplication), that's the same as if we had swapped and then stretched: $\text{swap}(a * v) = a * \text{swap}(v)$. Any function that has these two properties is a *linear* function.

Definition 2.4: Linear function

A linear function is a function f such that $f(ax + by) = af(x) + bf(y)$ for all scalars a and b and all vectors x and y .

Linear functions are common in computer graphics and machine learning, as they include a variety of different ways of transforming vectors of numbers.

2.4.1 Examples of linear functions

- Rotations
- Scaling and stretching
- Reflections

What all of these linear functions have in common is that we can break them apart and understand them piece by piece. Thinking again of the map, if I'm trying to find my way to the party still (hopefully there's still some punch left) and the map I've been given has been stretched out by 10% in the North–South direction, and has been flipped in the East–West direction, that's not too hard to figure out. Since both the stretching out and flipping are linear functions, someone can set me on the right path by telling me what happened to the North–South direction and the East–West directions separately. In fact, we just did that at the beginning of this paragraph!

TIP

If you learn just one thing from this book, the most important take-away that we have to offer is that you can understand linear functions and thus quantum operations by breaking them up into components. We will see in the rest of the book that, since operations in quantum computing are described by linear functions, we can understand quantum algorithms by breaking them apart in the same way as we broke up our map example. Don't worry if that doesn't make a lot of sense at the moment, as it's a way of thinking that takes some getting used to.

This is because once I understand what happens to the North vector (let's call it $[[1], [0]]$ as before), and to the West vector (let's call it $[[0], [1]]$), then I can figure out what happens to *all* vectors by using the linearity property. For example, if I am told there's a really pretty sight 3 blocks North and 4 blocks West of me, and I want to figure out where that is on my map, I can do so piece by piece:

- I need to stretch the North vector out by 10% and multiply it by 3, getting $[[3.1], [0]]$.
- I need to flip the West vector and multiply it by 4, getting $[[0], [-4]]$.
- I finish by adding what happens to each direction, getting $[[3.1], [-4]]$.

Linear functions are pretty special! ❤️

In the example above, we were able to stretch our vectors using a linear function. This is because linear functions aren't sensitive to scale. Swapping North–South and East–West does the same thing to vectors, whether they're represented in steps, blocks, miles, furlongs, or parsecs. That's not true of most

functions, though. Consider a function that squares its input, $f(x) = x^2$. The larger x is, the more it gets stretched out.

That linear functions work the same way no matter how large or small their inputs are is precisely what lets us break them down piece by piece: once we know how a linear function works at *any* scale, we know how they work at *all* scales.

Thus, I need to look 3.1 blocks North and 4 blocks East on my map.

TIP

Later, we'll see how the bits "0" and "1" can be thought of as directions or vectors, not too different from North or East. In the same way that North and East aren't the best vectors to help you understand Minneapolis, we'll find that "0" and "1" aren't always the best vectors to help you understand quantum computing.

Figure 2.11. North and West aren't always the best directions to use if you want to understand where you're going. See this map of downtown Minneapolis, where a large section of the downtown grid is rotated to match the bend in the Mississippi river. Photo by davecito.



This way of understanding linear functions by breaking them apart piece by piece works for rotations, too. If my map has the compass rotated by 45° clockwise (wow, I need a serious lesson in cartography), so that North becomes Northeast, and West becomes Northwest, then I can still figure out where things are piece by piece. Using the same example, the North vector now gets mapped to approximately $[[0.707], [0.707]]$ on the map, and the West vector gets mapped to $[[-0.707], [0.707]]$.

When we sum up what happens in the example above, we thus get $3 * [[0.707], [0.707]] + 4 * [[-0.707], [0.707]]$, which is equal to $(3 - 4) * [[0.707], [0]] + (3 + 4) [[0], [0.707]]$, giving us $[[-0.707], [4.95]]$.

It might seem that this has less to do with linearity and more to do with that North and West are somehow special. We could have, however, run through *exactly* the same argument but writing down Southwest as $[[1], [0]]$ and Northwest as $[[0], [1]]$. This works because Southwest and Northwest are perpendicular to each other, allowing us to break down any other direction as a combination of Northwest and Southwest. Other than ease of reading a compass that we buy off a shelf, there's nothing that makes North or West special. If you've ever tried to drive around downtown Minneapolis (see [2.11](#)), it quickly becomes apparent that North and West aren't always the best way to understand directions!

Formally, we call any set of vectors that lets us understand directions by breaking them apart piece by piece in this way a *basis*.

NOTE

Technically, we'll be concerned here with what mathematicians call an *orthonormal basis*, as that's most often useful in quantum computing. All that means is that the vectors in a basis are perpendicular to all the other basis vectors and have a length of 1.

Let's try an example of writing a vector in terms of a basis. The vector $\vec{v} = [[2], [3]]$ can be written as $2 \vec{b}_0 + 3 \vec{b}_1$ using the basis $\vec{b}_0 = [[1], [0]]$ and $\vec{b}_1 = [[0], [1]]$.

Definition 2.5: Basis

If any vector \vec{v} in d dimensions can be written as a sum of multiples of $\vec{b}_0, \vec{b}_1, \dots, \vec{b}_{d-1}$, we say that $\vec{b}_0, \vec{b}_1, \dots, \vec{b}_{d-1}$ are a **basis**. In two dimensions, one common basis is horizontal and vertical.

More generally, if we know the output of a function f for each vector in a basis, we can compute f for any input. This is similar to how we used truth tables to describe a classical operation by listing the outputs of an operation for each possible input.

Problem solving with linearity

Let's say f is a linear function that represents how our map is stretched and twisted, how could we find where we need to go? We want to compute the value $f(np.array([[2], [3]]))$ (a somewhat arbitrary value) given our basis $f(np.array([[1], [0]]))$ (horizontal) and $f(np.array([[0], [1]]))$ (vertical).

$[1]])$) (vertical)? We also know from looking parts of the map legend we see that the map warps the horizontal direction to $\text{np.array}([[1], [1]])$ and the vertical direction to $\text{np.array}([[1], [-1]])$

Steps to compute $f(\text{np.array}([[2], [3]]))$

- We use our basis, $\text{np.array}([[1], [0]])$ and $\text{np.array}([[0], [1]])$, to write that $\text{np.array}([[2], [3]])$ is equal to $2 * \text{np.array}([[1], [0]]) + 3 * \text{np.array}([[0], [1]])$.
- Using this new way to write our input to the function, we want to compute $f(2 * \text{np.array}([[1], [0]]) + 3 * \text{np.array}([[0], [1]]))$.
- Next, we use that f is linear to write $f(2 * \text{np.array}([[1], [0]]) + 3 * \text{np.array}([[0], [1]]))$ as $2 * f(\text{np.array}([[1], [0]])) + 3 * \text{np.array}(f([[0], [1]]))$:

Listing 2.4. Using NumPy to help compute $f(\text{np.array}([[2], [3]]))$

```
>>> import numpy as np
>>> horizontal = np.array([[1], [0]])                                ①
>>> vertical = np.array([[0], [1]])
>>> vec = 2 * horizontal + 3 * vertical                            ②
>>> vec
array([[2],
       [3]])
>>> f_horizontal = np.array([[1], [1]])                               ③
>>> f_vertical = np.array([[1], [-1]])
>>> 2 * f_horizontal + 3 * f_vertical                                ④
array([[ 5],
       [-1]])
```

- ➊ First, we define variables `horizontal` and `vertical` to represent the basis we will use to represent $[[2], [3]]$.
- ➋ We can write $[[2], [3]]$ by adding multiples of `horizontal` and `vertical`.
- ➌ We next define how f acts on `horizontal` and `vertical` by introducing new variables `f_horizontal` and `f_vertical` to represent $f(\text{horizontal})$ and $f(\text{vertical})$, respectively.
- ➍ Because f is linear, we can define how it works for $[[2], [3]]$ by replacing `horizontal` and `vertical` by the outputs `f_horizontal` and `f_vertical`.

Using this insight, we can make a table of how a linear function transforms each of its inputs. These tables are called *matrices*, and are complete descriptions of linear functions. If I tell you the matrix for a linear function, then you can compute that function for *any* vector. For example, the transformation from the North/East convention to the East/North convention for map directions transforms the instruction "go one unit North" from being written as $[[1], [0]]$ to being written as $[[0], [1]]$. Similarly, the instruction "go one unit East" goes from being written as $[[0], [1]]$ to being written as $[[1], [0]]$. If I stack up the outputs for both sets of instructions, I get the following matrix:

Listing 2.5. Stacking the vectors for swapping East/North conventions on a map

```
>>> swap_north_east = np.array([[0, 1], [1, 0]])
>>> swap_north_east
array([[0, 1],
       [1, 0]])
```

TIP

This is a very important matrix in quantum computing as well! We will see much more of this matrix throughout the book.

To apply the linear function represented by a matrix to a particular vector, we multiply the matrix and the vector, as illustrated in [2.12](#).

WARNING

While the order in which we add vectors doesn't matter, the order in which we multiply matrices matters quite a lot. If we rotate our map by 90° and then look at it in the mirror, we'll get a very different picture than if we rotate what we see in the mirror by 90°. Both rotation and flipping are linear functions, and so we can write down a matrix for each; let's call them R and F , respectively. If we flip a vector \vec{x} , we get $F\vec{x}$. Rotating the output gives us $RF\vec{x}$, a very different vector than if we rotated first, $FR\vec{x}$.

Figure 2.12. How to multiply a matrix by a vector: In this example, the matrix for f tells us that $f([[1], [0], [0]])$ is $[[1], [2], [9]]$.

$$\begin{matrix} f([1], [0], [0]) \\ f([0], [1], [0]) \\ f([0], [0], [1]) \end{matrix} \begin{bmatrix} 1 & 3 & 7 \\ 2 & 4 & 6 \\ 9 & 8 & 5 \end{bmatrix} \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 20 \\ 22 \\ 45 \end{bmatrix} = \begin{matrix} 1 \times 3 + 3 \times 1 + 7 \times 2 \\ 2 \times 3 + 4 \times 1 + 6 \times 2 \\ 9 \times 3 + 8 \times 1 + 5 \times 2 \end{matrix}$$

A matrix describing a linear function f can be thought of as a stack of the outputs of f , one for each row.

Just as the first index of a matrix represents its rows and the second index represents its columns, the first matrix being multiplied is read out row-by-row.

The second matrix or vector being multiplied is read out in columns.

Since the first factor had three rows and the second factor had one column, the product has three rows and one column.

Matrix multiplication formalizes the way that we computed f given its outputs for a particular set of inputs by "stacking up" the outputs of f for vectors like $[[1], [0], [0]]$ and $[[0], [1], [0]]$, as illustrated in [2.12](#). While the actual sizes of the matrices and vectors may change, this idea that a matrix can describe a linear transformation stays the same. For the rest of this chapter, we will look linear transformations on vectors of length 2. We can think of each row (the outermost index in NumPy) of a

matrix as how the function acts on a particular input.

DEEP DIVE: Why do we multiply functions?

When we multiply a matrix by a vector (or even by a matrix by a matrix), we're doing something that seems a bit odd at first. After all, matrices are another way of representing linear functions, so what does it mean to multiply a function by its input, let alone by another function?

To answer this, it's helpful to go back to ordinary algebra for a moment, in which we have that for any variables a , b , and c , $a(b + c) = ab + ac$. This property, known as the *distributive property* is fundamental to how multiplication and addition interact with each other. In fact, it's so fundamental, that the distributive property is one of the key ways we define what multiplication is—in number theory and other more abstract parts of math, researchers often work with objects known as *rings*, where all we really know about multiplication is that it distributes over addition. Though posed as an abstract concept, the study of rings and other similar algebraic objects has broad applications, especially in cryptography and error correction.

The distributive property looks very similar to the linearity property, though, that $f(x + y) = f(x) + f(y)$. If we think of f as being a part of a ring, then the distributive property is identical to the linearity property. Put differently, as much as programmers like to reuse code, mathematicians like to reuse concepts. Thinking of multiplying matrices together lets us treat linear functions in the many of the same ways as we're used to from algebra.

Thus, if we want to know the $-i$ th element of a vector x that has been rotated by a matrix M , we can find the output of M for each element in X -, sum the resulting vectors, and take the i th element. In NumPy, matrix multiplication is represented by the @ operator.

NOTE

The code sample below only works in Python 3.5 or later.

Listing 2.6. Matrix multiplication with the @ operator

```
>>> M = np.array([
...     [1, 1],
...     [1, -1]
... ], dtype=complex)
>>> M @ np.array([[2], [3]], dtype=complex)
array([[ 5.+0.j],
       [-1.+0.j]])
```

Why NumPy?

We could have written all of the matrix multiplication above out by hand, but there's a few reasons that it's very nice to work with NumPy instead. Most of the core of NumPy uses constant-time indexing, and is implemented in native code, such that it can take advantage of built-in processor instructions for fast linear algebra. Thus, NumPy will often be much, much faster than manipulating lists by hand.

In Listing 2.7, we show an example where NumPy can speed up multiplying even very small matrices by 10 \times . As we get to larger matrices in Chapters 4 and later, using NumPy over doing things by hand gives us even more of an advantage.

Listing 2.7. Timing NumPy evaluation for matrix multiplication

```

$ ipython
In [1]: def matmul(A, B):
...:     n_rows_A = len(A)
...:     n_cols_A = len(A[0])
...:     n_rows_B = len(B)
...:     n_cols_B = len(B[0])
...:     assert n_cols_A == n_rows_B
...:     return [
...:         [
...:             sum(
...:                 A[idx_row][idx_inner] * B[idx_inner][idx_col]
...:                 for idx_inner in range(n_cols_A)
...:             )
...:             for idx_col in range(n_cols_B)
...:         ]
...:         for idx_row in range(n_rows_A)
...:     ]
...:
In [2]: import numpy as np
In [3]: X = np.array([[0+0j, 1+0j], [1+0j, 0+0j]])
In [4]: Z = np.array([[1+0j, 0+0j], [0+0j, -1+0j]])
In [5]: matmul(X, Z)
Out[5]: [[0j, (-1+0j)], [(1+0j), 0j]]
In [6]: X @ Z
Out[6]:
array([[ 0.+0.j, -1.+0.j],
       [ 1.+0.j,  0.+0.j]])
In [7]: %timeit matmul(X, Z)
10.3 µs ± 176 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
In [8]: %timeit X @ Z
926 ns ± 4.42 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

```

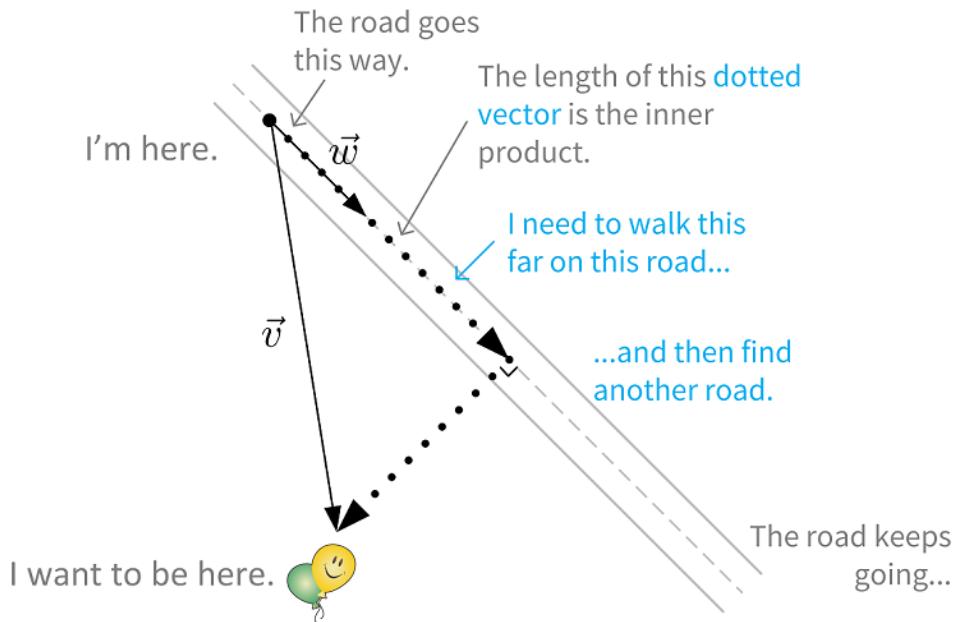
- ➊ This time, we'll use the IPython interpreter for Python, as it provides a few extra tools that are helpful in this example. Please see Appendix A for instructions on how to install IPython.
- ➋ We start by finding the sizes of each matrix that we need to multiply. If we're representing matrices by lists of lists, then each element of the outer list is a row. That is, an $n \times m$ matrix has n rows and m columns when written out this way.
- ➌ The inner dimensions of both matrices need to agree in order for matrix multiplication to make sense. Thinking of each matrix as representing a linear function, the first index (the number of rows) tells us how large each output is, while the second index (the number of columns) tells us how large each input is. Thus we need the outputs from the first function to be applied (the one on the right) to be of the same size as the inputs to the second function. This line checks that condition.
- ➍ To actually compute the matrix product of A and B, we need to compute each element in the product and pack them into a list of lists.
- ➎ We can find each element by summing over where the output from B is passed as input to A, similar to how we represented the product of a matrix with a vector in 2.12.
- ➏ For comparison, we can import NumPy, which provides us with a matrix multiplication implementation that uses modern processor instructions to accelerate the computation.
- ➐ We'll initialize two matrices as NumPy arrays as test cases. We'll see much more about these two particular matrices throughout the book.

- ➊ Matrix multiplication in NumPy is represented by the @ operator in Python 3.5 and later.
- ➋ The %timeit "magic command" tells IPython to run a small piece of Python code many times and report the average amount of time that it takes.

2.4.2 Party with inner products

There's one last thing we need to worry about in finding the party. Earlier, I said I was ignoring the problem of whether there was a road that would let me go in the direction I needed to, but this is a really bad idea when I'm wandering through an unfamiliar city. To make my way around, I need a way to evaluate how far I should walk along a given road to get where I'm going. Thankfully, linear algebra gives us a tool, the *inner product* to do just that. Inner products are a way of projecting one vector \vec{v} onto another vector \vec{w} , telling us how much of a "shadow" \vec{v} casts on \vec{w} .

Figure 2.13. How to find a party with inner products.



We can compute the inner product of two vectors by multiplying their respective elements and summing the result. Note that this multiply-and-sum recipe is the same as what we do in matrix multiplication! Multiplying a matrix that has a single row with a matrix that has a single column does exactly what we need. Thus, to find the projection of \vec{v} onto \vec{w} , we need to turn \vec{v} into a row vector by taking its *transpose*, written \vec{v}^T .

Example 2.3

The transpose of $\vec{w} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$ is $\vec{w}^T = [2 \ 3]$.

NOTE

Later, we'll see that we also need to take the complex conjugate of each element, but we'll set that aside for now.

In particular, the matrix product of \vec{v}^T (the transpose of \vec{v}) with \vec{w} gives us a 1×1 matrix containing the inner product we want. Suppose I need to go 2 blocks south and 3 blocks east, but I can only go on a road that points more south-southeast. Since I still need to travel south, this road helps me get where I need to go. But how far should I walk before this road stops helping?

TIP**Square roots and lengths**

The square root of a number x is a number $y = \sqrt{x}$ such that we get x back when we square y , $y^2 = x$. We'll use the square root a lot throughout the book, as square roots are essential to finding the length of vectors. In computer graphics, for instance, quickly finding the lengths of vectors is essential to making games work (see en.wikipedia.org/wiki/Fast_inverse_square_root for some fun history about how square roots are used in gaming).

Whether vectors describe how we get to parties, or as we'll see later, those vectors describe the information that a quantum bit represents, we'll use square roots to reason about their lengths.

Listing 2.8. Computing vector dot products with NumPy

```
>>> import numpy as np
>>> v = np.array([-2, -3])
>>> south_east = np.array([1, -1])
>>> np.linalg.norm(south_east)
1.4142135623730951
>>> w = np.array([1, -1]) / np.sqrt(2)
>>> np.linalg.norm(w)
0.9999999999999999
>>> v.transpose()
array([-2, -3])
>>> v.transpose() @ w
array([0.70710678])
```

1
2
3
4
5
6
7
8
9

- ➊ In this case, \vec{v} is the vector describing where I need to go, namely two blocks north and three blocks east.
- ➋ If the road available points southeast, then it goes one block south for every block east that it goes.
- ➌ We can find the length of a vector using Pythagoras' theorem by taking the sum of the absolute values of each element, then taking the square root. In NumPy, this is done with the `np.linalg.norm` function, as the length of a vector is sometimes also called its norm.

- ④ The length of $[[1], [-1]]$ is thus $\sqrt{(1)^2 + (-1)^2} = \sqrt{2} \approx 1.4142$.
- ⑤ Thus, when we define \vec{W} to be the *direction* southeast, we need to divide by $\sqrt{2}$.
- ⑥ Checking, we see that the length of \vec{W} is now approximately 1.
- ⑦ The transpose turns $\vec{V} = [[-2], [-3]]$ into the "row" $[-2, -3]$.
- ⑧ We can then multiply the transpose of \vec{V} with \vec{W} the same way as we multiplied matrices with vectors earlier.
- ⑨ Doing so, we see that I need to walk $1 / \sqrt{2} \approx 0.707$ blocks along this road before it stops helping me to the party.

Finally we have made it to the party (only slightly late) and are ready to try out that new doorbell!

2.5 Qubits: States and Operations

Thankfully, the party was *lovely*, and gave us a chance to see our friends again and even to make some acquaintances! The only thing that could have made it better is if Eve could have been there. Rather than feeling lonely, though, after such a lovely party, let's use **qubits** to tell Eve how we feel!

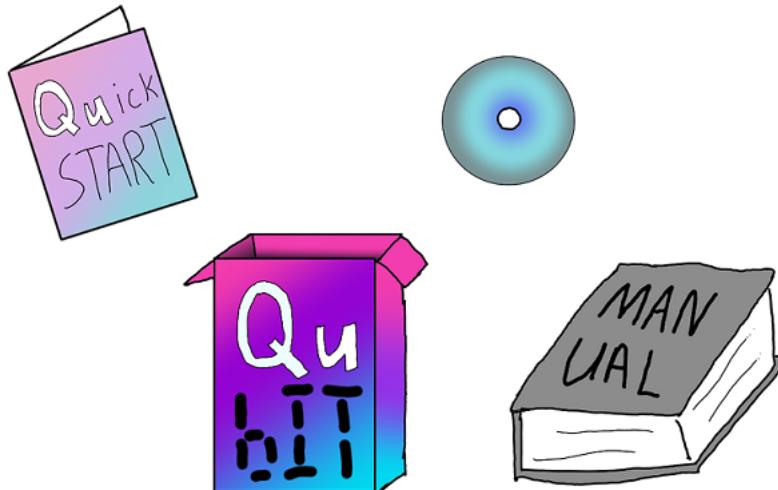
Qubits are the basic unit of information in a quantum computer. They can be physically implemented by systems that have 2 states like classical bits, but that behave according to the laws of quantum mechanics, which allows for some behaviors that classical bits are not capable of. Let's treat qubits like you would any other fun and new computer part: plug it in and see what happens!

Scenario

Say you just got a brand new computer part for your desktop. You have a thick manual, a quick start guide, and a driver CD. What steps do you take to figure out how to set it up correctly and verify that it works ok?

The easiest approach would be to just plug it in and hope it works! But some very hard working technical writers and devs worked to put together that quick start guide and system requirements... We will treat the rest of this chapter as a sort of quick start guide to qubits, so let's see if they work!

Figure 2.14. Our brand new quantum development (the CD they provide is just a novelty coaster).



IMPORTANT

Simulated Qubits

For almost all of this book, we won't be using actual qubits, but will be using classical simulations of qubits. This lets us learn how quantum computers will work, and to get started programming small instances of the kinds of problems that quantum computers can solve, even if we don't yet have access to the kinds of quantum hardware we'll need to solve practical problems.

The trouble with this is that simulating qubits on classical computers takes an exponential amount of classical resources in the number of qubits, such that the most powerful classical computing services can simulate up to about 40 qubits before having to simplify or reduce the types of quantum programs being run. For comparison, current commercial hardware maxes out at about 70 qubits at the time of this writing. Devices with that many qubits are extremely difficult to simulate with classical computers, but currently available devices are still too noisy to complete most useful computational tasks.

Imagine having to write a classical program with only 40 classical bits to work with! While 40 bits is quite small compared to the gigabytes that we are used to working with in classical programming, there are still some really interesting things we can do with only 40 qubits, and that help us prototype what an actual quantum advantage might look like.

2.5.1 State of the qubit

Looking through the listed system requirements for our quantum hardware, the first thing listed is *qubits*, followed by a bunch of stuff like "initial state: $|0\rangle$ ". This is more helpful than it might first seem, as our manual is telling us about how to describe and initialize our new qubits. We have used locks, baseballs, and other classical systems to represent our classical bit values of 0 or 1. There are many physical systems we can

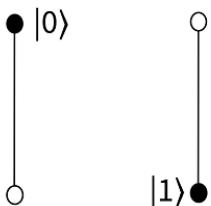
use to act as our qubit, and *states* are the "values" our qubit can have.

Similar to the 0 and 1 states of classical bits, we can write labels for quantum states. The qubit states that are most similar to the classical "0" and "1" are $|0\rangle$ and $|1\rangle$, as we draw in 2.15. These are referred to as "ket 0" and "ket 1" respectively. With this in mind, the " $|0\rangle$ " in our manual tells us what state our qubits start off in when we pull them out of the box.

NOTE
Ket?

The term "ket" comes from a kind of whimsical naming that is seen in quantum computing owing its history to a particularly silly pun. As we'll see more of when we look at measurements, there's another kind of object called a *bra* that gets written like $\langle 0|$. When you put a bra and a ket together, you get a pair of brackets $\langle \rangle$. The use of bras and kets to write out math for quantum mechanics is often called *Dirac notation* after Paul Dirac, who both invented the notation and the truly groan-worthy pun that we're now stuck with. We will see more of this style of whimsey throughout the book.

Figure 2.15. Bracket notation for qubits.



One thing to be mindful of, though, is that a state is a convenient model use to predict how a qubit behaves, not some inherent property of the qubit itself. This distinction becomes especially important when we consider measurement later in the chapter — as we will see, we cannot directly measure the state of a qubit.

IMPORTANT

In real systems, we will never be able to extract or perfectly learn the state of a qubit given a finite number of copies.

Don't worry if this doesn't all make sense yet, we'll see plenty of examples as we go through the book. What's important to keep in mind for now is that qubits aren't states.

If we want to simulate how a baseball moves once it's thrown, we might start by writing down it's current location, how fast it's going and in what direction, which way it's spinning, and so forth. That list of numbers helps us represent a baseball on a piece of paper or in a computer so that we can predict what that baseball will do, but we wouldn't say that the baseball **is** that list of numbers. To get our simulation started, we'd have to take a baseball we're interested in and *measure* where it is, how fast it's going, and so forth.

We say that the full set of data we need to accurately simulate the behaviour of a baseball is the *state* of that baseball. Similarly, the state of a qubit is the full set of data we need to simulate it and to predict what results we'll get when we measure it. Just as we need to update the state of a baseball in our simulator as it goes along, we'll update the state of a qubit when we apply operations to it.

TIP**The map is not the territory**

One way to remember this subtle distinction is that a qubit **is described by a state** and but it is not true that a qubit **is a state**.

Where things get a little more subtle is that while we can measure a baseball without doing anything to it other than copying some classical information around, as we'll see throughout the rest of the book, we can't perfectly copy the quantum information stored in a qubit—when we measure a qubit, we have an effect on its state. This can be sometimes confusing, as we record the full state of a qubit when we simulate it, such that we could look at the memory in our simulator whenever we want. There's nothing we can do with actual qubits that lets us look at their state, so if we "cheat" by looking at the memory of a simulator, we won't be able to run our program on real hardware.

Put differently, while it can be useful for debugging our classical simulators as we are building them to look at states directly, we have to make sure we are only writing algorithms based on information we could plausibly learn from real hardware.

NOTE**Cheating with our eyes shut**

As mentioned above, when we are using a simulator, the simulator must store the state of our qubits internally—this is why simulating quantum systems requires so much work. Every qubit could in principle be correlated with every other qubit, so we need exponential resources in general to write down the state in our simulator (we'll see more about this in Chapter 4). If we "cheat" by looking directly at the state stored by a simulator, then we can only ever run our program on a simulator, not on actual hardware. We'll see in later chapters how to cheat more safely, by using assertions and by making cheating unobservable. 😊

2.5.2 *The game of Operations*

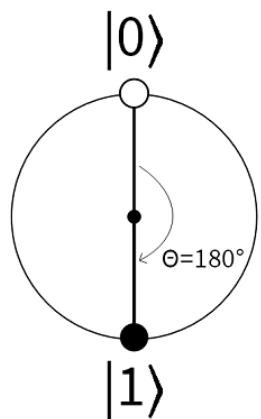
Now that we have names for these states, let's show how to represent the information they contain. With classical bits we can record the information contained in the bit at any time as simply a value on a line: 0 or 1. This worked because the only operations we could do consisted of flips (or 180° rotations) on this line. Quantum mechanics allows us to apply more kinds of operations to qubits, including rotations by less than 180° . That is, qubits differ from classical bits in what operations we can do with them.

IMPORTANT

While operations on classical bits are logical operations that can be made by combining NOT, AND, and OR in different ways, quantum operations consist of *rotations*.

For instance, if we want to turn the state of a qubit from $|0\rangle$ to $|1\rangle$ and vice-versa, the quantum analogue of a NOT operation, we rotate the qubit clockwise by an angle of 180° .

Figure 2.16. A visualization of the quantum equivalent of a NOT operation operating on a qubit in the $|0\rangle$ state, leaving the qubit in the $|1\rangle$ state.



The quantum analogue to the classical NOT gate is equivalent to a rotation of the state of the qubit of 180° .

$$\text{NOT } |0\rangle = |1\rangle$$

We have seen how rotation by 180° is the analogue to a NOT gate, but what other rotations can we do?

IMPORTANT**Reversibility**

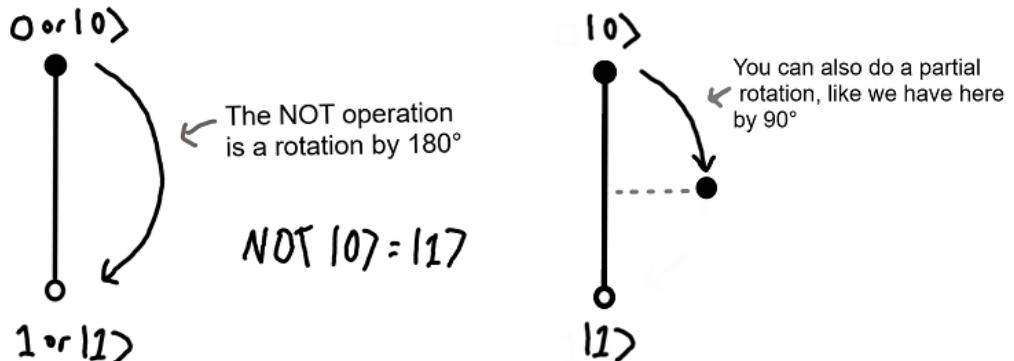
When we rotate a quantum state, we can always get back to the same state we started with by rotating backwards. This property, known as reversibility, turns out to be fundamental to quantum computing. With the exception of measurement, which we'll learn more about later in this chapter, all quantum operations must be reversible.

Not all of the classical operations that we're used to are reversible, though. Operations like AND and OR aren't reversible as they are typically written, so they cannot be implemented as quantum operations without a little bit more work. We'll see how to do this in Chapter 6 when we introduce the "uncompute" trick for expressing other classical operations as rotations.

On the other hand, classical operations like XOR can easily be made reversible, so we can write them out as rotations using a quantum operation called the "controlled NOT" operation, as we will see in Chapter 4.

If rotate a qubit in the $|0\rangle$ state clockwise by 90° instead of 180° , we get a quantum operation that we can think of as a "square root" of a NOT operation.

Figure 2.17. Rotating a state by less than 180 degrees.



In the same way as we earlier defined the square root \sqrt{x} of a number x as being a number y such that $y^2 = x$, we can define the square root of a quantum operation. If we apply a 90° rotation twice, we get back the NOT operation, so we can think of the 90° rotation as the square root of NOT.

NOTE

Halves and Half-Not

Every field has its stumbling blocks. Ask a graphics programmer whether positive y means "up" or "down," for instance. In quantum computing, the rich history and interdisciplinary nature of the field sometimes comes across as a double-edged sword in that each different way of thinking about quantum computing comes with its own conventions and notations.

One way this manifests is that it's really easy to make mistakes with where to put factors of two. In this book, we've chosen to follow the convention used by Microsoft's Q# language.

We now have a new state that is neither $|0\rangle$ nor $|1\rangle$, but an equal combination of them both. In precisely the same sense that we can describe "northeast" by adding the directions "north" and "east," we can write this new state as shown in 2.18.

Figure 2.18. Rotating a state by 90°.

In the same way we might think of directions like "north" and "east," quantum states like $|0\rangle$ and $|1\rangle$ are directions.

Just as we can point ourselves "northeast" by looking north and then rotating towards the east, we get a new state that points between $|0\rangle$ and $|1\rangle$ by starting in $|0\rangle$ and rotating towards $|1\rangle$.

$$\cos(90^\circ/2) |0\rangle + \sin(90^\circ/2) |1\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$$

We can rotate between states using the same math we use to describe rotations of map directions; we just have to watch out for factors of 2.

For example, if we want to rotate the $|0\rangle$ state by 90°, we use the cosine and sine functions to find the new state.

Definition 2.6: $|+\rangle$ and $|-\rangle$

We call this state the $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ state (due to the sign between the terms).

We say that the $|+\rangle$ state is a *superposition* of $|0\rangle$ and $|1\rangle$.

If the rotation was by a -90° (anti-clockwise), then we call the resulting state $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$ instead.

Try writing out the rotations above using -90° to see that you get $|-\rangle$!

A mouthful of math

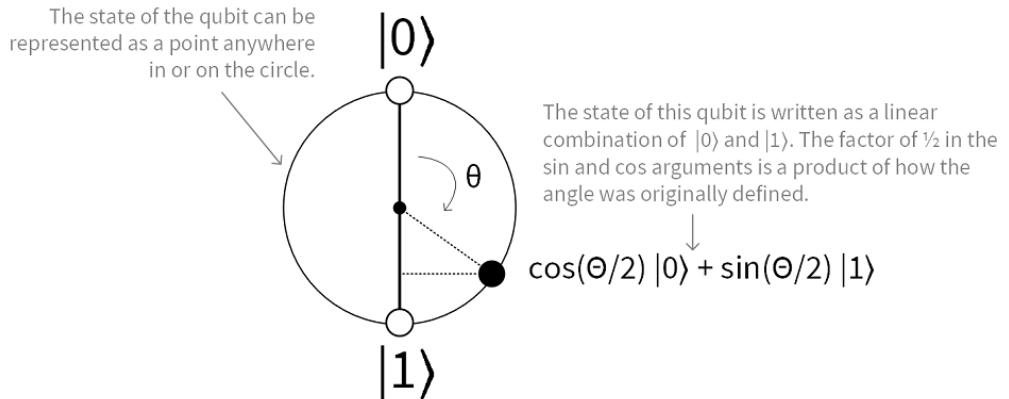
At first glance, something like $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ would be terrible to have to say out loud, making it rather useless in conversation. In practice, however, quantum programmers often take some shortcuts when speaking outloud or sketching things out at the whiteboard.

For instance, the " $\sqrt{2}$ " part always has to be there, since vectors representing quantum states always have to be length one; that means we can sometimes be a little bit casual and write things like " $|+\rangle = |0\rangle + |1\rangle$," relying on our audience to remember to divide by $\sqrt{2}$. If we're giving a talk, or discussing quantum computing over some nice tea, we may say this as "ket plus is ket 0 plus ket 1," but the reuse of the word "plus" gets a little confusing without bras and kets to help. To emphasize verbally that addition allows us to represent superposition, we might say "the plus state is an equal superposition of zero and one" instead.

The state of a qubit can be represented as a point on a circle that has two labeled states on the poles: $|0\rangle$ and $|1\rangle$

More generally, we will picture rotations by arbitrary angles θ between qubit states as follows in [2.19](#).

Figure 2.19. A visualization of the state of a qubit.



Mathematically, we can write the state of any point on the circle that represents our qubit as $\cos(\theta/2) |0\rangle + \sin(\theta/2) |1\rangle$, where $|0\rangle$ and $|1\rangle$ are different ways of writing the vectors $[[1], [0]]$ and $[[0], [1]]$ respectively.

TIP

One way to think of ket notation is as giving *names* to vectors that we commonly use. When we write $|0\rangle = [[1], [0]]$, we're saying that $[[1], [0]]$ is important enough that we name it after "0." Similarly, when we wrote that $|+\rangle = [[1], [1]] / \sqrt{2}$, we gave a name to the vector representation of a state that we will use all the time throughout the book.

Another way to say this would be a qubit is generally the *linear combination* of the vectors of $|0\rangle$ and $|1\rangle$ with coefficients that describe the angle that $|0\rangle$ would have to be rotated to get to the state. To write this in a way that is useful for programming, we can write out how rotating a state affects each of the $|0\rangle$ and $|1\rangle$ states:

Let's look at rotating $|0\rangle$ by an angle θ again, and see how we can write it out even when we don't know what θ is.

As before, we start by writing out the rotation using sines and cosines.

Sometimes, using matrix notation instead of Dirac notation (kets) is more helpful. We can use that $|0\rangle = [[1], [0]]$ here, since both are ways of writing down the same qubit state.

$$\begin{aligned} \cos(\theta/2) |0\rangle + \sin(\theta/2) |1\rangle &= \cos(\theta/2) \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \sin(\theta/2) \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos(\theta/2) \\ \sin(\theta/2) \end{bmatrix} \end{aligned}$$

Similarly, $|1\rangle = [[0], [1]]$.

Once we've written each state using matrix notation, we can just add the corresponding elements together.

For instance, we get $\cos(\theta/2)$ for the first row, since $\cos(\theta/2) + 0 = \cos(\theta/2)$.

TIP This is precisely the same as when we used a basis of vectors earlier to represent a linear function as a matrix.

There are other quantum operations that we will learn about in this book, but these are the easiest to visualize as rotations. Here is a table summarizing the states we have learned to create from these rotations:

Table 2.2. Table showing state labels, expansions in Dirac notation, and representations as vectors.

State label	Dirac notation	Vector representation
$ 0\rangle$	$ 0\rangle$	$[[1], [0]]$
$ 1\rangle$	$ 1\rangle$	$[[0], [1]]$
$ +\rangle$	$(0\rangle + 1\rangle)/\sqrt{2}$	$[[1/\sqrt{2}], [1/\sqrt{2}]]$
$ -\rangle$	$(0\rangle - 1\rangle)/\sqrt{2}$	$[[1/\sqrt{2}], [-1/\sqrt{2}]]$

2.5.3 Measuring Qubits

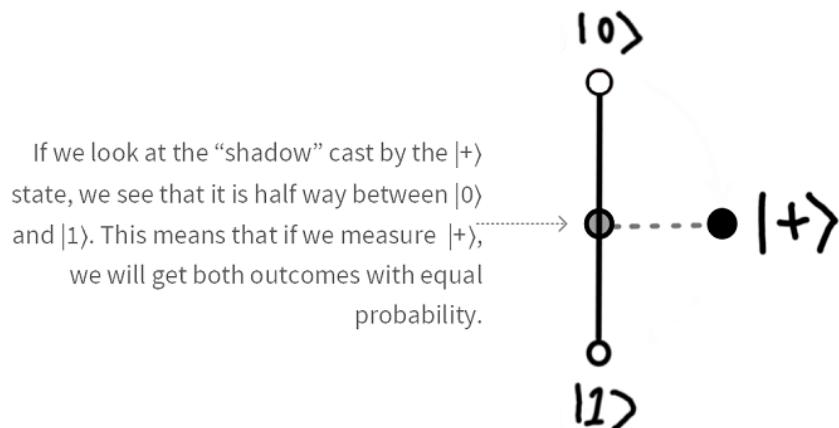
When we actually want to retrieve the information stored in a qubit, we need to measure it. Ideally, we would like a measurement device that would let us directly read out all the information about the state at once. As it turns out, this is not possible by the laws of quantum mechanics.

That said, measurement *can* allow us to learn information about the state relative to particular directions in the system. For instance, if we have a qubit in the $|0\rangle$ state and we look to see if it is in the $|0\rangle$ state, we'll always get that it is.

On the other hand, if we have a qubit state in the $|+\rangle$ state and we look to see if it is in the $|0\rangle$ state, we'll get a "0" outcome with 50% probability.

This is because the $|+\rangle$ state overlaps equally with the $|0\rangle$ and $|1\rangle$ states, such that we'll get both outcomes with the same probability.

Figure 2.20. The $|+\rangle$ state overlaps equally with both $|0\rangle$ and $|1\rangle$, because the "shadow" it casts is exactly in the middle.

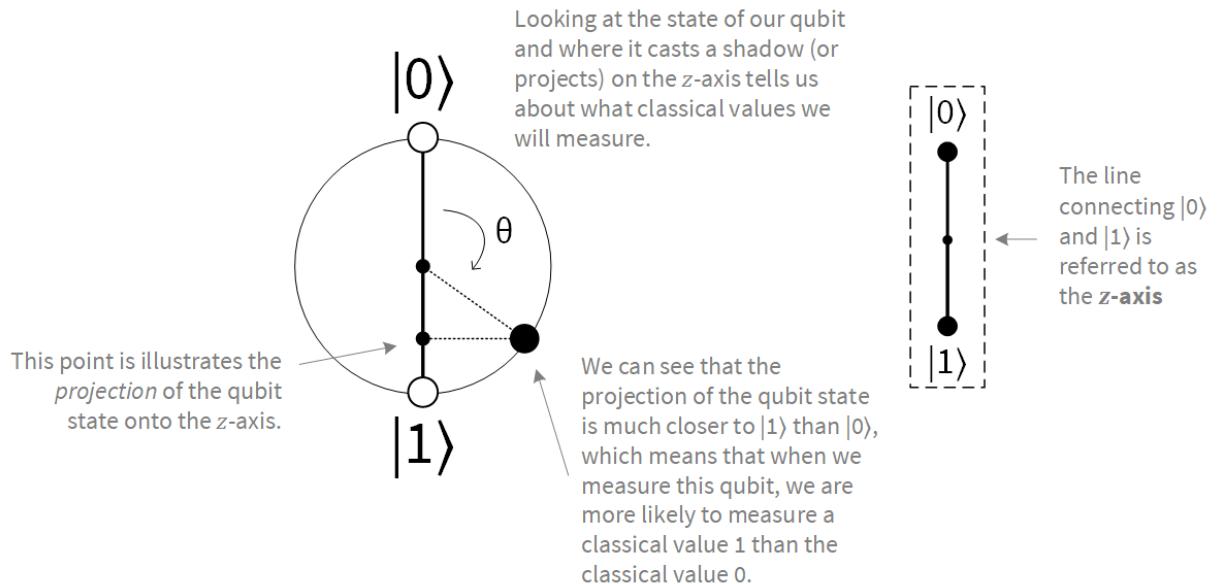


IMPORTANT

Measurement outcomes of qubits are classical bit values! Put differently, whether we measure a classical bit or a qubit, our result is always a classical bit.

Most of the time, we will choose to measure whether we have a $|0\rangle$ or a $|1\rangle$; that is, we'll want to measure along the line between the $|0\rangle$ and $|1\rangle$. For convenience, we give this axis a name, calling it the Z-axis. We can visually represent this by *projecting* our state vector onto the Z-axis (see 2.22), using the inner product we saw earlier. Think of shining a flashlight from where we draw the state of a qubit back onto the Z-axis; the probability for getting a 0 or 1 result is determined by shadow the state leaves on the Z-axis.

Figure 2.21. A visualization of a quantum measurement, which can be thought of as projecting the state along a particular direction.



DEEP DIVE: Why isn't measurement linear?

It may seem odd, after having made such a big deal of the linearity of quantum mechanics, that we immediately introduce measurement as being non-linear. If we're allowed non-linear operations like measurement, can we also implement other non-linear operations like cloning qubits?

The short version is that while everyone agrees on the math behind measurement, there's still a lot of philosophical discussion about the best way to understand why quantum measurement acts the way that it does. These discussions fall under the name of *quantum foundations*, and attempt to do more than simply understand what quantum mechanics is and what it predicts, by also understanding *why*. For the most part, foundations explores different ways to *interpret* quantum mechanics. In the same way that we can understand classical probability by considering counterintuitive thought experiments such as game show strategies or how casinos can win even from games that seem to lose money, quantum foundations develops new interpretations through small thought experiments that probe at different aspects of quantum mechanics. Thankfully, some of the results from quantum foundations can help us make sense of measurement.

In particular, one critical observation is that you can always make quantum measurements linear again by including the state of the measurement apparatus into your description; we'll see some of the mathematical tools needed to do so in Chapters 4 and 6. When taken to its extreme, this observation leads to interpretations such as the *many-worlds interpretation*. The many-worlds interpretation solves the interpretation of measurement by insisting that we only consider states that include measurement devices, such that the apparent nonlinearity of measurement doesn't really exist.

At the other extreme, we can interpret measurement by noting that the nonlinearity in quantum measurement is precisely the same as in a branch of statistics known as Bayesian inference. Thus, quantum mechanics only appears nonlinear when we forget to include that there is an agent performing the measurement who then learns from each result. This observation leads to thinking of quantum mechanics not as a description of the world, but as a description of what we know about the world.

Though these two kinds of interpretations disagree at a philosophical level, both offer different ways of resolving how a linear theory such as quantum mechanics can sometimes appear to be non-linear. Regardless of which interpretation helps you to understand the interaction between measurement and the rest of quantum mechanics, you can take solace in that the results of measurement are always described by the same math and by the same simulations. Indeed, relying on simulations (sometimes sarcastically called the "shut up and calculate" interpretation) is the oldest and most celebrated of all interpretations.

What this squared length of each projection *represents* is the probability that the state you are measuring would be found along that direction. If you have a qubit in the $|0\rangle$ state and try to measure it along the direction of the $|1\rangle$ state, you will get a probability of zero, because the states are opposite each other when we draw them on a circle. Thinking in terms of pictures, the $|0\rangle$ state has no projection onto the $|1\rangle$ state — in the sense of 2.22, $|0\rangle$ doesn't leave a shadow on $|1\rangle$.

TIP

If something happens with probability 1, then that event **always** occurs. If something happens with probability 0, then that event is **impossible**. For example, the probability that a typical 6-sided die rolls a "7" is zero, since that roll is impossible. Similarly, if a qubit is in the $|0\rangle$ state, getting a "1" result from a Z-axis measurement is impossible, since $|0\rangle$ has no projection onto $|1\rangle$.

If you have a $|0\rangle$ and try to measure it along the $|0\rangle$ direction, however, you will get a probability of 1 because the states are parallel (and of length 1 by definition). Let's walk through what measuring a state that is neither parallel nor perpendicular would look like.

Example

Say you had a qubit in state $(|0\rangle + |1\rangle) / \sqrt{2}$ (same as $|+\rangle$ from our table), and you wanted to measure it or project it along the Z-axis.

Then, we can find the probability that the classical result will be a 1 by projecting $|+\rangle$ onto $|1\rangle$.

We can find the projection of one state onto another by using the *inner product* between their vector representations. In this case, we write the inner product of $|+\rangle$ and $|1\rangle$ as $\langle 1 | + \rangle$, where $\langle 1 |$ is the transpose of $|1\rangle$, and where putting the two bars against each other indicates taking the inner product.

NOTE

In the next chapter, we'll see that $\langle 1 |$ is the conjugate transpose of $|1\rangle$, but we'll set that aside for now.

We can write this out as follows.

To compute the projection, we start by writing down the “bra” that we want to project onto.

$$\langle 1 | (|0\rangle + |1\rangle) / \sqrt{2}$$

Next, we distribute the bra.

$$= (1/\sqrt{2})(\langle 1|0\rangle + \langle 1|1\rangle)$$

We can write each bra-ket pair as an inner product between two vectors.

$$= (1/\sqrt{2}) ([|0], [|1|] \cdot [|1], [|0|] + [|0], [|1|] \cdot [|0], [|1|])$$

Calculating each inner product makes things a lot simpler!

$$= (1/\sqrt{2})(0 + 1)$$

We now have the overlap between $|0\rangle$ and $|1\rangle$.

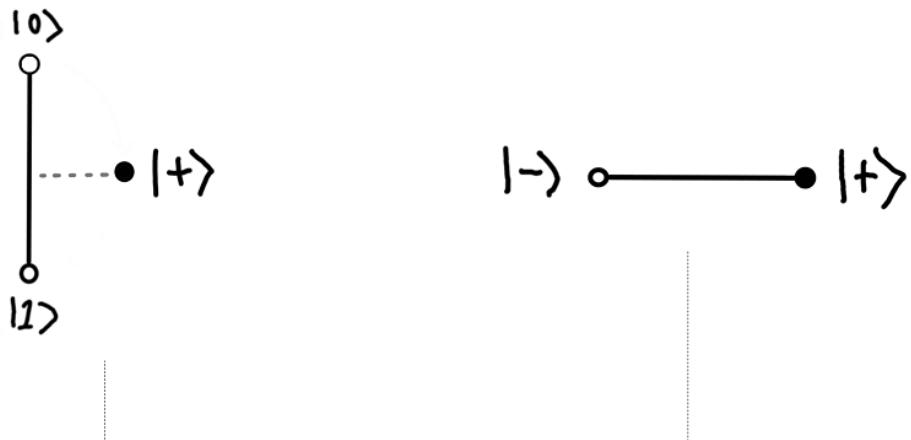
$$= 1/\sqrt{2}.$$

To turn this projection into a probability, we square it, getting that the probability of observing a "1" outcome when we prepare a $|+\rangle$ state is $\frac{1}{2}$.

We will often project onto the Z axis because it is convenient in many real experiments, but we could have also measured along the X -axis, to see if we have a $|+\rangle$ or a $|-\rangle$ state.

Measuring along the X axis, we would have gotten $|+\rangle$ with certainty and would never have gotten $|-\rangle$.

Figure 2.22. Measuring $|+\rangle$ along the X axis always results in $|+\rangle$.



The shadow that the $|+\rangle$ leaves on the Z axis is halfway between $|0\rangle$ and $|1\rangle$, so we see both outcomes with equal probability.

On the other hand, the shadow left by $|+\rangle$ on the X axis is entirely on $|+\rangle$, so we never see the outcome for $|-\rangle$.

NOTE

We can get a fully certain measurement outcome *only* because we know the "right" direction to measure ahead of time in this case—if we are simply handed a state with no information about what the "right" measurement direction is, we cannot predict any measurement outcome perfectly.

2.5.4 Generalizing measurement: basis independence

Sometimes you may not know how your qubit was prepared so you will not know how to measure the bits properly. More generally, any pair of states that don't overlap (that are opposite poles) defines a measurement in the same way. The actual outcome of a measurement is a classical bit value that indicates which pole the state is aligned with when we perform the measurement.

More general measurements still

Quantum mechanics allows for much more general kinds of measurements—we'll see a few of these as we go along, but mostly we focus in this book on the case of checking between two opposite poles. This choice is a pretty convenient way of controlling most quantum devices, and can be used in almost any of the commercial platforms for quantum computing that are currently available.

Mathematically, we use notation like " $\langle \text{measurement} | \text{state} \rangle$ " to represent measuring a qubit. The left component $\langle \text{measurement}$ is called a *bra*, and we have seen the *ket* part on the right already. So together they are called a *braket*!

Bras are very similar to *kets*, except that to switch from one to the other you have to take the transpose (turn rows to columns and vice versa) of the bra or ket you have,

$$|0\rangle^T = [[1], [0]]^T = [[1, 0]]$$

Another way to think of this is that taking the transpose turns column vectors (*kets*) into row vectors (*bras*).

NOTE

Since we're only working with real numbers for now, we won't need to do anything else to go between kets and bras, but when we work with complex numbers in the next chapter, we'll need the complex conjugate as well.

Bras let us write down measurements, but to see what measurements actually *do*, we need one more thing at our disposal: a rule for how to use a bra and a ket together to get the *probability* for seeing that measurement result. In quantum mechanics, measurement probabilities are found by looking at the length of the projection or shadow that the ket for a state leaves on a bra for a measurement. We know from our experience from the party that we can find projections and lengths using inner products. In Dirac notation, the inner product of a bra and a ket is written as $\langle \text{measurement} | \text{state} \rangle$, giving us just the rule we need.

For example, if we have prepared a state $|+\rangle$ and we want to know the probability that

we observe a "1" when we measure in the Z basis, then projecting in the way we saw with [2.22](#) we can find the the length we need.

The projection of $|+\rangle$ onto $\langle 1|$ tells us that we see a "1" outcome with probability

$$\text{Pr}(1 | +) = |\langle 1 | + \rangle|^2 = |\langle 1 | 0 \rangle + \langle 0 | 0 \rangle|^2 / 2 = |0 + 1|^2 / 2 = 1 / 2$$

Thus, 50% of the time, we'll get a "1" outcome. The other 50% of the time, we'll get a "0" outcome.

Definition 2.7: Born's rule

If we have a quantum state $|\text{state}\rangle$ and we perform a measurement along the $\langle \text{measurement} |$ direction, we can write the probability that we will observe "measurement" as our result as

$$\text{Pr}(\text{measurement} | \text{state}) = |\langle \text{measurement} | \text{state} \rangle|^2$$

In words, the probability is the square of the magnitude of the inner product of the measurement bra and the state ket.

This expression is called *Born's rule*.

In [2.3](#), we've listed several other examples of using Born's rule to predict what classical bits we will get when we measure qubits.

Table 2.3. Table Examples of using Born's rule to find measurement probabilities

If we prepare...	...and we measure...	...then we see that outcome with probability.
$ 0\rangle$	$\langle 0 $	$ \langle 0 0 \rangle ^2 = 1$
$ 0\rangle$	$\langle 1 $	$ \langle 1 0 \rangle ^2 = 0$
$ 0\rangle$	$\langle + $	$ \langle + 0 \rangle ^2 =$ $ \langle (0 + \langle 1) 0 \rangle / \sqrt{2} ^2 =$ $(1 / \sqrt{2} + 0)^2 =$ $1 / 2$
$ +\rangle$	$\langle + $	$ \langle + + \rangle ^2 =$ $ \langle (0 + \langle 1) (0\rangle + 1\rangle) / 2 ^2 =$ $ \langle 0 0 \rangle + \langle 1 0 \rangle + \langle 0 1 \rangle + \langle 1 1 \rangle ^2 / 4 =$ $ 1 + 0 + 0 + 1 ^2 / 4 = 2^2 / 4 =$ 1
$ +\rangle$	$\langle - $	$ \langle - + \rangle ^2 = 0$
$- 0\rangle$	$\langle 0 $	$ - \langle 0 0 \rangle ^2 = -1 ^2 = 1^2$
$- +\rangle$	$\langle - $	$ - \langle + - \rangle ^2 =$ $ - \langle (0 - \langle 1) (0\rangle - 1\rangle) / 2 ^2 =$ $ - \langle 0 0 \rangle - \langle 1 0 \rangle + \langle 0 1 \rangle + \langle 1 1 \rangle ^2 / 4 =$ $ -1 - 0 + 0 + 1 ^2 / 4 = 0^2 / 4 =$ 0

TIP

In [2.3](#), we used that $\langle 0 | 0 \rangle = \langle 1 | 1 \rangle = 1$ and $\langle 0 | 1 \rangle = \langle 1 | 0 \rangle = 0$. (Try checking this for yourself!) When two states have an inner product of zero, we say that they are *orthogonal* (or *perpendicular*).

That $|0\rangle$ and $|1\rangle$ are orthogonal makes a lot of calculations easier to do quickly.

We now have made it to the bottom of the Qubit quick start guide! Let's review the requirements we needed to satisfy to make sure we had working qubits.

Definition 2.8: Qubit

A qubit is any physical system satisfying three properties:

- The system can be perfectly simulated given knowledge of vector of numbers (the "state").
- The system can be transformed using quantum operations (e.g.: rotations).
- Any measurement of the system produces a single classical bit of information, following Born's rule.

Anytime we have a qubit (a system with the above three properties) we can describe it using the same math or simulation code, without further reference to what kind of system we are working with. This is similar to how we need not know whether a bit is defined by the direction of a pinball's motion or the voltage in a transistor in order to write down NOT and AND gates, or to write software which uses those gates to do interesting computation.

Phase

In the last two rows of [2.3](#), we saw that multiplying a state by a phase of -1 didn't affect measurement probabilities. This isn't a coincidence at all, but points to one of the more interesting things about qubits. Because Born's rule only cares about the squared absolute value of the inner product of a state and a measurement; multiplying a number by (-1) doesn't affect its absolute value. We call numbers such as $+1$ or -1 , whose absolute value is equal to 1 , *phases*. In the next Chapter when we work more with complex numbers, we'll see a lot more about phases.

For now, though, we say that multiplying an entire vector by -1 is an example of applying a **global phase** while changing from $|+\rangle$ to $|-\rangle$ is an example of applying a relative phase between $|0\rangle$ and $|1\rangle$.

While global phases don't ever affect measurement results, there's a big difference between the states $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ and $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$:

the coefficients in front of $|0\rangle$ and $|1\rangle$ are the same in $|+\rangle$ and are different by a phase of (-1) in $|-\rangle$. We will see much more of the difference between these two concepts in Chapters 3, 4, 6, and 7.

NOTE Similarly to how we use the word "bit" to mean both a physical system that stores information and the information stored in a bit, we will also use the word "qubit" to mean both a quantum device and the quantum information stored in that device.

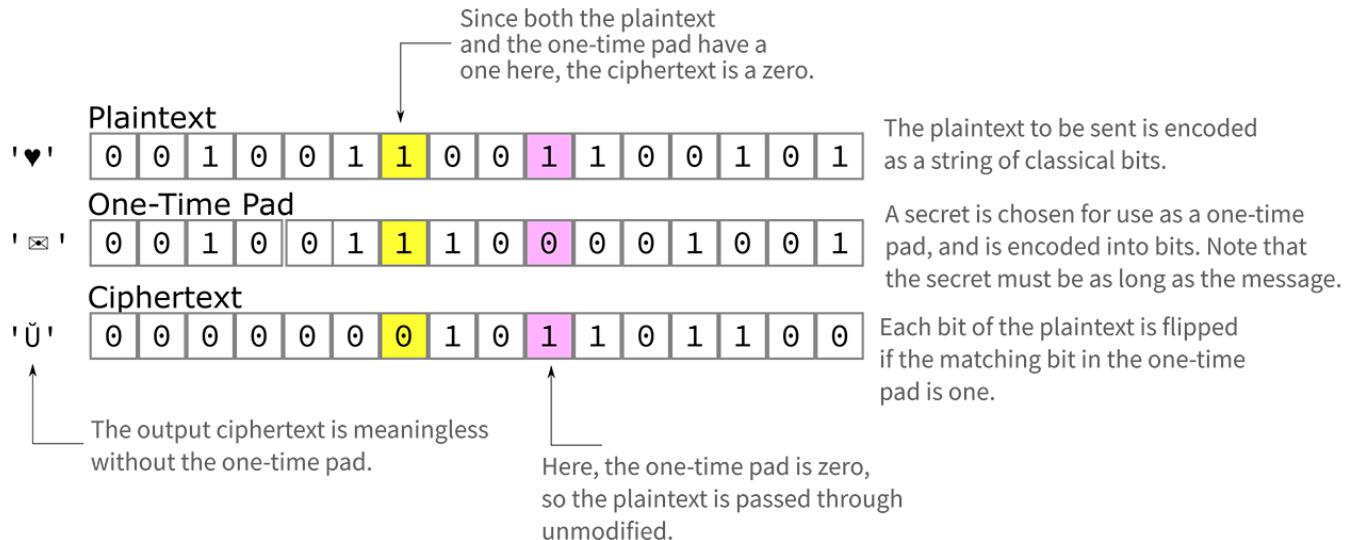
2.5.5 *Simulating qubits in code*

The quantum quick start guide lists cryptography as an initial application to check individual qubits in your new device.

Suppose you would like to keep your ❤ for Eve a secret lest anyone else finds out. How can you scramble up your message to Eve so that only she can read it?

We'll explore this application more in the next Chapter, but the most basic step we need for any good *encryption* algorithm is a source of random numbers that's difficult to predict. Let's write down exactly how we would combine our secret and random bits to make a secure message to sent to Eve. In [2.24](#), we show an example of how if both Eve and I know the same secret sequence of random classical bits, we can use that sequence to communicate securely. At the start of the chapter, we saw how we could write the message, or *plaintext*, that we want to send to Eve (in this case, "❤") as a string of classical bits. The one-time pad is a sequence of random classical bits that will act as a way to scramble or encrypt our message. This scrambling is done by taking the bitwise XOR of the message and one-time pad bits for each position in the sequence. This then produces a sequence of classical bits called the *ciphertext*. To anyone else trying to read our message, the ciphertext will just look like random bits. For example, it's impossible to tell if a bit in the ciphertext is "1" because of the plaintext or the one-time pad.

Figure 2.23. An example of how to use random bits to encrypt secrets, even over the Internet or another untrusted network.



Now you might ask how to get the random bit strings for our one-time pad? We can make our own quantum random number generator with qubits! It may seem odd but we will now simulate qubits with classical bits to make our quantum random number generator. The random numbers it will generate won't be any more secure than the computer we use to do our simulation, but it lets us get a good start in understanding qubits and how they work.

Let's send Eve our message! In the same way as a classical bit can be represented in code by the values `True` and `False`, we've seen that we can represent the two qubit states $|0\rangle$ and $|1\rangle$ as *vectors*. That is, qubit states are represented in code as lists of lists of numbers.

Listing 2.9. Representing qubits in code with NumPy

```
>>> import numpy as np      ①
>>> ket0 = np.array(      ②
...     [[1], [0]]
... )
>>> ket0
array([[1],
       [0]])               ③
>>> ket1 = np.array(
...     [[0], [1]])
... )
>>> ket1
array([[0],
       [1]])
```

- ➊ We use the NumPy library for Python to represent vectors, as NumPy is highly optimized and will make our lives much easier.
- ➋ We name our variable `ket0` after the notation $|0\rangle$, in which we label qubit states by the "ket" half of $\langle\rangle$ "brakets."
- ➌ NumPy will print out 2×1 vectors as columns.

As we saw above, we can construct other states such as $|+\rangle$ by using linear combinations of $|0\rangle$ and $|1\rangle$.

In exactly the same sense, we can use NumPy to add the vector representations of $|0\rangle$ and $|1\rangle$ to construct the vector representation of $|+\rangle$:

Listing 2.10. The vector representation of $|+\rangle$

```
>>> ket_plus = (ket0 + ket1) / np.sqrt(2) ➊
>>> ket_plus
array([[0.70710678+0.j],
       [0.70710678+0.j]]) ➋
```

- ➊ We can see NumPy using vectors to store the $|+\rangle$ state, which is a linear combination of $|0\rangle$ and $|1\rangle$.
- ➋ We will see the number 0.70710678 a lot in this book, as it is a rather good approximation to $\sqrt{2}$, the length of the vector $[[1], [1]]$.

In classical logic, if we wanted to simulate how an operation would transform a list of bits, we could use a *truth table*. Similarly, since quantum operations other than measurement are always linear, to simulate how an operation transforms the state of a qubit, we can use a matrix that tells us how each state is transformed.

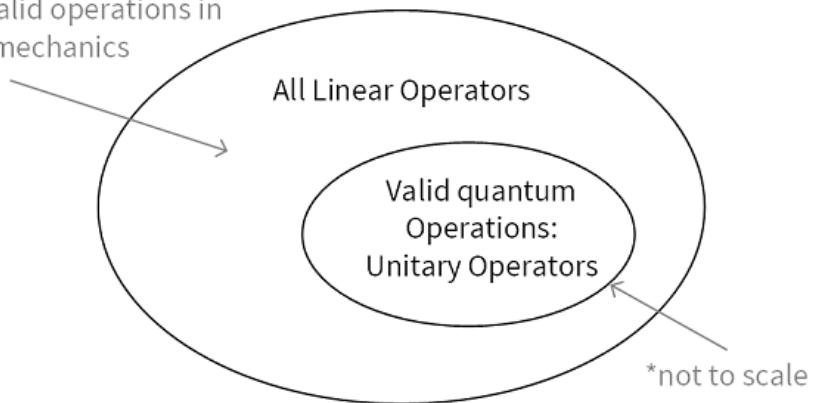
NOTE **Linear operators and quantum operations**

Describing quantum operations as linear operators is a good start, but not all linear operators are valid quantum operations! If we could implement an operation described by a linear operator such as $2 \times \mathbb{1}$ (that is, twice the identity operator), then we would be able to violate that probabilities are always numbers between zero and one. We also require that all quantum operations other than measurement are *reversible*, as this is a fundamental property of quantum mechanics.

It turns out that the operations realizable in quantum mechanics are described by matrices U whose inverses U^{-1} can be computed by taking the conjugate transpose, $U^{-1} = U^*$. Such matrices are called *unitary matrices*.

Figure 2.24. Visualizing types of valid quantum operations.

Not all linear operators describe valid operations in quantum mechanics



One particularly important quantum operation is called the *Hadamard operation*, which transforms $|0\rangle$ to $|+\rangle$ and $|1\rangle$ to $|-\rangle$.

As we saw above, measuring $|+\rangle$ along the Z-axis gives us either a "0" or a "1" result with equal probability.

Since we wanted random bits in order to send secret messages, this makes the Hadamard operation really useful for us in making our QRNG.

Using vectors and matrices, we can define the Hadamard operation by making a table of how it acts on the $|0\rangle$ and $|1\rangle$ states, as shown in [Table 2.4](#).

Table 2.4. Table Representing the Hadamard operation as a table.

Input state	Output state
$ 0\rangle$	$ +\rangle = (0\rangle + 1\rangle) / \sqrt{2}$
$ 1\rangle$	$ -\rangle = (0\rangle - 1\rangle) / \sqrt{2}$

Because quantum mechanics is linear, this is a fully complete description of the Hadamard operation!

In matrix form, we write down [2.4](#) as `H = np.array([[1, 1], [1, -1]]) / np.sqrt(2)`.

Listing 2.11. Defining the Hadamard operation

```
>>> H = np.array([[1, 1], [1, -1]]) / np.sqrt(2)           ①
>>> H @ ket0
array([[0.70710678],
       [0.70710678]])
>>> H @ ket1
array([[ 0.70710678],
       [-0.70710678]])
```

- ① We define a variable `H` to hold the matrix representation H of the Hadamard operation that we saw in [2.4](#). We'll need `H` throughout the rest of this chapter, so it's helpful to define it here.

Definition 2.9: Hadamard operation

The Hadamard operation is a quantum operation which can be simulated by the linear transformation

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Any operation on quantum data can be written as a matrix in this way. If we wish to transform $|0\rangle$ to $|1\rangle$ and vice versa (the quantum generalization of the classical NOT operation that we saw earlier, corresponding to a 180° rotation), we do the same thing as we did to define the Hadamard operation.

Listing 2.12. Representing the quantum NOT gate

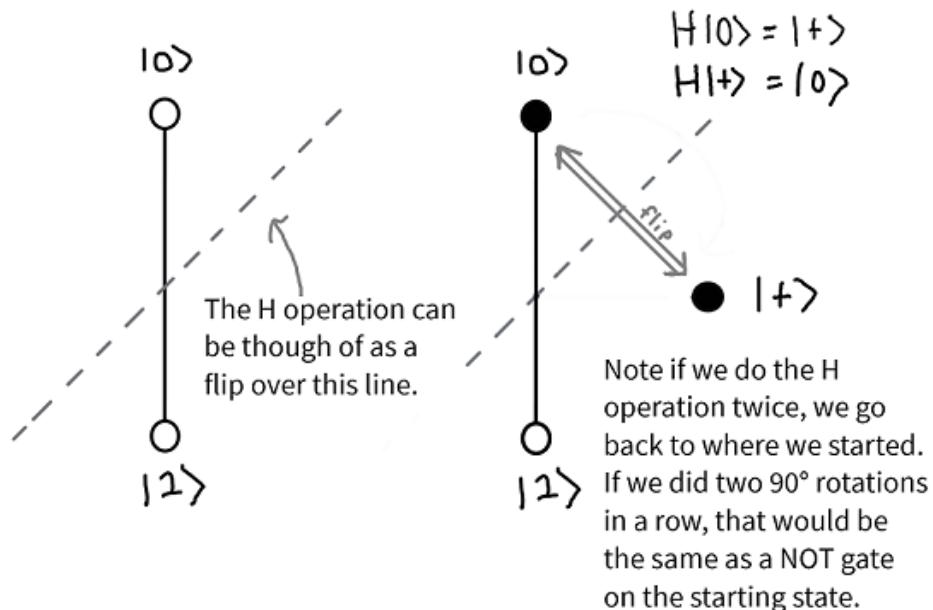
```
>>> X = np.array([[0, 1], [1, 0]])           ①
>>> X @ ket0
array([[0],
       [1]])
>>> (X @ ket0 == ket1).all()                ②
True
>>> X @ H @ ket0
array([[0.70710678],
       [0.70710678]])
```

- ① The quantum operation corresponding to the classical NOT operation is typically called the X operation; we represent the matrix for X with a Python variable `X`.
- ② We can confirm that `X` transforms $|0\rangle$ to $|1\rangle$. The NumPy method `all()` returns `True` if every element of `X @ ket0 == ket1` is `True`; that is, if every element of the array `X @ ket0` is equal to the corresponding element of `ket1`.

- ③ The X operation doesn't do anything to $H|0\rangle$, since X will swap $|0\rangle$ and $|1\rangle$ and $H|0\rangle$ is already a sum of the two kets: $(|0\rangle + |1\rangle) / \sqrt{2} = (|1\rangle + |0\rangle) / \sqrt{2}$. We can confirm this by using the `@` operator again to multiply X by a Python value representing the state $|+\rangle = H|0\rangle$. We can express that value as $H @ \text{ket}0$.

Returning to the map analogy, we can think of H as a *reflection* about the \angle direction.

Figure 2.25. The H operation as a reflection or flip about \angle .

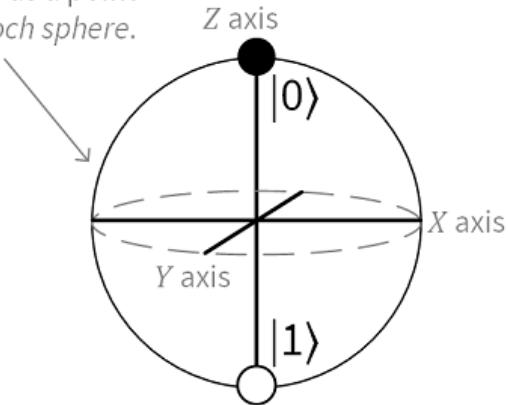


The third dimension awaits!

For qubits, the map analogy helps us understand how to write down and manipulate the states of single qubits. So far, however, we've only looked at those states that can be written down using real numbers. In general, quantum states can use complex numbers. If rearrange our map a bit and make it three-dimensional, then we can include complex numbers without any problem. This way of thinking about qubits is called the *Bloch sphere*, and can be a very useful way of thinking about quantum operations as rotations and reflections, as we'll see more of in Chapter 5.

Figure 2.26. Visualizing qubit states as points on a sphere.

The state of the qubit now is represented as a point anywhere on the *Bloch sphere*.



DEEP DIVE: Infinitely many states?

It may seem from 2.27 that there are infinitely many different states of a qubit. For any two different points on a sphere, we can always find a point that's "between" them. While this is true, it can also be a little bit misleading. Thinking of the classical situation for a moment, a coin which lands heads 90% of the time is distinct from a coin that lands heads 90.0000000001% of the time. In fact, we can always make a coin whose bias is "between" the bias of two other coins in this way. Flipping a coin can only ever give us one classical bit of information, though. On average, it would take about 10^{23} flips to tell a coin that lands heads 90% of the time apart from one that lands heads 90.0000000001% of the time. For all intents and purposes, we can treat these two coins as identical because we cannot do an experiment which reliably tells them apart. Similarly for quantum computing, there are limits to our ability to tell apart the infinitely many different quantum states that we recognize from the Bloch sphere picture.

The fact that a qubit has infinitely many states is not what makes it unique. Sometimes people say that a quantum system can be "in infinitely many states at once", which is why they say quantum computers can offer speedups. **THIS IS FALSE!** As pointed out above, we can't distinguish states that are very close together so the "infinitely many" part of the statement can't be what gives our quantum computer an advantage. We will talk more in the upcoming chapters about the "at once" part, but suffice it to say it is not the number of states that our qubit can be in that makes quantum computers cool!

2.6 Programming a Working QRNG

Now that we have a few quantum concepts to play with, let's apply what we've learned to program a quantum random number generator (QRNG) so that we can send ♥'s without a worry. We are going to build a quantum random number generator that returns either a 0 or a 1.

Random bits or random numbers?

It may seem limiting that our random number generator can only output one of two numbers, either 0 or 1. Quite to the contrary, though, this is enough to generate random numbers in the range 0 to N for any positive integer N . It's easiest to see this starting with the special case that N is $2^n - 1$ for some positive integer n , in which case we simply write down our random numbers as n -bit strings. For example, we can make random numbers between 0 and 7 by generating three random bits r_0 , r_1 , and r_2 , then returning $4r_2 + 2r_1 + r_0$.

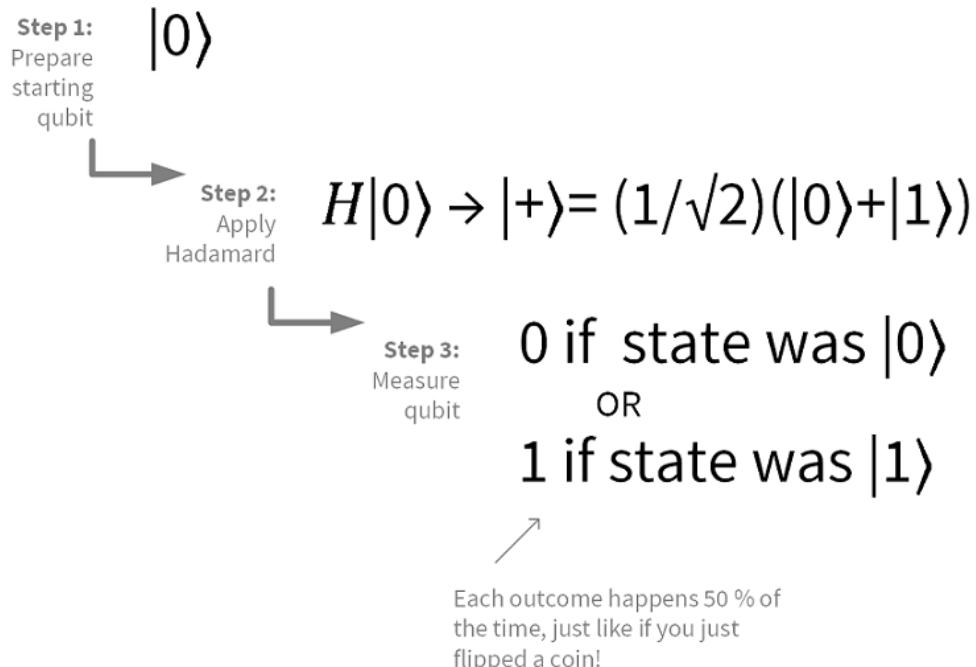
The case is slightly more tricky if N isn't given by a power of two, in that we have "left over" possibilities that we need to deal with. For instance, if we need to roll a six-sided die, but only have an eight-sided die on hand (maybe we played a Druid last time at RPG night), then we need to decide what to do when that die rolls either a 7 or an 8. The best thing we can do if we want a fair six-sided die is to simply reroll when that happens. Using this approach, we can build arbitrary fair dice from coin flips—handy for whatever game we want to play. Long story short, we aren't limited by having just two outcomes from our RNG!

As with any quantum program, our quantum random number generator program will be a sequence of instructions to a device that performs operations on a qubit. In pseudocode, a quantum program for implementing a QRNG consists of three instructions:

2.6.1 QRNG

1. Prepare a qubit in the state $|0\rangle$.
2. Apply the Hadamard operation to our qubit, so that it is in the state $|+\rangle = H|0\rangle$.
3. Measure the qubit to get either a 0 or 1 result with 50/50 probability.

Figure 2.27. Steps to writing the QRNG program we want to test out from the new hardware kit.



That is, we want a program that looks something like the following:

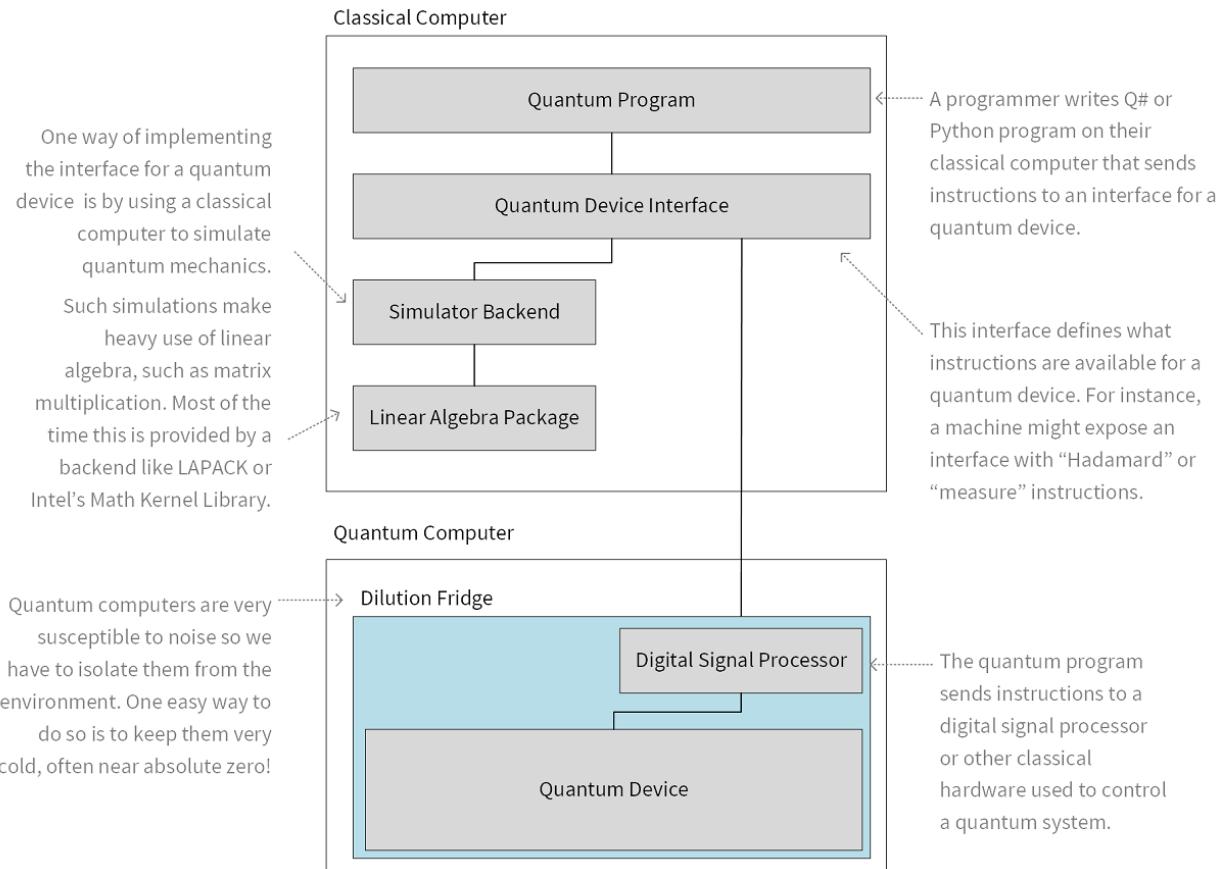
Listing 2.13. Example pseudocode for a QRNG program

```
def qrng():
    q = Qubit()
    H(q)
    return measure(q)
```

Using matrix multiplication, we can use a classical computer like a laptop to simulate how `qrng()` would act on an ideal quantum device.

Our `qrng` program calls into a software stack that abstracts away whether we're using a classical simulator or an actual quantum device.

Figure 2.28. An example of what a software stack for a quantum program might look like.



There are a lot of parts we see here to the stack, but don't worry we will talk about them as we go. For right now we are focusing on the top section (labeled "*Classical Computer*"), and will start by writing code for a quantum program as well as a simulator backend in Python.

NOTE

In Chapter 6, we'll pivot to making use of the simulator backend provided with Microsoft's Quantum Development Kit instead.

With this view of a software stack in mind, then, we can write our simulation of a QRNG by first writing a `QuantumDevice` class with abstract methods for allocating qubits, performing operations, and measuring qubits. We can then implement this class with a simulator and then call into that simulator from `qrng()`.

To design the interface for our simulator in a way that looks like 2.29, let's list out what we need our quantum device to be able to do:

2.6.2 Quantum device interface requirements.

- Users must be able to allocate and return qubits.

Listing 2.14. An example of specifying an interface into a quantum device as a set of abstract methods.

```
class QuantumDevice(metaclass=ABCMeta):
    @abstractmethod
    def allocate_qubit(self) -> Qubit:           ①
        pass

    @abstractmethod
    def deallocate_qubit(self, qubit : Qubit):      ②
        pass

    @contextmanager
    def using_qubit(self):                         ③
        qubit = self.allocate_qubit()
        try:
            yield qubit
        finally:
            qubit.reset()                           ④
            self.deallocate_qubit(qubit)
```

- ① Any implementation of a quantum device must implement this method, allowing users to obtain qubits.
- ② When users are done with a qubit, implementations of the `deallocate_qubit` will allow users to return the qubit back to the device.
- ③ We can provide a Python *context manager* to make it easy to allocate and deallocate qubits safely.
- ④ The context manager makes sure that no matter what exceptions are raised, each qubit is reset and deallocated before being returned to the classical computer.

The qubits themselves then can expose the actual transformations that we need:

2.6.3 Qubit interface requirements

- Users must be able to perform Hadamard operations on qubits.
- Users must be able to measure qubits to get out classical data.

Listing 2.15. An example of specifying an interface into the qubits on a quantum device as a set of abstract methods.

```
class Qubit(metaclass=ABCMeta):
    @abstractmethod
    def h(self): pass                            ①

    @abstractmethod
    def measure(self) -> bool: pass              ②

    @abstractmethod
    def reset(self): pass                        ③
```

- ➊ The `h` method can be implemented to transform a qubit *in place* (not making a copy) using the Hadamard operation `np.array([[1, 1], [1, -1]]) / np.sqrt(2)`.
- ➋ The `measure` method can be implemented to allow users to measure qubits and extract classical data.
- ➌ The `reset` method makes it easy for users to prepare the qubit from scratch again.

With this in place, we can return to our definition of `qrng` using these new classes.

Listing 2.16. qrng.py

```
def qrng(device : QuantumDevice) -> bool:
    with device.using_qubit() as q:
        q.h()
    return q.measure()
```

If we implement the `QuantumDevice` interface with a class called `SingleQubitSimulator`, then we can pass this to `qrng` to run our QRNG implementation on a simulator.

Listing 2.17. qrng.py

```
if __name__ == "__main__":
    qsim = SingleQubitSimulator()
    for idx_sample in range(10):
        random_sample = qrng(qsim)
        print(f"Our QRNG returned {random_sample}.")
```

We now have everything we write our `SingleQubitSimulator`. We start by defining a couple of constants for the vector $|0\rangle$ and the matrix representation of the Hadamard operation H .

Listing 2.18. simulator.py

```
KET_0 = np.array([
    [1],
    [0]
], dtype=complex) ①
H = np.array([
    [1, 1],
    [1, -1]
], dtype=complex) / np.sqrt(2) ②
```

- ➊ Since we'll be using $|0\rangle$ a lot in our simulator it helps to define a constant for it. <2 Similarly, we'll use the Hadamard matrix H to define how the Hadamard operation transforms states, so we define a constant for that as well.

Next, we define what a simulated qubit looks like. From the perspective of a simulator, a qubit wraps a vector that stores the current state of the qubit. We use a NumPy array to represent our qubit's state.

Listing 2.19. simulator.py

```

class SimulatedQubit(Qubit):
    def __init__(self):          ①
        self.reset()

    def h(self):                ②
        self.state = H @ self.state

    def measure(self) -> bool:
        pr0 = np.abs(self.state[0, 0]) ** 2      ③
        sample = np.random.random() <= pr0         ④
        return bool(0 if sample else 1)            ⑤

    def reset(self):
        self.state = KET_0.copy()

```

- ➊ As a part of the Qubit interface, we ensure that the reset method prepares our qubit in the $|0\rangle$ state. We can use that when we create the qubit to make sure that qubits always start in the correct state.
- ➋ The Hadamard operation can be simulated by applying the matrix representation H to the state that we're storing at the moment, then updating to our new state.
- ➌ We stored the state of our qubit as a vector, so we know that the inner product with $|0\rangle$ is simply the first element of that vector. For instance, if the state is $\text{np.array}([[a], [b]])$ for some numbers a and b , then the probability of observing a 0 outcome is $|a|^2$. We can find this using $\text{np.abs}(a) ** 2$. This gives us the probability that a measurement of our qubit returns 0.
- ➍ To turn the probability of getting a 0 into a measurement result, we generate a random number between 0 and 1 using np.random.random and check if it's less than $pr0$.
- ➎ Finally, we return out to the caller a 0 if we got a 0 and a 1 if we got a 1.

NOTE**What random number came first: 0 or 1?**

In making this QRNG, we'll have to call a **classical** random number generator. This may feel a bit circular, but it comes about because our classical simulation is just that: a simulation. A simulation of a quantum random number generator won't be any more random than the hardware and software we use to implement that simulator.

That said, the quantum program `qrng.py` itself does not need to call a classical RNG, but calls into the simulator. If we were to run `qrng.py` on an actual quantum device, the simulator and hence the classical RNG would be substituted out for operations on the actual qubit. At that point, we would have a stream of random numbers that would be impossible to predict thanks to the laws of quantum mechanics.

Running our program, we now get the random numbers we expected!

Listing 2.20. Output from running qrng.py

```
$ python qrng.py
Our QRNG returned False.
Our QRNG returned True.
Our QRNG returned True.
```

```
Our QRNG returned False.
Our QRNG returned False.
Our QRNG returned True.
Our QRNG returned False.
Our QRNG returned False.
Our QRNG returned False.
Our QRNG returned True.
```

Congratulations!

You've not only written your first quantum program, but you've also written a simulation backend and used it to run your quantum program in the same way as you'd run on an actual quantum computer.

DEEP DIVE: Schrödinger's Cat

You may have already seen or heard of the quantum program above, but under a very different name. Often, the QRNG program is described in terms of the "Schrödinger's cat" thought experiment: a cat is in a closed box with a vial of poison that will be released if a particular random particle decays. Before you open the box to check, how do you know if it is alive or dead?

The [state] of the entire system would express this by having in it the living and dead cat (pardon the expression) mixed or smeared out in equal parts.

– Erwin Schrödinger

Historically, Schrödinger proposed this description in 1935 to express his view that some implications of quantum mechanics are "ridiculous" by means of a thought experiment that highlights how counterintuitive these implications are. Such thought experiments, known as *gedanken*, are a celebrated tradition in physics, and can help us understand or critique different theories by pushing them to extreme or absurd limits.

In reading about Schrödinger's cat nearly a century later, however, it's helpful to remember everything that's happened in the intervening years. Since his original letter, the world has seen:

- War on a scale never before imagined,
- The first steps that humanity has taken to explore beyond our own planet,
- The rise of commercial jet travel,
- The understanding and first effects of anthropogenic climate change,
- A fundamental shift in how we communicate (television all the way through the Internet),
- A wide availability of affordable computing devices, and
- The discovery of a wondrous variety of subatomic particles.

Put simply, the world we live in isn't the same world in which Schrödinger tried to make sense of quantum mechanics. We have a lot of advantages in trying to understand, none the least of which being that we can quickly get our hands on quantum mechanics by programming simulations using classical computers. For example, the `h` instruction we saw earlier puts our qubit in a similar situation as the cat in the gedanke above, but with the advantage that it's much easier to experiment with our program than

with a thought experiment. Throughout the rest of the book, we'll make use of our quantum programs to learn the parts of quantum mechanics we need to write quantum algorithms.

2.7 Summary

In this chapter you learned:

- Recognize classical and quantum bits (qubits),
- Predict how different quantum operations transform qubits with linearity,
- program a qubit simulator that can simulate quantum random number generators.

Sharing Secrets with Quantum Key Distribution

This chapter covers:

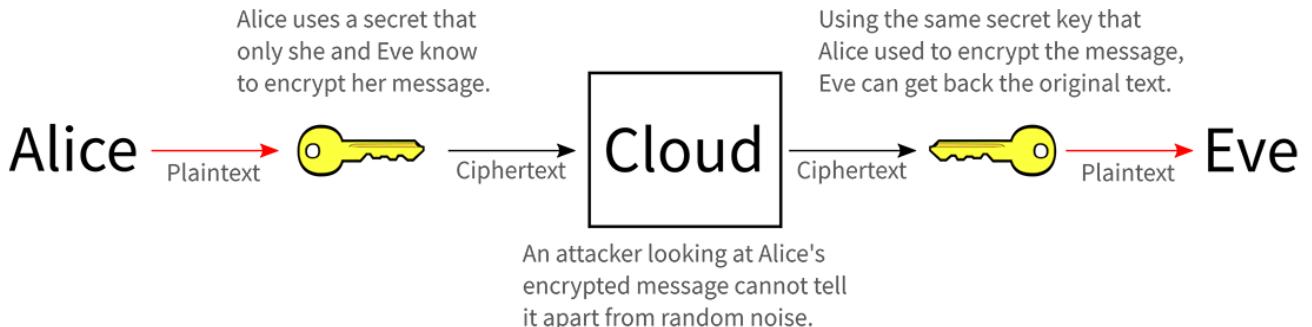
- Recognize the implications that quantum resources have for security
- Programming a simulator in Python for a quantum key distribution protocol
- Implementing the quantum NOT operation

3.1 All's Fair in Love and Encryption

In the previous Chapter, we saw that we could use random numbers as a resource to send secret messages like "♥" to our friends. We also saw that by using quantum bits or qubits, we could generate truly random numbers that are impossible for any adversary to predict.

That's only half the story, though, because we need to share those random numbers with our friends if we want to use the quantum random numbers to communicate securely with them. Those random numbers (often called a *key*) can be used with *encryption* algorithms which combine the randomness of the key with information people want to keep secret in such a way that only someone else with the *key* can see the information. We can see in [3.1](#) how two people could use a key (here random binary string) to encrypt and decrypt messages between themselves.

Figure 3.1. Mental model for how you and Eve might use encryption to communicate secretly, even over the Internet or another untrusted network.



In this Chapter, we will see that quantum technologies can help us with encryption or other cryptographic tasks, by letting us securely *distribute* our secret keys. There are classical methods for sharing random keys (e.g. RSA), but they have different guarantees about the security of the sharing. In short, using *quantum key distribution (QKD)* is *provably* secure, whereas classical key distribution methods are often *computationally* secure. This difference doesn't matter for most use cases, but if you are a government, activist group, bank, journalist, spy, or any other group where information security is a life-and-death matter, this is a huge deal.

Computational vs. Provable security

Provable security for our cryptographic protocols is the dream. A method or protocol for a cryptographic task is *provably secure* if we can write a proof showing it is secure using no assumptions about an adversary; i.e. that they can have all the time and computing power in the universe and still our protocol is secure! Most of our current cryptographic infrastructure is *computationally secure* which guarantees the security of a method or protocol with reasonable assumptions about the capabilities of an adversary. The designer or user of the protocol can choose thresholds for what finite computer resources looks like (e.g. largest current super computer or all the computers on the planet) and what a reasonable time is (e.g. 100 years, 10000 years, the age of the universe).

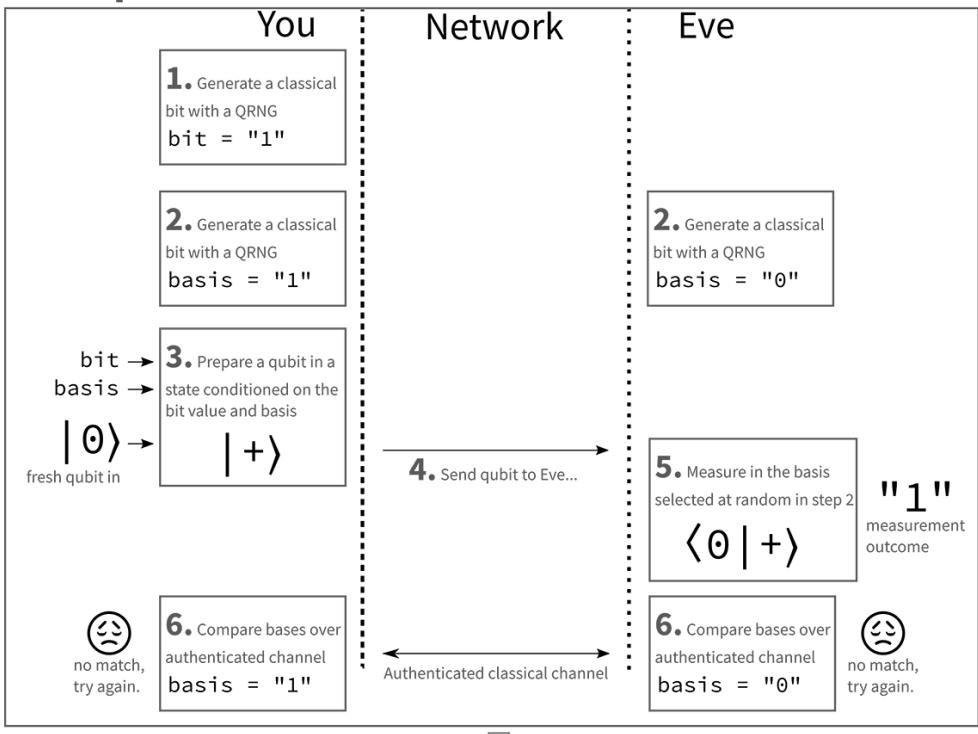
When we share a key with QKD, it does not guarantee that the key will get to the other person. This is because someone can always do a denial of service attack (e.g. cut the optical fiber between sender and receiver), which is the same for any other classical protocol. A good analogy for what QKD can promise is similar to the tamper proof seal on food products. When a peanut butter company wants to ensure that when you open the jar it is exactly as it was when it left the factory floor, they put one of these tamper proof seals on the container. The company makes a promise that if the seal makes it to you (the consumer) intact, the peanut butter will be good, and that no third party has done anything to it. Transmitting a cryptographic key with a QKD protocol is like putting a tamper proof seal on the bits in transit. If someone tries to compromise the key in transit, the receiver will know and not use that key. Sealing the bits in transit

does *not*, however, guarantee that the bits make it to the receiver.

There are many protocols that we can use to implement the general QKD scheme. In this chapter we will be working with one of the most common QKD protocols BB84, but there are many others as well that we won't have time to get in to. We will build up to this throughout this chapter, but you can see in [3.2](#) the steps to the BB84 protocol.

Figure 3.2. Timing diagram for the BB84 protocol, a particular variant of a QKD protocol.

BB84 protocol



QKD is an example of a quantum program that uses a single qubit, as well as a spin-off technology from quantum computing. What makes it attractive to develop is that we already have the hardware to implement it today! There are a number of companies that have for the last ~15 years have been commercially selling QKD hardware, however the important next steps for the technology involve hardware and software security vetting of these systems.

WARNING**Try this at home, but not with you secrets!**

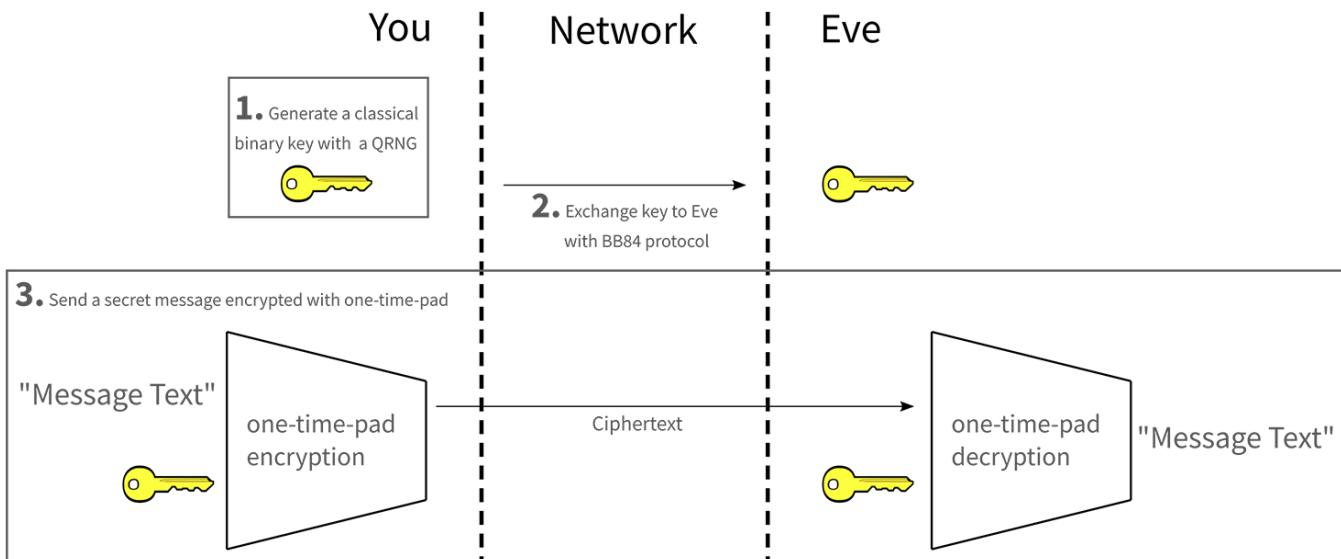
The examples we are implementing/using here in this book will *simulate* provably secure protocols. Given we are not running the examples on quantum devices, they are *not* provably secure. Even when implementing these protocols with real quantum hardware, these security proofs do nothing to stop side channel attacks or social engineering from separating you from your key 😺 We will talk more about these proof later in this chapter when we talk about the No-cloning theorem.

Let's dive into how QKD works! For our purposes here, let's say you and Eve are the two people from the previous chapter that want to exchange a key so you can send secret messages 😊 The scenario is as follows.

You wish to send a secret message to your friend. Using your quantum random number generator from Chapter 2 and the quantum key distribution protocol BB84, and one-time-pad encryption, design a program to send messages that can be provably secure.

You can visualize the scenario as a kind of timing diagram like [3.3](#).

Figure 3.3. Your scenario for this chapter: Sending a secret message to Eve with BB84 and one-time-pad encryption.



To start off, note that the key you need to send is a string of classical bits. How can we use qubits to send those classical bits? We will start by learning how to encode classical information in qubits, and then learn the specific steps of the BB84 protocol. In the next section, we will look at a new quantum operation that will help you encode classical bits with qubits.

3.1.1 Quantum NOT operations

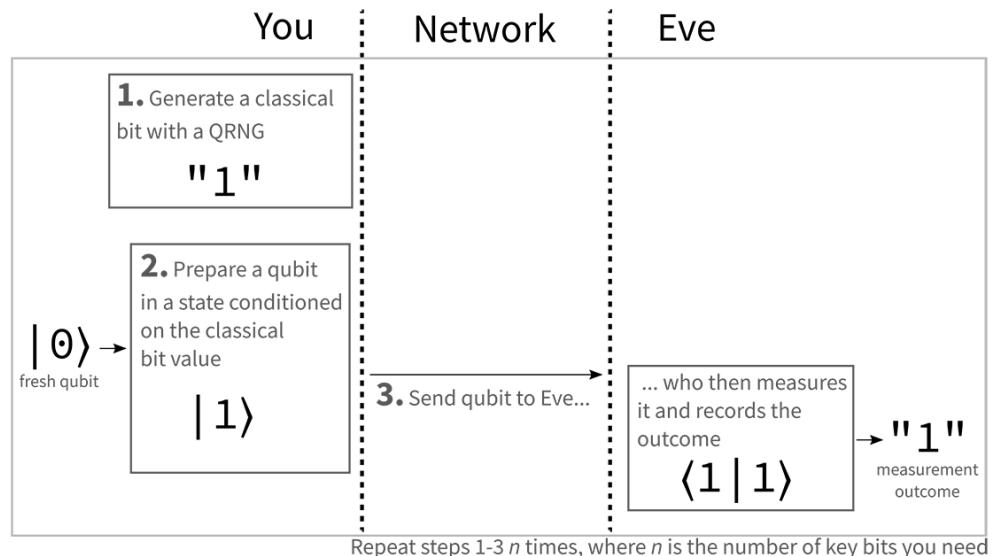
If we have some classical information, say a single binary bit, how can we encode this classical information with a quantum resource like a qubit? Take a look at the following algorithm.

Algorithm for sending a random classical bit string encoded in qubits

1. Using a quantum random number generator to generate a random key bit to send.
2. Starting with a qubit in the $|0\rangle$ state and then prepare it in a state that represent that bit value from step 1; here you use $|0\rangle$ if the classical bit was a 0, and $|1\rangle$ if the classical bit was a 1.
3. That prepared qubit is sent to Eve who then measures it and records the classical bit value they get.
4. Repeat steps 1-3 until you and Eve have as much key as you want (usually dictated by the cryptographic protocol you want to use after).

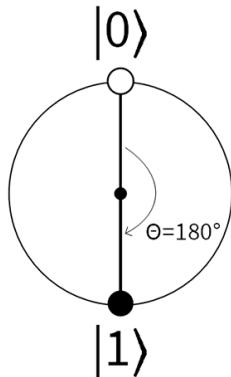
You can see a timing diagram for this algorithm in [3.4](#).

Figure 3.4. A visualization of the algorithm for sending a classical bitstring with qubits.



Now we need a way to switch the qubit from $|0\rangle$ to $|1\rangle$, so we need another quantum operation in our toolbox. In step 2, you can use a *quantum NOT* operation, which is similar to our classical NOT operation, that rotates the qubit from $|0\rangle$ to $|1\rangle$ (see [2.16](#)).

Figure 3.5. A visualization of the quantum equivalent of a NOT operation operating on a qubit in the $|0\rangle$ state, leaving the qubit in the $|1\rangle$ state.



The quantum analogue to the classical NOT gate is equivalent to a rotation of the state of the qubit of 180° .

$$\text{NOT } |0\rangle = X|0\rangle = |1\rangle$$

We refer to this quantum NOT operation as the x operation. Step 2 from above then could be re-written as follows:

1. *If your classical bit from step 1 was 0, do nothing; if it was a 1, apply a quantum NOT operation (a.k.a. x operation) to your qubit.*

This algorithm works 100% of the time because when Eve goes to measure the qubit they receive, the $|0\rangle$ and $|1\rangle$ states can be perfectly distinguished with a measurement in the Z -axis. This kind of seems like a lot of work for you and Eve to just share some random classical bits, but we will see how adding some quantum behaviors to this basic protocol will make it more useful! Let's look at how we could implement this in code:

Listing 3.1. You and Eve exchange classical bits via qubits, encoding the message with the $|0\rangle$ and $|1\rangle$ states.

```
def prepare_classical_message(bit : bool, q : Qubit) -> None:           ①
    if bit:
        q.x()

def eve_measure(q : Qubit) -> bool:
    return q.measure()                                              ③

def send_classical_bit(device : QuantumDevice, bit : bool) -> None:
    with device.using_qubit() as q:
        prepare_classical_message(bit, q)
        result = eve_measure(q)
        q.reset()
    assert result == bit                                              ④
```

- 1 To prepare our qubit with the classical bit we want to send, we need as input the bit value and a qubit to use. This function does not return anything because the consequences of the operations we apply to our qubit are tracked in the single qubit simulator itself.

- ② If you're sending a 1, you can use the NOT operation x to prepare q in the $|1\rangle$ state because the x operation will rotate $|0\rangle$ to $|1\rangle$ and vice versa.
- ③ This seems silly to separate measuring as another function given it's one line, but we will change up how Eve will measure the qubit in the future so this is a helpful setup.
- ④ We can check that measuring q gives the same classical bit as you sent.

The simulator you wrote in the previous chapter *almost* has what you need to implement this. You just need to add an instruction corresponding to the x operation. The x instruction can be represented with a matrix X , just as we represented the h instruction using the matrix H . In the same way as we wrote down H in Chapter 2, we can write down the matrix X as

Equation 3.1

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Exercise 3.1: truth tables and matrices

In Chapter 2, we saw that unitary matrices play the same role in quantum computing that *truth tables* play in classical computing. We can use that to figure out what the matrix X has to look like in order to represent the quantum NOT operation, x . Let's start by making a table of what the matrix X has to do to each input state in order to represent what the x instruction does:

Input	Output
$ 0\rangle$	$ 1\rangle$
$ 1\rangle$	$ 0\rangle$

This table tells us that if we multiply the matrix X by the vector $|0\rangle$, we need to get $|1\rangle$, and similarly that $X|1\rangle = |0\rangle$.

Either by using NumPy or by hand, check that the matrix

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

matches what we have in our truth table above.

Let's go on and add the functionality we need to our simulator in order to run the above snippet. We will be working with the simulator you wrote in the previous chapter, but if you need a refresher, you can find the code in the GitHub repo for this book: github.com/crazy4pi314/learn-qc-with-python-and-qsharp First, we need to update the interface for our quantum device, by adding a new method our qubit must have.

Listing 3.2. Adding the x operation to the interface specification for a qubit.

```
class Qubit(metaclass=ABCMeta):
    @abstractmethod
    def h(self): pass

    @abstractmethod
    def x(self): pass ①

    @abstractmethod
    def measure(self) -> bool: pass

    @abstractmethod
    def reset(self): pass
```

- ① We can model implementing the quantum NOT operation after the `h` operation from Chapter 1.

Now that our interface for a qubit knows that we want an implementation of the `x` operation, let's add that implementation!

Listing 3.3. Adding the x operation to the implementation of a qubit simulator.

```
KET_0 = np.array([
    [1],
    [0]
], dtype=complex)
H = np.array([
    [1, 1],
    [1, -1]
], dtype=complex) / np.sqrt(2)
X = np.array([
    [0, 1],
    [1, 0]
], dtype=complex) / np.sqrt(2) ①

class SimulatedQubit(Qubit):
    def __init__(self):
        self.reset()

    def h(self):
        self.state = H @ self.state

    def x(self): ②
        self.state = X @ self.state

    def measure(self) -> bool:
        pr0 = np.abs(self.state[0, 0]) ** 2
        sample = np.random.random() <= pr0
        return bool(0 if sample else 1)

    def reset(self):
        self.state = KET_0.copy()
```

- ① Let's add the a variable `X` to store the matrix X that we need to represent the `x` operation.

- ② Just like the `h` function defined above, we want to implement the quantum operation `x` by just applying the matrix stored in `X` to the state vector.

3.1.2 Sharing classical bits with qubits

Awesome! Now let's try out using our upgraded Python qubit simulator to share a secret classical bit with a qubit. This is not quite the same as a quantum key distribution protocol yet, but it serves as a good foundation for the types of function and steps that our end goal QKD protocol has.

Open an IPython session where you are keeping the code for your simulator by running `ipython` in your terminal. After importing the Python files, create an instance of the single qubit simulator and generate a random bit to use as the classical bit we want to send. (Good thing we have a quantum random number generator!) Then using a fresh qubit, prepare it based on the classical bit value you want to send Eve. Eve then measures the qubit, and we can see if you both have the same classical bit value!

Listing 3.4. Using our single qubit simulator to send classical bits with qubits

```
>>> qrng_simulator = SingleQubitSimulator()                                     1
>>> key_bit = int(qrng(qrng_simulator))                                       2
>>> qkd_simulator = SingleQubitsSimulator()                                    3
>>> with qkd_simulator.using_qubit() as q:
...     prepare_message(key_bit, q)                                              4
...     print(f"You prepared the classical key bit: {key_bit}")
...     eve_measurement = int(eve_measure(q))
...     print(f"Eve measured the classical key bit: {eve_measurement}")          5
...
You prepared the classical key bit: 1
Eve measured the classical key bit: 1
```

- ① We need a simulated qubit to use for our quantum random number generator.
- ② Re-using the `qrng` function that we wrote in the previous chapter, we can generate a random classical bit to use for our key.
- ③ We are using a new qubit simulator instance here for the key exchange, *strictly speaking* we don't need to. You will see in the next chapter how to expand the simulator to work with multiple qubits.
- ④ You encode your classical bit in the qubit provided by `qkd_simulator`. If the classical bit was 0, you do nothing to `qkd_simulator`, and if the classical bit was 1 then you use the `x` method to change the qubit to the $|1\rangle$ state.
- ⑤ Eve measures the qubit from `qkd_simulator` and then stores the bit value as `eve_measurement`.

Our example of secret sharing with qubits above should be deterministic, which is to say that every time you prepare and send a bit, Eve will correctly measure the same value. Is this secure? If you suspect it is not secure, you are definitely on to something :) In the next section we will discuss the security of our prototype secret sharing scheme, and look at ways you can improve it!

3.2 A tale of two bases

You and Eve now have a way of sending classical bits using qubits, but what happens if an adversary gets a hold of that qubit? They could just use the `measure` instruction to get the same classical data that Eve does. That's a huge problem and would reasonably make you wonder why one would use qubits to share keys in the first place.

Thankfully, quantum mechanics offers us a way to make this exchange more secure! What are some modifications we could make to our protocol above? We could have, for instance, decided to represent a classical "0" message with a qubit in the $|+\rangle$ state and a "1" message with a qubit in the $|-\rangle$ state.

Listing 3.5. You and Eve exchange classical bits via qubits, but encode the message with the $|+\rangle$ / $|-\rangle$ states.

```
def prepare_classical_message_plusminus(bit : bool, q : Qubit) -> None:
    if bit:
        q.x()
    q.h()                                ①

def eve_measure_plusminus(q : Qubit) -> bool:
    q.h()                                ②
    return q.measure()

def send_classical_bit_plusminus(device : QuantumDevice, bit : bool) -> None:
    with device.using_qubit() as q:
        prepare_classical_message_plusminus(bit, q)
        result = eve_measure_plusminus(q)
        assert result == bit
```

- ① Everything up to this line of `prepare_classical_message_plusminus` was the same as we were using before with `prepare_classical_message`. Applying the Hadamard gate at this point rotates the $|0\rangle$ / $|1\rangle$ states to $|+\rangle$ / $|-\rangle$ states.
- ② Here we use the `h` operation to rotate our $|+\rangle$ / $|-\rangle$ states back to the $|0\rangle$ / $|1\rangle$ states because our `measure` operation is defined to only measure the $|0\rangle$ / $|1\rangle$ states correctly. Another way of thinking about this is that you are rotating the measurement to match the rotation we are currently working in ($|+\rangle$ / $|-\rangle$). It's all a matter of perspective!

Now you have two different ways of sending qubits that you and Eve could use when sending qubits (see 3.1 for a summary). These two different ways of sending messages we call *bases*, which each contain two completely distinguishable (orthogonal) states. This is similar to the previous chapter where we looked at map directions (ex. North and West) which defined a convenient *basis* for describing directions. Here, we have used the $|0\rangle$ and $|1\rangle$ states as one basis (called the *Z* basis), and $|+\rangle$ and $|-\rangle$ as another (called the *X* basis). The names for these bases refer to the axis along which you can perfectly distinguish the states (see 3.6).

Figure 3.6. Now in addition to using the Z -basis to encode a classical bit on a qubit, we can use the X -basis.

If the classical bit is encoded in the X basis, we can use $|-\rangle$ and $|+\rangle$ instead of $|0\rangle$ and $|1\rangle$. Since $|+\rangle$ and $|-\rangle$ are opposite each other (this time on the X -axis), they are still perfectly distinguishable from each other.

If the classical bit is encoded in the Z basis, we can use $|0\rangle$ and $|1\rangle$, as these two states are perfectly distinguishable from each other (that is, they are opposite each other on the Z -axis).

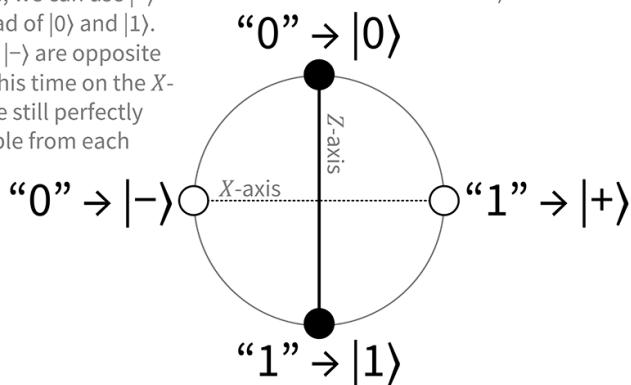


Table 3.1. Different classical messages we want to send, and how to encode them in the Z and X bases.

	"0" message	"1" message
"0" (or Z) basis	$ 0\rangle$	$ 1\rangle = X 0\rangle$
"1" (or X) basis	$ +\rangle = H 0\rangle$	$ -\rangle = H 1\rangle = HX 0\rangle$

NOTE

In quantum computing there is never really a *correct* basis, so much as there are convenient bases that we choose to use by convention.

If you both don't know which way of sending you are using for a particular bit, you both have a problem. What happens if we mix sending our messages in Z -basis and X -basis? Good news, you can use your simulator to try it out and see what happens.

Listing 3.6. You and Eve exchange classical bits via qubits, but are not using the same basis.

```
def prepare_classical_message(bit : bool, q : Qubit) -> None:
    if bit:
        q.x()

def eve_measure_plusminus(q : Qubit) -> bool:
    q.h()
    return q.measure()
```

①

②

```
def send_classical_bit_wrong_basis(device : QuantumDevice, bit : bool) -> None:
    with device.using_qubit() as q:
        prepare_classical_message(bit, q)
        result = eve_measure_plusminus(q)
        assert result == bit, "Two parties do not have the same bit value" ❸
```

- ❶ Here you will be using the method we saw before to prepare your qubit in the Z -basis basis by using the `h` method.
- ❷ Eve measures in the X -basis because she does a Hadamard gate on her qubit before measuring.
- ❸ The function does not return anything, so if you and Eve end up with key bits that don't match, it will raise an error.

Listing 3.7. Sending qubits in the Z -basis, and measuring in the X -basis

```
>>> qsim = SingleQubitSimulator()
>>> send_classical_bit_wrong_basis(qsim, 0) ❶
AssertionError: Two parties do not have the same bit value
```

- ❶ We will just pick our bit value is 0, and you may have to run this line a few times before you get the above error.

You can try this out experimentally, and you will find that you get the above `AssertionError` (the key exchange failed) about half of the time. Why is that? To start with, Eve is measuring in the X -basis, so she can only tell $|+\rangle$ and $|-\rangle$ apart perfectly. What will she measure if she is not given a perfectly distinguishable state for her basis (like here where she is given a $|0\rangle$)? We can write the $|0\rangle$ state in the X -basis as

Equation 3.2

$$|0\rangle = (|+\rangle + |-\rangle) / \sqrt{2}.$$

Recall that in Chapter 2 we defined $|+\rangle$ in a similar way by adding $|0\rangle$ and $|1\rangle$ together. The $|+\rangle$ state was also called a *superposition* of the $|0\rangle$ and $|1\rangle$ states.

NOTE

Any time a state can be written as a linear combination of states like this, it is considered to be a superposition of the states that are added together.

Exercise 3.2: verify that $|0\rangle$ is a superposition of $|+\rangle$ and $|-\rangle$.

Try using what we learned about vectors in the previous chapter to verify that $|0\rangle = (|+\rangle + |-\rangle) / \sqrt{2}$, either by hand or using Python.

Hint: recall that $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ and that $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$.

Now to calculate the actual measurement with Born's rule from Chapter 2. Recall we

can calculate the probability of a measurement outcome by measuring a particular state with the expression

Equation 3.3

$$\text{Pr(measurement | state)} = |\langle \text{measurement} | \text{state} \rangle|^2$$

Writing out the measurement of the $|0\rangle$ state in the X basis you can see that we will get 0 (or $|+\rangle$) half of the time and 1 (or $|-\rangle$) the other half.

Equation 3.4

$$\text{Pr}(|+\rangle|0\rangle) = |(+|0\rangle)|^2 + ((|+|+> + |+|->)/\sqrt{2})^2 = (1+0)^2/2 = \frac{1}{2}$$

Exercise 3.3: measuring qubits in different bases

Using the example above as a guide,

- calculate what the probability of getting the measurement outcome $|-\rangle$ when measuring the $|0\rangle$ state in the $|-\rangle$ direction.
- Also calculate what the probability of getting the $|-\rangle$ measurement outcome with the input state of $|1\rangle$.

That tells us that if Eve does not know the right basis to measure in, then the measurements she makes are as good as randomly guessing. This is because in the wrong basis, the qubit is in a superposition of the two states that define the basis. One 'key' to how QKD works is that without the right additional information (the basis the qubit is encoded in) any measurement of the qubit is basically useless. Now to ensure our security, we have to make it difficult for an adversary to learn that extra information to know the right basis to measure in. The QKD protocol we will look at next has a solution for this, and a proof (out of scope here) that describes the chance that the attacker has any information about the key!

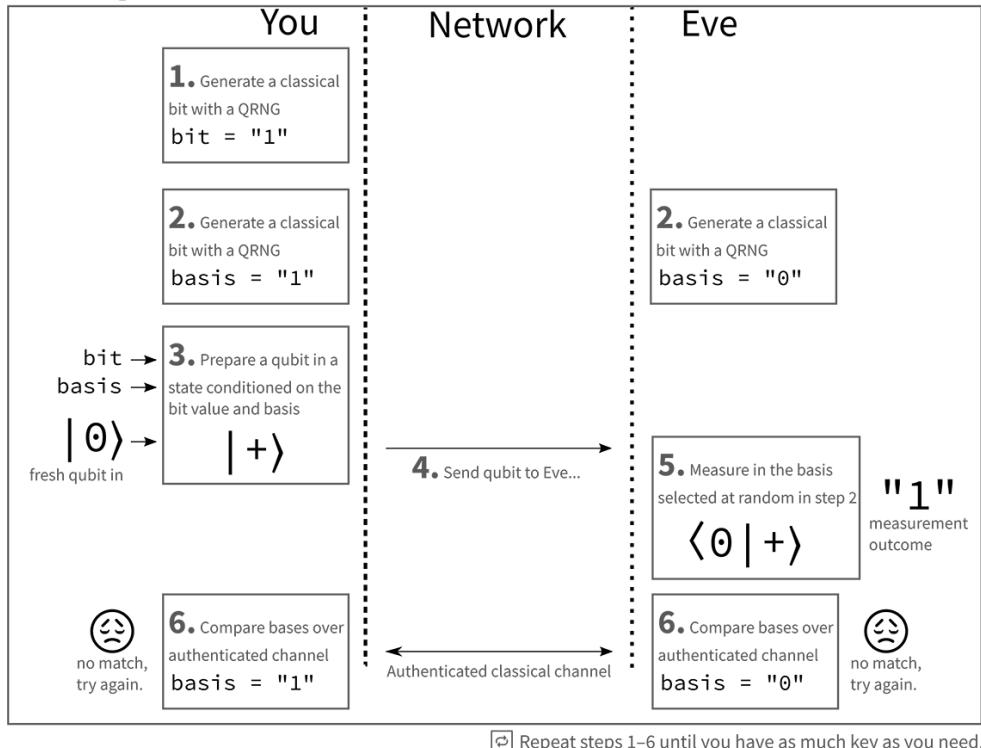
3.3 Quantum Key Distribution: BB84

We have now seen how to share keys in two different bases, and what happens if you and Eve don't use the same basis. You might again ask why we use this to make sharing the key to our secret keys more secure? There are a wide variety of different QKD protocols, each with specific advantages and use cases (not unlike RPG character classes). The most *common* protocol for QKD is called BB84, named for an appropriately cryptic encoding of the two authors initials and the year it was published (Bennet and Brassard 1984).

BB84 is very similar to what we have worked out so far to share keys, but has one important difference in how you and Eve choose your bases. In BB84, both parties choose their basis randomly (and independently), which means they will end up using the same basis 50% of the time. You can see a figure with the steps of the BB84 protocol in 3.7.

Figure 3.7. The steps in the BB84 protocol, a particular version of a QKD protocol.

BB84 protocol



As a consequence of randomly choosing bases, you and Eve also have to do some communication over authenticated* classical channels (like the internet) to be able to take the keys you each have and transform them into a key that you believe is identical to the key their partner has. This is because this is real life, and when the qubits are exchanged, it will be possible for both the environment and third party individuals to manipulate or modify the state of the qubit.

Key expansion

There is one detail we glossed over in our description of the classical communication channel that you and Eve have, namely in that it must be *authenticated*. That is, when you send classical messages to Eve as a part of running BB84, it's OK if someone else can read them, but you need to make sure it's really

Eve that you're talking to. To prove that someone wrote and sent a particular message you actually **already** have to have some form of shared secret that you can use to validate someone else's identity. So we have to already have some shared secret with the other person in BB84. This secret can be smaller than the message you're trying to send, so BB84 is technically more of a **key expansion** protocol.

Table 3.2. What state you should send for each random message and basis choice

	"0" message	"1" message
"0" (or Z) basis	$ 0\rangle$	$ 1\rangle = X 0\rangle$
"1" (or X) basis	$ +\rangle = H 0\rangle$	$ -\rangle = H 1\rangle = HX 0\rangle$

3.3.1 Steps of the BB84 protocol:

1. You choose a random one-bit message to send by sampling your QRNG
2. You and Eve each choose a random basis with your respective QRNGs (no communication between them)
3. You prepare a qubit in the randomly selected basis, representing your randomly selected message (see 3.2).
4. You send your prepared qubit in the quantum channel to Eve.
5. Eve measures the qubit when it arrives, performing the measurement in her randomly selected basis, and recording the classical bit outcome.
6. Communicate on an authenticated classical channel with Eve and share which bases you used for preparing and measuring the qubit. If they match, keep the bit and add it to the key.
7. Repeat steps 1–6 until you have as much key as you need.

An error free world

Since we are simulating the BB84 protocol, we know that the qubit Eve will receive is exactly the same as what you sent. BB84 more realistically will be done in batches where n qubits are exchanged first, and then a round of sharing the basis values (error correction happen). You also have to at the end shrink the key even further with privacy amplification algorithms to account for the fact that an eavesdropper could have gotten partial information from the errors you detected. We omitted these steps in our implementation of BB84 to keep things simple, however the steps are critical for real-world security 😊

Let's jump in and implement the BB84 QKD protocol in Python! We will start by writing a function that will run the BB84 protocol (assuming lossless transmission) for a single bit transmission. That does not guarantee that we get one key bit from this run however, if you and Eve choose different bases then that exchange will have to be thrown out.

First, it is helpful to setup some functions that will help us simplify how we write out the full BB84 protocol. You and Eve need to do things like sample random bits and prepare and measure the message qubit, which are separated here for clarity.

Listing 3.8. Some helper functions before we get to the full BB84 key exchange.

```

def sample_random_bit(device : QuantumDevice) -> bool:
    with device.using_qubit() as q:
        q.h()
        result = q.measure()
        q.reset()
    return result

def prepare_message_qubit(message : bool, basis : bool, q : Qubit) -> None: ②
    if message:
        q.x()
    if basis:
        q.h()

def measure_message_qubit(basis : bool, q : Qubit) -> bool:
    if basis:
        q.h()
    result = q.measure()
    q.reset()
    return result

def convert_to_hex(bits : List[bool]) -> str: ④
    return hex(int(
        "".join(["1" if bit else "0" for bit in bits]),
        2
    ))

```

- ① `sample_random_bit` is *almost* the same as our `qrng` function before, except here we will reset the qubit after measuring as we know we want to be able to use it more than once.
- ② Here the qubit is encoded with the key bit value in the randomly selected basis.
- ③ Similar to `sample_random_bit` after Eve measures the message qubit, she should reset it because in the simulator we will be reusing it for the next exchange.
- ④ To help condense the display of long binary keys, a helper function is used to convert the representation to a shorter hex string.

Listing 3.9. BB84 protocol for sending a single classical bit between two parties.

```

def send_single_bit_with_bb84(your_device : QuantumDevice, eve_device : QuantumDevice) -> tuple:

    [your_message, your_basis] = [
        sample_random_bit(your_device) for _ in range(2) ①
    ]

    eve_basis = sample_random_bit(eve_device) ②

    with your_device.using_qubit() as q:
        prepare_message_qubit(your_message, your_basis, q) ③

    # QUBIT SENDING... ④

    eve_result = measure_message_qubit(eve_basis, q) ⑤

    return ((your_message, your_basis), (eve_result, eve_basis)) ⑥

```

- ➊ Here you can randomly choose a bit value and basis using your modified qrng from before, here the `sample_random_bit` function.
- ➋ Eve needs to randomly choose a basis with their own qubit, which is why she is using a separate `QuantumDevice`.
- ➌ With all the preparation done, you then need to prepare your qubit to send to Eve.
- ➍ Since all of our computation happens inside a simulator on your computer, nothing needs to be done to "send" the qubit from you to Eve. In real life however, this is where all the bad stuff happens: errors, loss, even attempted eavesdropping.
- ➎ Now Eve has your qubit, and measures it in the randomly selected basis she chose earlier.
- ➏ This function returns the key bit values and bases you and Eve would have at the end of this one round.

Qubits and No-cloning

From what we've seen so far, it seems like our adversary could cheat by eavesdropping on the qubits in the quantum channel and making copies. Here's the gig: The eavesdropper (called Bob here) first would need to (without detection):

Steps for Bob to eavesdrop on you and Eve

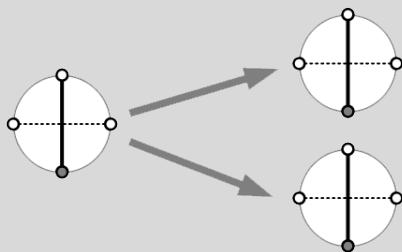
- 1) Copy qubits as they are being sent between you and Eve, and then store them.
- 2) Next, while you and Eve finish the classical part of the protocol, Bob listens to the bases they announce and keep track of ones you both chose the same.
- 3) For the qubits corresponding to bits where you and Eve used the same basis, Bob would measure their copies of the qubits in the same basis as well.

Ta-dal! You, and Eve *and* Bob would all have the same key! If this seems like a problem, you are right. Don't worry though, quantum mechanics has the solution. It turns out the problem with Bob's plan is in step 1, where they need to make *identical* copies of the qubits that you and Eve are exchanging. The good news is that making an exact copy of a qubit without knowing what it was beforehand is forbidden by quantum mechanics. The rule that qubits cannot be identically copied without prior knowledge of the state is called the *No-cloning theorem* and is stated as follows.

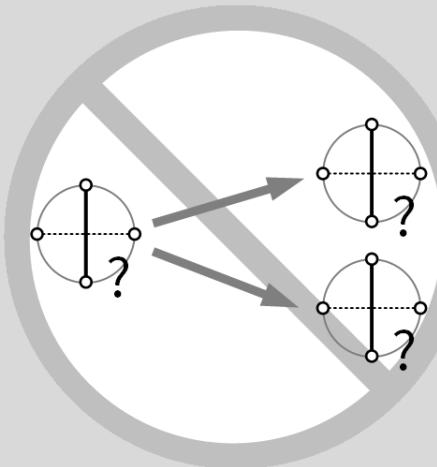
No quantum operation can perfectly copy the state of an arbitrary qubit onto another qubit.

Figure 3.8. The No-cloning theorem visualized.

If we have some classical description of the state of the qubit, then we can make copies of it. Here we have a qubit in the $|1\rangle$ state and we can make “copies” of it by preparing more by taking a $|0\rangle$ and applying the x operation.



If we have no prior information about a qubit, then we **cannot** make perfect copies of it.



You will be able to do the simple proof of this in the next chapter, once we learn how to describe the state of more than one qubit 😊.

Alternately, if Bob could measure a qubit without disturbing it, he could get around needing a copy of the qubits he intercepts. This is not possible because once you measure a qubit, it "collapses" or changes in a way that can be detectable to Eve. So measuring in transit is not something Bob can do without being detected, so his eavesdropping would fail.

Now, exchanging one classical key bit is not going to be sufficient to send a whole key so now we need to use the above technique to send multiple bits.

Listing 3.10. BB84 protocol for exchanging key with Eve until a specified amount of key is reached.

```
def simulate_bb84(n_bits : int) -> tuple:
    your_device = SingleQubitSimulator()
    eve_device = SingleQubitSimulator()

    key = []
    n_rounds = 0

    while len(key) < n_bits:
        n_rounds += 1
        ((your_message, your_basis), (eve_result, eve_basis)) = \
            send_single_bit_with_bb84(your_device, eve_device)
```

```

if your_basis == eve_basis:
    assert your_message == eve_result
    key.append(your_message)

print(f"Took {n_rounds} rounds to generate a {n_bits}-bit key.")

return key

```

1

- ① At this point, you and Eve can publicly announce the bases you each used to measure this bit. If everything worked right, your results should agree whenever your bases agree. We'll check that here with an assert.

The key is now in the bag, so we can move on to using the key and the one-time-pad encryption algorithm to send a secret message!

3.4 Using our secret key to send secret messages

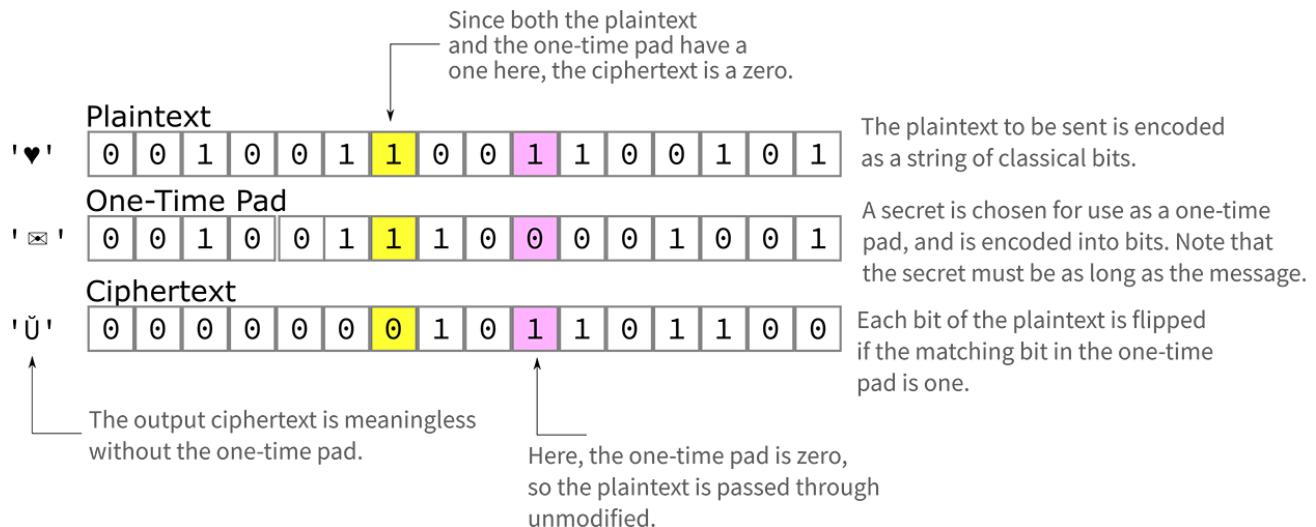
You and Eve have now sorted out how to use the BB84 protocol to share a random, binary key generated by a QRNG. The last step now is to use this key to share a secret message with Eve! You and Eve had previously decided that the best encryption protocol to use is a one-pad to send your secret messages. This turns out to be one of the most secure encryption protocols, and given you are sharing keys in one of the most secure ways possible it makes sense to keep up that standard!

Let's say that you were trying to send Eve that you like Python, so the message you want to send is "❤️🐍". Since we are using a binary key, we need to convert the representation of our unicode message to binary, which is the following lengthy list of bits.

```
"110110000011101 1101110010010110 110110000011101
1101110000001101 110110000011101 110111001011101"
```

This binary representation of our message is our *message text* and now we want to combine that with a key to get a ciphertext that is safe to send over the network. Once you have the key from the BB84 protocol (at least as long as your message) next you need to use a one-time-pad encryption scheme to then encode your message. We saw this encryption technique already in Chapter 2, see [2.24](#) for a quick refresher.

Figure 3.9. An example of one-time-pad encryption which uses random bits to encrypt secret messages.



To implement this, you will need to use a classical bitwise XOR (the `^` operator in Python) to combine the message and your key to create the ciphertext that you can safely then send to Eve. To decrypt your message Eve will do the same bitwise XOR operation with the ciphertext and her key (which should be the same as yours). This will give her back the message because any time you XOR a bitstring with another one twice, you will be left with the original bitstring. Here is what this would look like in Python.

Listing 3.11. BB84 protocol for exchanging key with Eve until a specified amount of key is reached.

```
def apply_one_time_pad(message : List[bool], key : List[bool]) -> List[bool]:
    return [
        message_bit ^ key_bit
        for (message_bit, key_bit) in zip(message, key)
    ]
```

- ➊ The `^` operator is a bitwise XOR in Python. This applies a single bit of our key as a one-time-pad to our message text.

Exercise 3.4: one-time-pad encryption

If you had the ciphertext `10100101` and the key `00100110`, what was the message that was originally sent?

Let's put it all together and share our message ("♥✉") to Eve by running

the bb84.py file we have been building up!

Listing 3.12. Sharing a secret message with Eve with BB84 and one-time-pad encryption.

```
if __name__ == "__main__":
    print("Generating a 96-bit key by simulating BB84...")
    key = simulate_bb84(96)
    print(f"Got key {convert_to_hex(key)}.")

    message = [
        1, 1, 0, 1, 1, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 1, 0, 0,
        1, 0, 0, 1, 0, 1, 1, 0,
        1, 1, 0, 1, 1, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 1, 0, 0,
        0, 0, 0, 1, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 1, 0, 0,
        1, 0, 1, 1, 1, 0, 1, 1
    ]
    print(f"Using key to send secret message: {convert_to_hex(message)}.")

    encrypted_message = apply_one_time_pad(message, key)
    print(f"Encrypted message: {convert_to_hex(encrypted_message)}.")

    decrypted_message = apply_one_time_pad(encrypted_message, key)
    print(f"Eve decrypted to get: {convert_to_hex(decrypted_message)}")
```

Listing 3.13. Running the full solution to the scenario for the chapter.

```
$ python bb84.py
Generating a 96-bit key by simulating BB84...
Took 170 rounds to generate a 96-bit key.
Got key: 0xb35e061b873f799c61ad8fad. ①
Using key to send secret message: 0xd83ddc96d83ddc0dd83ddcbb. ②
Encrypted message: 0x6b63da8d5f02a591b9905316. ③
Eve decrypted to get: 0xd83ddc96d83ddc0dd83ddcbb. ④
```

- ➊ Since your and Eve's bases will agree roughly half of the time, it should take about two rounds of BB84 for each bit of key you want to generate.
- ➋ The exact key you generate will be different every time you run the BB84 simulation — that's a huge part of the point of the protocol, after all!
- ➌ Our message here is what we get by writing down each of the Unicode code points for "推崇". When we combine our secret message with the key we got above, using the key as a one-time-pad, our message is completely scrambled.
- ➍ When Eve uses the same key, she gets our original secret message back!

3.5 Summary

In this chapter you learned:

- Recognize the implications that quantum resources have for security
- Implement the quantum NOT operation
- Program a simulator in Python for a quantum key distribution protocol

Quantum key distribution is one of the most important spin-off technologies from quantum computing and has huge potential impact for our security infrastructure. While it is currently quite easy to setup QKD for parties that are relatively close to each other (< 200 km), there are significant challenges to deploying a global system for QKD. Usually the physical system used in QKD is a photon and it is hard to send single particles of light long distances without losing them.

Now that you have built up a single qubit simulator and programmed some single qubit applications, you are now ready to start playing around with multiple qubits! In the next chapter we will take the simulator you have built and add features to allow it to simulate multiple qubit states and use it to play nonlocal games with Eve. ❤️

Nonlocal Games: Working with multiple qubits

This chapter covers:

- Simulate state preparation, operations, and measurement results for multiple qubits,
- Program a simulator for multiple qubits leveraging the QuTiP Python package and tensor products,
- Recognize the proof quantum mechanics is consistent with our observations of the universe by simulating experimental results.

4.1 Nonlocal Games

At this point, we have seen how single-qubit devices can be programmed to accomplish useful tasks such as random number generation and quantum key distribution. The most exciting computational tasks, however, require using multiple qubits together. In this chapter, you will learn about nonlocal games: a way to validate our quantum mechanical descriptions of the universe with friends with multi-qubit systems.

We will dive into a new Python package called QuTiP that will allow us to program quantum systems faster and has some cool built-in features for simulating quantum mechanics. Then you will learn how to leverage QuTiP and program a simulator for *multiple* qubits and see how that changes (or doesn't!) the 3 main tasks for our qubits: state preparations, operations, and measurement.

4.1.1 What are nonlocal games?

We have all played games of one type or another, whether sports, board games, video

games or roleplaying games. Games are one of the best ways we have to explore new worlds, to test our limits of strength, endurance and understanding. Turns out Eve loves to play games, and the latest encrypted message from her was the following text.

"Hi player! I am keen to play a game called CHSH. It is a *nonlocal* game were we play with a referee. I'll send the instructions in the next message. STOP"

What makes the game proposed by Eve *nonlocal* is the fact that the players are (sadly) not in the same place while playing the game. The players participate in the game by sending and receiving messages with a central referee but don't have a chance to talk to each other while playing the game. What is really cool about it is that by playing it we can show that classical physics just doesn't cut it to describe the results we get in these games with particular strategies. The particular winning strategy we will look at here involves the players sharing a pair of qubits before the game starts. We will dive into just what entangling two qubits means as we go through this chapter, but let's just start with describing the full rules of our nonlocal game.

NOTE

A referee adjudicating a nonlocal game can ensure that they players don't communicate by separating the players by a large enough distance that no light from one player could reach the other before the game ends.

4.1.2 Testing quantum physics itself: The CHSH game

The nonlocal game the Eve has suggested playing is called *the CHSH game*.¹

You get the next message from Eve and it contains the following information, as well the figure shown in 4.1.

¹ The name "CHSH" comes from the initials of the researchers who originally created the game, Clauser, Horne, Shimony, and Holt.

Note: The original paper can be found here if you are interested:
journals.aps.org/prl/abstract/10.1103/PhysRevLett.23.880.

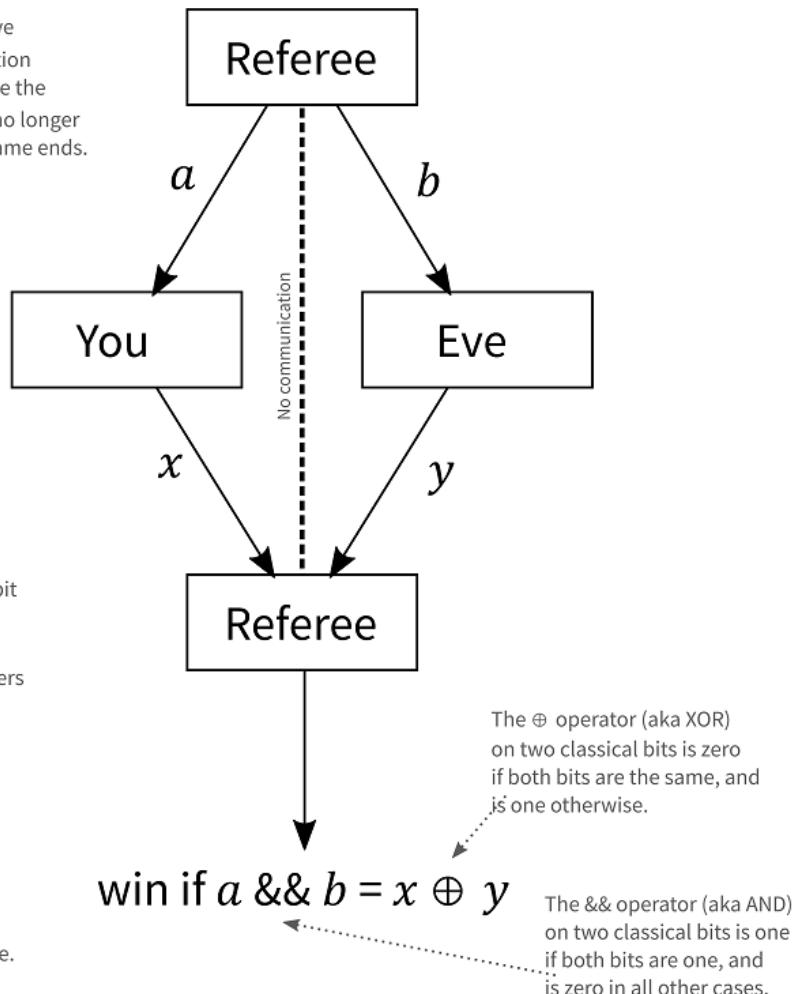
Figure 4.1. The CHSH game, a nonlocal game with two players and a referee.

1. A referee sends you and Eve a one-bit question. Your question is labeled a and Eve's is b . Once the game starts, you and Eve can no longer communicate until after the game ends.

2. The referee demands a one-bit answer from each player, here your answer is labeled x and Eve's is labeled y .

3. Both players send a one-bit answer back to the referee. The referee then scores the game and declares if the players won.

4. The scoring rules require that you and Eve answer with the correct parity given the input bits from the Referee.



The CHSH game is comprised of two players, and a referee. You can play as many rounds of the game as you like, and each round has 3 steps. As Eve mentioned in her first message, once a round starts, the players cannot communicate and make their own (possibly pre-planned) decisions.

Steps for one round of the CHSH game

1. The referee starts the round by giving you and Eve each a one classical bit. The referee chooses these bits *independently* and at *uniformly random*, so you could get a 0 or 1 each with 50% probability, and same for Eve. This means that there are four possible ways that the referee can start the game (your bit, Eve's bit): (0,0), (0,1), (1,0), or (1,1).
2. You and Eve must each **independently** decide on a single classical bit to give

back to the referee as a response.

3. The referee then calculates the parity (XOR) of your and Eve's classical bit responses. As listed in 4.1, in three out of the four cases, you and Eve must respond with *even* parity (your answers must be equal) in order to win, while in the fourth case, your answers must be different. These are definitely *unusual* rules, but not too bad compared to some multi-day board games.

Table 4.1. Win conditions for the CHSH game.

Your Input	Eve's Input	Response parity to win
0	0	Even
0	1	Even
1	0	Even
1	1	Odd

We can expand on 4.1 to get all of the possible outcomes of the game, see Table 4.2

Table 4.2. All possible states of the CHSH game with win conditions. Input bits come from the referee, and both of you respond to the referee as well.

Your Input	Eve's Input	Your Response	Eve's Response	Parity	Win?
0	0	0	0	Even	Yes
0	0	0	1	Odd	No
0	0	1	0	Odd	No
0	0	1	1	Even	Yes
0	1	0	0	Even	Yes
0	1	0	1	Odd	No
0	1	1	0	Odd	No
0	1	1	1	Even	Yes
1	0	0	0	Even	Yes
1	0	0	1	Odd	No
1	0	1	0	Odd	No
1	0	1	1	Even	Yes
1	1	0	0	Even	No
1	1	0	1	Odd	Yes
1	1	1	0	Odd	Yes
1	1	1	1	Even	No

Let's look at some Python code to simulate this game. Since your and Eve's responses to the referee are allowed to depend on the message that the referee gives you, you can represent each player's actions as a "function" that the referee calls.

Exercise 4.1: Umpire state of mind

Since the referee is purely classical, we'll model them as using classical random number generators. This leaves open the possibility, though, that you and Eve could cheat by guessing the referee's questions. A

possible improvement might be to use the QRNGs from Chapter 2. Modify the code sample in 4.1 so that the referee can ask questions of you and Eve by measuring a qubit that starts off in the $|+\rangle$ state.

Listing 4.1. Python implementation for the CHSH game

```

import random
from functools import partial
from typing import Tuple, Callable
import numpy as np

from interface import QuantumDevice, Qubit
from simulator import Simulator

Strategy = Tuple[Callable[[int], int], Callable[[int], int]]    ①

def random_bit() -> int:
    return random.randint(0, 1)                                     ②

def referee(strategy : Callable[[], Strategy]) -> bool:
    you, eve = strategy()
    your_input, eve_input = random_bit(), random_bit()           ③
    parity = 0 if you(your_input) == eve(eve_input) else 1          ④
    return parity == (your_input and eve_input)                     ⑤

def est_win_probability(strategy : Strategy,
                       n_games : int = 1000) -> float:            ⑥
    return sum(
        referee(strategy)
        for idx_game in range(n_games)
    ) / n_games                                         ⑦

```

- ➊ Here we are declaring a new type `Strategy` to define the tuple of functions that represent your and Eve's one bit functions that represent your individual strategies. Using Python's `typing` module lets us document that a value of type `Strategy` is a tuple of two functions, each of which takes an `int` and returns an `int`. We can think of these functions as representing what you and Eve each do with the bits given to you by the referee.
- ➋ The classical random number generator the referee will use.
- ➌ You and Eve confer before the game begins and decide on a strategy. The `strategy` function will assign a function to `you` and `eve` one-bit functions that represent what you will do based on your input.
- ➍ The referee picks two random bits, one for each player.
- ➎ Give each player their random bit, then compute the parity of their responses.
- ➏ Check 4.1 to see if the players won.
- ➐ To finish your implementation, you will need a function that runs the CHSH game many times in a row and checks how often you and Eve win. Here, you can use Python's built-in `sum` function to count the number of times `referee` returns `True` for a particular `strategy`. Divide by the number of games that you played then gives you an estimate for the probability that your and Eve's strategy wins the CHSH game.

Note that in 4.1 we don't have a definition yet for the input `strategy` to the referee. Now that we have the rules of the game implemented in Python, let's talk *strategy* and

get to implementing a classical strategy for playing the CHSH game.

TIP

It's helpful to choose variable naming conventions that make it obvious what role each variable plays in your code. Above, we chose to use the prefix `n_` in the variable `n_games` to indicate that the variable refers to a number or size, and have used the prefix `idx_` to refer to the index for each individual game. Much like driving, it's good if your code is predictable.

4.1.3 Classical strategy

The simplest strategy for both you and Eve to pursue is to ignore your inputs entirely. Looking at [Table 4.3](#), if both of you agreed before the game that you would never change your outputs (i.e. always return 0), you will win 75% of the time (this does assume the referee chooses the random bits for each player uniformly).

Table 4.3. Best classical strategy for the CHSH game, where you both always respond with 0 with win conditions.

Your Input	Eve's Input	Your Response	Eve's Response	Parity	Win?
0	0	0	0	Even	Yes
0	1	0	0	Even	Yes
1	0	0	0	Even	Yes
1	1	0	0	Even	No

If we were to write this strategy as a Python function, we would have the code in [4.2](#).

Listing 4.2. An easy, constant strategy for players of a CHSH game. Also the best strategy you can use with only classical resources.

```
def constant_strategy() -> Strategy:
    return (
        lambda your_input: 0,
        lambda eve_input: 0
    )
```

Listing 4.3. Testing how often we could expect to win a round of CHSH if we use the `constant_strategy`.

```
>>> est_win_probability(constant_strategy) ①
0.771
```

- ① Note that you may get slightly more or less than 75% when you try this. This is because the win probability is estimated using a finite number of rounds (in stats, this is called a *binomial distribution*). For this example, we'd expect error bars of about 1.5%.

Ok, so that's an easy strategy, but is there anything more clever you can do? Given that you and Eve only have classical resources, sadly this is **probably** the best you can do.

Short of cheating 😶 (e.g. communicating or guessing the referee's inputs) you cannot win this game more than 75% of the time on average.

This all leads up to the obvious question: what if you and Eve could use qubits? What would be your best strategy then, and how often would you win? What does it say about our understanding of the universe if we have proof that you cannot win CHSH more than about 75% of the time, and then we find a way that we *can* beat that win rate? As you might guess, we can do better than 75% win rate playing CHSH if the players share quantum resources, i.e. have qubits. Later in the chapter we will get into quantum based strategies for CHSH, but spoiler we are going to need to simulate more than one qubit.

4.2 Working with multiple qubit states

So far in this book, we've only worked with one qubit at a time. To play a nonlocal game, for example, each player will need their own qubit. This raises the question, how do things change when the system we are considering has more than one qubit? The main difference is that we cannot describe each qubit individually and have to think in terms of a state that describes the whole system.

IMPORTANT

When describing a group or register of qubits, you generally *cannot* just describe each qubit individually. The most useful quantum behaviors can only be seen when you describe the state of a group or register of qubits.

The next section will help relate this system level view with a similar, classical programming concept of a register.

4.2.1 Registers

Suppose that we have a *register* of classical bits; that is, a collection of many classical bits. We can index through each bit in that register and look at its value independently, even though they are still a part of that register. The contents of the register can represent a more complex value, like bits that together represent a Unicode character (as we saw in Chapter 3), but this higher level interpretation is not necessary.

When we store information in a classical register, the number of different states of that register grows very rapidly as we add more bits. If we have three bits, for example, there are eight different states that our register can be in; see 4.4 for an example. We say for the classical register state `101` that the zeroth bit is a 1, the first is 0, and the second is 1. When these values are concatenated together give us the string `101`.

Listing 4.4. Using Python to write out all the possible states of a classical three bit register.

```
>>> from itertools import product
>>> ",".join(map(lambda x: x[0]+x[1], product("01", repeat=3)))
'000, 001, 010, 011, 100, 101, 110, 111'
```

If we have four bits, we can store one of 16 different states; if we have n bits, we can store one of 2^n . We say that the number of different possible states of a classical register *grows exponentially* with the number of bits. The bit strings output by 4.4 show the actual data in the register for each state of the register. They also serve as convenient labels for one of 8 possible messages that we can encode with 3 classical bits.

What does all this have to do with qubits? We saw in Chapters 2 and 3 that any state of classical bit also describes a qubit state. The same holds for registers of qubits as well. For instance, the three-bit state "010" describes the three-qubit state $|010\rangle$.

TIP

In Chapter 2, we saw that qubit states described by classical bits in this way are called computational basis states; here, we'll use the same term for states of multiple qubits that are described by strings of classical bits.

Just like with single qubits, however, the state of a register of multiple qubits can also be made up by adding different qubit states together. In the exact same way that we can write down $|+\rangle$ as $(|0\rangle + |1\rangle) / \sqrt{2}$ to get another valid qubit state, our three-qubit register can be in a wide variety of different states.

Example three-qubit states

- $(|010\rangle + |111\rangle) / \sqrt{2}$
- $(|001\rangle + |010\rangle + |100\rangle) / \sqrt{3}$

TIP

Why the square roots?

We'll see more as we go along, but just as we needed the square root of 2 to make the measurement probabilities work out for $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$, we need to divide by $\sqrt{2}$ and $\sqrt{3}$ in our examples above to make sure all the probabilities for each measurement are realistic; i.e. add up to one.

This example of the linearity of quantum registers is called the *superposition principle*.

IMPORTANT

Superposition principle

The *superposition principle* tells us that we can add together two different states of a quantum register together to get another valid state.

To write down the state of a quantum register in a computer, we'll again use vectors, just as we did in Chapter 2. The main difference is how many numbers we list in each

vector. Let's look at what it looks like to write down the state of a two-qubit register on a computer. For example, the vector for the two-qubit state $(|00\rangle + |11\rangle) / \sqrt{2}$ can also be written as the state $(1 \times |00\rangle + 0 \times |01\rangle + 0 \times |10\rangle + 1 \times |11\rangle) / \sqrt{2}$. If we make a list of what we had to multiply each computational basis state by in order to get the state we wanted, we have precisely the information we need to write down in our vector. In 4.5, we've written down $(|00\rangle + |11\rangle) / \sqrt{2}$ as a vector.

Listing 4.5. Using Python to write out an example of a two-qubit state.

```
>>> import numpy as np
>>> two_qubit_state = np.array([[ 1,
...     1,
...     0,
...     0,
...     1,
... ]]) / np.sqrt(2)
```

- ➊ We start the same way, using the `np.array` function to make a new vector.
- ➋ Each entry in this vector describes a different computational basis state. This entry tells us that we have to multiply $|00\rangle$ by 1.
- ➌ Similarly, this entry tells us how much of the $|01\rangle$ state we need to add to get the state that we want.
- ➍ Finally, we divide by $\sqrt{2}$ to make sure all the measurement probabilities work out, just as we did with the $|+\rangle$ state in Chapters 2 and 3.

The numbers in the vector from 4.5 are coefficients that we multiply by each of the computational basis states which we then add together to make the a new state. These coefficients are also called the *amplitudes* for each of the basis states in the sum.

TIP

Thinking with directions

Another way to think about this example is by thinking back to the maps that we saw in Chapter 2. Each different computational basis state tells us about a *direction* that a qubit state can be pointed in. We can think of the state of a two-qubit state as a direction in four dimensions instead of the two-dimensional maps we saw in Chapter 2. Since this book is two-dimensional rather than four-dimensional, we unfortunately can't draw a picture here, but sometimes thinking of vectors as directions can be more helpful than thinking of vectors as lists of numbers.

4.2.2 Why is it hard to simulate quantum computers?

We have seen above that as the number of bits grow, the number of different states a register can be in grows exponentially. While this won't be a problem for us in playing a nonlocal game with only two qubits, you'll want to use more than just two qubits as you proceed through this book.

When you do, you'll also have exponentially many different computational basis states for your quantum register, meaning that you will need exponentially many different

amplitudes in your vectors as you grow the size of your quantum register. To write down the state of a 10-qubit register, you'll need to use a vector that is a list of $2^{10} = 1024$ different amplitudes. For a 20-qubit register, you'll need about a million amplitudes in our vectors. By the time we get to 60 qubits, you'll need about 1.15×2^{18} numbers in our vectors. That's about one amplitude for each grain of sand on the planet.

Table 4.4. How much memory is required to store a quantum state?

# of qubits	# of amplitudes	Memory	Size comparison
1	2	128 bits	
2	4	256 bits	
3	8	512 bits	
4	16	1 kilobit	
8	256	4 kilobytes	Tap-enabled credit card
10	1024	16 kilobytes	
20	1,048,576	16 megabytes	
26	67,108,864	1 gigabyte	Raspberry Pi RAM
28	268,435,456	4 gigabytes	iPhone XS Max RAM
30	1,073,741,824	16 gigabytes	Laptop or desktop RAM
40	1,099,511,627,776	16 terabytes	
50	1,125,899,906,842,624	16 petabytes	
60	1,152,921,504,606,846,976	16 exabytes	
80	1,208,925,819,614,629,174,706,176	16 yottabytes	Approximate size of the internet
410	2.6×10^{123}	4.2×10^{124} bytes	Computer the size of the universe

In Table 4.4, we summarize what this exponential growth means for us as we try to simulate quantum computers using classical computers like phones, laptops, clusters, or cloud environments. This table shows that even though it's very challenging to reason about quantum computers using classical computers, you can pretty easily reason about small examples. With a laptop or desktop, you can simulate up to about 30 qubits without too much hassle. As you'll see throughout the rest of the book, this is more than enough to understand how quantum programs work, and to understand how quantum computers can be used to solve interesting problems.

DEEP DIVE: Are quantum computers exponentially more powerful?

You may have heard that the number of different numbers we have to keep track of to simulate a quantum computer using a classical computer is why quantum computers are more powerful, or that a quantum computer can store that much information. This isn't exactly true, though. A mathematical theorem known as *Holevo's theorem* tells us that a quantum computer made up of 410 qubits can store at most 410 classical bits of information, even if it would take us a classical computer about the size of the entire universe to write down the state of that quantum computer.

Put differently, just because it's hard to simulate a quantum computer doesn't mean that it does something useful. We'll see throughout the rest of the book that it takes a bit of art to figure out how to use a quantum computer to do solve useful problems.

Moreover, we run into exactly the same problem in some classical applications! If we want to keep track of a classical probability distribution, for instance, we would need to write down a number for each possible classical state.

4.2.3 Tensor products for state preparation

Describing quantum registers as vectors describing computational basis states is all well and good, but even if we know the state we want to get to, we need to know how to prepare it. For example, if one player in a nonlocal game has a qubit in the $|0\rangle$ state and the other has a qubit in the $|1\rangle$ state, we can combine those two single-qubit states in a straight-forward way to describe the state of the game as $|01\rangle$. What does it mean to "combine" the states of two (or more) qubits? We can do this by adding one more concept to our mathematics toolbox, called the *tensor product*.

In the same way that we used the product function in 4.4 above to combine labels for a three classical bit register, we can use the concept of a tensor product, written as \otimes , to combine the quantum states for each qubit together to make a state that describes multiple qubits. The output of product was a list of all possible states of those three classical bits. The output of a tensor product is similarly, a state that lists all computational basis states for a quantum register. You can use NumPy to compute tensor products for you; NumPy provides an implementation of the tensor product as the function `np.kron`, as shown in 4.6.

Why `kron`?

The name `np.kron` may seem odd for a function that implements tensor products, but the name is short for a related mathematical concept called the "Kronecker product." NumPy's use of `kron` as short for "Kronecker" follows the convention used in MATLAB, R, Julia and other scientific computing platforms.

Listing 4.6. Using NumPy to create a 2 qubit register state by combining $|0\rangle$ and $|1\rangle$ with a tensor product to make $|01\rangle$.

```
>>> import numpy as np
>>> ket0 = np.array([[1], [0]])          ①
>>> ket1 = np.array([[0], [1]])          ②
>>> np.kron(ket0, ket1)
array([[0],
       [1],
       [0],
       [0]])                                ③
```

- ① We start by defining vectors for the single-qubit states $|0\rangle$ and $|1\rangle$, just as we did in Chapters 2 and 3.

- 2 We can build the vector for $|0\rangle \otimes |1\rangle$ by calling NumPy's implementation of the tensor product. For historical reasons, the tensor product is represented in NumPy by the `kron` function.
- 3 The vector returned by `np.kron` has a 1 for the entry corresponding to the $|01\rangle$ computational basis state and zeroes everywhere else, so we recognize this vector as being the state $|01\rangle$.

This example shows us that $|0\rangle \otimes |1\rangle = |01\rangle$. That is, if we have the state of each qubit individually, we can combine them by using the tensor product to describe the state of the whole register.

We can combine as many qubits as we want this way; say you had four qubits, all in the $|0\rangle$ state. The register of all four qubits could be described as $|0\rangle \otimes |0\rangle \otimes |0\rangle \otimes |0\rangle$.

```
>>> import numpy as np
>>> ket0 = np.array([[1], [0]])
>>> from functools import reduce
>>> reduce(np.kron, [ket0] * 4)      1
array([[1],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0]]) 2
```

- 1 The `reduce` function provided with the Python standard library lets us apply a two-argument function like `kron` between each element in a list. Here, we use `reduce` instead of `np.kron(ket0, np.kron(ket0, np.kron(ket0, np.kron(ket0, ket0))))`.
- 2 We get back a four-qubit state vector representing $|0\rangle \otimes |0\rangle \otimes |0\rangle \otimes |0\rangle = |0000\rangle$.

IMPORTANT

We can't always go the other way!

The two-qubit state $(|00\rangle + |11\rangle) / \sqrt{2}$ that we saw earlier *cannot* be written as the tensor product of two single-qubit states. Multiple-qubit states that can't be written out as tensor products are called *entangled*. We'll see a lot more about entanglement throughout the rest of this Chapter, and in the rest of the book.

4.2.4 Tensor products for qubit operations on registers

Now that we know how to use the tensor product to combine quantum states together, what is `np.kron` actually doing? In essence, the tensor product is a table of every different way of combining its two arguments, as shown in 4.2.

Figure 4.2. The tensor product of two matrices shown step by step.

Let's consider a simple example, and take the tensor product of two matrices A and B :

$$A = \begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

The tensor product is denoted using the \otimes operator, just like products are indicated with \times .

$$\begin{aligned} C &= A \otimes B \\ &= \begin{bmatrix} 1 \times B & 3 \times B \\ 5 \times B & 7 \times B \end{bmatrix} \quad \text{To start, you first need to make copies of } B \text{ for each element in } A. \text{ It's a bit like making a big matrix by tiling } B. \\ &= \begin{bmatrix} 1 \times 2 & 1 \times 4 & 3 \times 2 & 3 \times 4 \\ 1 \times 6 & 1 \times 8 & 3 \times 6 & 3 \times 8 \\ 5 \times 2 & 5 \times 4 & 7 \times 2 & 7 \times 4 \\ 5 \times 6 & 5 \times 8 & 7 \times 6 & 7 \times 8 \end{bmatrix} \quad \text{After that, you can expand each copy of } B \text{ to get the pairs of numbers you need to multiply together to find the big matrix.} \\ &= \begin{bmatrix} 2 & 4 & 6 & 12 \\ 6 & 8 & 18 & 24 \\ 10 & 20 & 14 & 28 \\ 30 & 40 & 42 & 56 \end{bmatrix} \quad \text{At this point, you can find the answer by using ordinary (a.k.a. scalar) multiplication.} \end{aligned}$$

Each element of the resulting big matrix tells us the product of one element from A and one element from B . That is, the tensor product $A \otimes B$ is made up of all possible products of elements from A and B .

The same tensor product of two matrices in 4.2 is also shown in Python in 4.7.

Listing 4.7. Writing $|01\rangle$ as $1 \otimes X$ acting on $|00\rangle$.

```
>>> import numpy as np
>>> A = np.array([[1, 3], [5, 7]])      ①
>>> B = np.array([[2, 4], [6, 8]])      ②
>>> np.kron(A, B)
array([[ 2,  4,  6, 12],
       [ 6,  8, 18, 24],
       [10, 20, 14, 28],
       [30, 40, 42, 56]])
>>> np.kron(B, A)                      ③
array([[ 2,  6,  4, 12],
       [10, 14, 20, 28],
       [ 6, 18,  8, 24],
       [30, 42, 40, 56]])
```

- ① The matrices A and B are just arbitrary 2×2 matrices that we are using as examples here.
- ② As we saw above, `np.kron` is NumPy's implementation of the tensor product.

- ③ Note that the order of the arguments to the tensor product matters; although both `np.kron(A, B)` and `np.kron(B, A)` contain the same information, the entries in each are ordered quite differently!

Using the tensor product between two matrices, we can find how different quantum operations transform the state of a quantum register. We can also understand how a quantum operation transforms the state of multiple qubits by taking the tensor product of two matrices instead, letting us understand how your and Eve's moves in a nonlocal game affect your shared state.

For example, we know that we can write $|1\rangle$ as $X|0\rangle$ (that is, the result of applying the `x` instruction to an initialized qubit). This also gives us another way to write out multiple-qubit states like the $|01\rangle$ state we saw earlier. In this case, we can get $|01\rangle$ by applying an `x` instruction *only* to the second qubit of a two-qubit register. Using the tensor product, we can find a unitary matrix to represent this:

Listing 4.8. Writing $|01\rangle$ as $\mathbb{1} \otimes X$ acting on $|00\rangle$.

```
>>> import numpy as np
>>> I = np.array([[1, 0], [0, 1]])      1
>>> X = np.array([[0, 1], [1, 0]])      2
>>> IX = np.kron(I, X)                 3
>>> IX
array([[0, 1, 0, 0],           4
       [1, 0, 0, 0],
       [0, 0, 0, 1],
       [0, 0, 1, 0]])
>>> ket0 = np.array([[1], [0]])        5
>>> ket00 = np.kron(ket0, ket0)
>>> ket00
array([[1],
       [0],
       [0],
       [0]])
>>> IX @ ket00
array([[0],           6
       [1],
       [0],
       [0]])
```

- ① We start by defining a matrix that represents doing nothing to the first qubit, known as the identity matrix $\mathbb{1}$. Since $\mathbb{1}$ is hard to write in Python, we use `I` instead.
- ② Next, we define the unitary matrix X that lets us simulate the `x` instruction.
- ③ We can combine the two using the tensor product $\mathbb{1} \otimes X$.
- ④ The matrix $\mathbb{1} \otimes X$ consists of two copies of X , representing what happens to the second qubit for each possible state of the first qubit.
- ⑤ Let's see what happens when we use $\mathbb{1} \otimes X$ to simulate how the `x` instruction transforms the second qubit in a two-qubit register. We'll start with that register in the $|00\rangle = |0\rangle \otimes |0\rangle$ state.
- ⑥ We recognize the state we get back as the $|01\rangle$ state from earlier in this section. As expected, that is the state we get by flipping the second qubit from $|0\rangle$ to $|1\rangle$.

Exercise 4.2: Hadamard operation on a two qubit register.

How would you prepare a $|+0\rangle$ state? First, what vector would you use to represent the two-qubit state $|+0\rangle = |+\rangle \otimes |0\rangle$? You have an initial two qubit register in the $|00\rangle$ state. What operation should you apply to get the state you want? Hint: try $(H \otimes I)$ if you are stuck!

DEEP DIVE: Finally proving the no-cloning theorem

That operations on multiple qubits are also represented by unitary matrices lets us finally prove the No-cloning Theorem that we've seen a few times so far. The key insight is that that cloning a state isn't linear, and thus cannot be written as a matrix.

As with many proofs in mathematics, the proof of the No-cloning Theorem works by *contradiction*. That is, we assume the opposite of the theorem, and then show that we get something false as a result of that assumption.

Without further ado, then, we start by assuming that we have some wondrous instruction `clone` that can perfectly copy the state of its qubit. For instance, if we have a qubit `q1` whose state starts in $|1\rangle$, and a qubit `q2` whose state starts in $|0\rangle$, then after calling `q1.clone(q2)`, we would have the register $|11\rangle$.

Similarly, if `q1` starts in $|+\rangle$, then `q1.clone(q2)` should give us a register in the state $|++\rangle = |+\rangle \otimes |+\rangle$. The problem comes in reconciling what `q1.clone(q2)` should do in these two cases. We know that any quantum operation other than measurement must be linear, so let's give the matrix that lets us simulate `clone` a name; C seems pretty reasonable.

Using C , we can break down the case that we want to clone $|+\rangle$ into the case in which we want to clone $|0\rangle$ plus the case in which we want to clone $|1\rangle$. We know that $C|+0\rangle = C|++\rangle$, but we also know that $C|+0\rangle = C(|00\rangle + |10\rangle) / \sqrt{2} = (C|00\rangle + C|10\rangle) / \sqrt{2}$. Since `clone` needs to clone $|0\rangle$ and $|1\rangle$ as well as $|+\rangle$, we know that $C|00\rangle = |00\rangle$ and $C|10\rangle = |11\rangle$. That gives us that $(C|00\rangle + C|10\rangle) / \sqrt{2} = (|00\rangle + |11\rangle) / \sqrt{2}$, but we concluded earlier that $C|+0\rangle = |++\rangle = (|00\rangle + |01\rangle + |10\rangle + |11\rangle) / \sqrt{2}$.

We thus have a contradiction, and can conclude that we went wrong at the very first step, where we assumed that `clone` could exist! Thus, we have shown the No-cloning Theorem.

One important thing to note from this argument is that you can always copy information from one qubit to another *if you know the right basis*. The problem came in when we didn't know if we should copy information about $|0\rangle$ vs $|1\rangle$ or $|+\rangle$ vs $|-\rangle$, as we could copy either $|0\rangle$ or $|+\rangle$, but not both. This isn't a problem classically, as we only ever have the computational basis to work with.

4.3 QuTiP of the Iceberg

Up to this point, we've used NumPy to write our qubit simulator, `SingleQubitSimulator()`. This is very helpful, as without NumPy, we'd need to write our own matrix analysis functions and methods. It is often convenient, however, to rely on Python packages with special support for quantum concepts, building on the excellent numerical support provided by NumPy and SciPy (an extension to the NumPy numerical capabilities).

4.3.1 Quantum objects in QuTiP

One particularly useful package is the QuTiP (Quantum Toolbox in Python www.qutip.org) package, which provides built-in support for representing states and measurements as bras and kets, respectively, and for building matrices to represent quantum operations.

Just as `np.array` is at the core of NumPy, all of our use of QuTiP will center around the `Qobj` class (short for "quantum object"). This class encapsulates vectors and matrices, providing additional metadata and useful methods that will make it easier for us to improve our simulator. You can see in 4.3 an example of creating a `Qobj` from a vector, which keeps track of some metadata.

Selected metadata that the `Qobj` class tracks:

- `data` holds the array representing the `Qobj`,
- `dims` the size of our quantum register. You can think of it as a way of keeping track of how we record the qubits we are dealing with,
- `shape` keeps the dimension of the original object we used to make the `Qobj`. Similar to the `np.shape` attribute,
- `type` what the `Qobj` represents (a state = ket, a measurement = bra, or an operator = oper).

Figure 4.3. Properties of the `Qobj` class.

The QuTiP package provides the `Qobj` class for representing "quantum objects."

The `Qobj` initializer takes a list of the elements of the new object, similar to `np.array`.

The screenshot shows a Jupyter QtConsole window. In cell [1], the user imports `qutip as qt` and `numpy as np`. In cell [2], they create a `Qobj` from a list of two arrays: `[[np.cos(0.2)], [np.sin(0.2)]]`. The output shows the `Qobj` has `dims = [[2], [1]]`, `shape = (2, 1)`, and `type = ket`. The `data` attribute is `[[0.98006658], [0.19866933]]`. Annotations explain these properties: `type` tells if it's a ket, bra, or matrix; `dims` breaks it into qubits; `shape` is like `np.shape`.

```
In [1]: import qutip as qt
...: import numpy as np
In [2]: qt.Qobj([[np.cos(0.2)], [np.sin(0.2)]])
Out[2]:
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[ 0.98006658]
 [ 0.19866933]]
In [3]:
```

Let's try importing QuTiP and asking it for the Hadamard operation, see 4.9.

NOTE

Make sure as you run things that you are in the right conda env, for more information see Appendix A.

Listing 4.9. Using the QuTiP Python package to load the package's representation of the Hadamard operation.

```
>>> import qutip as qt
>>> H = qt.hadamard_transform()
>>> H
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True ①
Qobj data =
[[ 0.70710678  0.70710678]
 [ 0.70710678 -0.70710678]]
```

- ① Note that QuTiP prints out some diagnostic information about each Qobj instance along with the data itself. Here, for instance, `type = oper` tells you that `H` represents an operator (a more formal term for the matrices we've seen so far), along with some information about the dimensions of the operator represented by `H`. Finally, the `isherh = True` output tells you that `H` is an example of a special kind of matrix called a *Hermitian operator*. We'll see more about why Hermitian operators are helpful in Chapter 9.

We can make new instances of Qobj in much the same way as we made NumPy arrays, by passing in Python lists to the Qobj initializer, see [4.10](#).

Listing 4.10. Using a vector to initialize a Qobj that represents the state of a qubit.

```
>>> import qutip as qt
>>> ket0 = qt.Qobj([[1], [0]]) ①
>>> ket0
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket ②
Qobj data =
[[1.]
 [0.]]
```

- ① One key difference between creating Qobj instances and arrays is that when we create Qobj instances, we always need two levels of lists. The outer list is a list of rows in the new Qobj instance.
- ② QuTiP prints some metadata about the size and shape of the new quantum object, along with the data contained in the new object. In this case, the data for the new Qobj that you constructed has two rows, each with one column. We identify that as the vector or ket that we use to write down the $|0\rangle$ state.

Exercise 4.3: Creating the Qobj for other states

How would you create a Qobj that initialized the $|1\rangle$ state? How about the $|+\rangle$ or $|-\rangle$ state? If you need to check back to *Simulating qubits in code* section of Chapter 2 for what vectors represent those states.

Where QuTiP really helps out, though, is that it provides us with a lot of nice

shorthand for the kinds of objects we need to work with in quantum computing. For instance, we could have also made `ket0` in the above sample by using the QuTiP `basis` function, see [4.11](#).

Listing 4.11. Using built-in features of QuTiP to easily create the $|0\rangle$ and $|1\rangle$ states.

```
>>> import qutip as qt
>>> ket0 = qt.basis(2, 0)                                     ①
>>> ket0
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket ②
Qobj data =
[[1.]
 [0.]]
>>> ket1 = qt.basis(2, 1)                                     ③
>>> ket1
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[0.]
 [1.]]
```

- ➊ The `basis` function takes two arguments. The first tells QuTiP that we want a qubit state; since a qubit can be in either of two different states, you can pass a 2 here to indicate a single qubit. It's 2 for a single qubit because the length of a vector that is needed to represent a single qubit is 2. The second argument tells QuTiP which basis state you want. Since we want $|0\rangle$, we pass a 0 here.
- ➋ Note that we get exactly the same output here as in the previous example.
- ➌ We can also construct a quantum object for $|1\rangle$ by passing a 1 instead of a 0.

...basis?

As we have seen before, the states $|0\rangle$ and $|1\rangle$ make up the *computational basis* for a single qubit. The QuTiP function `basis` gets its name from this definition, as it makes quantum objects to represent computational basis states.

There are more things in heaven and earth, than your qubits.

It may seem a little odd that we had to tell QuTiP that we wanted a qubit. After all, what else *could* we want? As it turns out, quite a bit (yes, pun very much so intended)!

There are many other ways to represent classical information than bits, such as *trits*, which have three possible values. We tend not to see classical information represented using anything other than bits when we write programs, though, as it's very useful to pick a convention and stick with it. Things other than bits still have their uses, though, in specialized domains such as telecommunications systems.

In the exact same fashion, quantum systems can have any number of different states, so that we can have qutrits, qu4its, qu5its, qu17its, and so forth, collectively known as *qudits*. While representing quantum information using qudits other than qubits can be useful in some cases, and can have some very interesting mathematical properties, qubits give us all we need to dive into quantum programming, so we'll focus on them until we get to Part III and meet our first *fermions*.

Exercise 4.4: Using `qt.basis` for multiple qubits

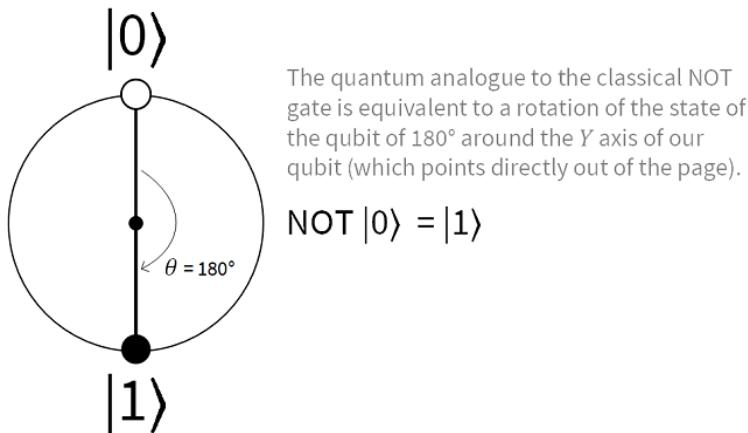
How could you use the `qt.basis` function to create a two qubit register in the $|10\rangle$ state? How could you create the $|001\rangle$ state? Remember that the second argument to `qt.basis` is an index to the computational basis states we saw earlier.

QuTiP also provides a number of different functions for making quantum objects to represent unitary matrices. For instance, we can make a quantum object for the X matrix by using the `sigmax` function:

```
>>> import qutip as qt
>>> qt.sigmax()
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[0. 1.]
 [1. 0.]]
```

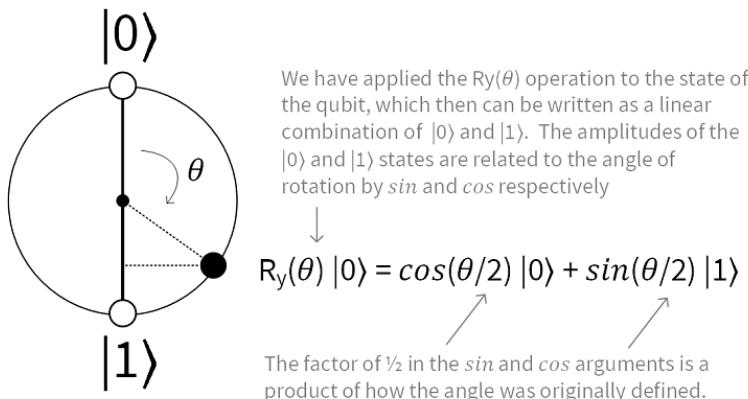
As we saw in Chapter 2, the matrix for `sigmax` represents a rotation of 180° ([4.4](#)).

Figure 4.4. A visualization of the quantum equivalent of a NOT operation operating on a qubit in the $|0\rangle$ state, leaving the qubit in the $|1\rangle$ state.



QuTiP also provides a function `ry` to represent rotating by whatever angle we like instead of 180° like the `x` operation. We have actually already seen the operation that `ry` represents in Chapter 2, when we considered rotating $|0\rangle$ by an arbitrary angle θ . See [4.5](#) for a refresher on the operation we now know as `ry`.

Figure 4.5. A visualization of QuTiP function `ry` which corresponds to a variable rotation of θ around the Y axis of our qubit (which points directly out of the page).



Now that we have a few more single-qubit operations down, how can we easily simulate multi-qubit operations in QuTiP? We can use QuTiP's tensor function to quickly get up and running with tensor products to make our multi-qubit registers and operations, as we show in [4.12](#).

Listing 4.12. Tensor products in QuTiP

```
>>> import qutip as qt
>>> psi = qt.basis(2, 0)          ①
>>> phi = qt.basis(2, 1)          ②
>>> qt.tensor(psi, phi)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.]
 [ 1.]
 [ 0.]
 [ 0.]]                                ③
>>> H = qt.hadamard_transform(1)    ④
>>> I = qt.qleye(2)               ⑤
>>> qt.tensor(H, I)               ⑥
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
Qobj data =
[[ 0.70710678  0.          0.70710678  0.          ]
 [ 0.          0.70710678  0.          0.70710678]
 [ 0.70710678  0.          -0.70710678  0.          ]
 [ 0.          0.70710678  0.          -0.70710678]]
>>> (                               ⑦
...     qt.tensor(H, I) * qt.tensor(psi, phi) -
...     qt.tensor(H * psi, I * phi)
... )
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.]
 [ 0.]]
```

```
[ 0.]
[ 0.]]
```

- ➊ Set `psi` to represent $|\psi\rangle = |0\rangle$.
- ➋ Set `phi` to represent $|\varphi\rangle = |1\rangle$.
- ➌ After calling `tensor`, QuTiP tells us the amplitudes for each classical label in $|\psi\rangle \otimes |\varphi\rangle = |0\rangle \otimes |1\rangle = |01\rangle$, using the same order as [4.4](#).
- ➍ Set `H` to represent the Hadamard operation discussed above.
- ➎ You can use the `qeye` function provided by QuTiP to get a copy of a `Qobj` instance representing the identity matrix that we first saw in [4.8](#). Since identity matrices are often written using the letter "I," many scientific computing packages use the name `eye` as a bit of a pun to refer to the identity matrix.
- ➏ The unitary matrices representing quantum operations combine using tensor products in the same way as states and measurements.
- ➐ If we apply a unitary to a state then take the tensor product, we get the same answer as if we applied the tensor product and then the unitary. Here we can tell these two approaches are give the same answer because their difference is 0. In math, we would say that for any unitary operators U and V and for any states $|\psi\rangle$ and $|\varphi\rangle$, $(U|\psi\rangle) \otimes (V|\varphi\rangle) = (U \otimes V)(|\psi\rangle \otimes |\varphi\rangle)$.

NOTE For a list of all of the built-in states and operations QuTiP has, see qutip.org/docs/latest/guide/guide-basics.html#states-and-operators

4.3.2 Upgrading the simulator

The goal now is to use QuTiP to upgrade our single-qubit simulator to a multi-qubit simulator with some of the features of QuTiP. We will do this by adding a few features to our single-qubit simulator from Chapters 2 and 3.

The most significant change we'll need to make to our simulator from previous chapters is that we can no longer assign a state to each qubit. Rather, we must assign a state to the entire *register* of qubits in our device, since some of the qubits may be entangled with each other.

Let's jump into making the modifications necessary to separate the concept of the state to the device level.

NOTE To see the code we wrote earlier, as well as the samples for this chapter see the GitHub repo for the book: github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

To review, we have two files for our simulator: the interface (`interface.py`), and the simulator itself (`simulator.py`). The device interface (`QuantumDevice`) defines a way of interacting with an actual or simulated quantum device, which is represented in Python as an object that lets us allocate and deallocate qubits.

We won't need anything new for the `QuantumDevice` class in the interface in order to model our CHSH game, since we'll still need to allocate and deallocate qubits. Where

we can actually add features is in the `Qubit` class provided along with our `SingleQubitSimulator` in `simulator.py`.

Now we need to consider what, if anything, needs to change in our interface for a `Qubit` we allocate from the `QuantumDevice`. In Chapter 2, we saw that the Hadamard operation was useful for rotating qubits between different bases to make a QRNG. Let's build on this by adding a new method to `Qubit` to allow quantum programs to send a new kind of rotation instruction that we will need to use the quantum strategy for CHSH.

Listing 4.13. The interface we have built in Chapter 3 for our `Qubit`, but adding a new `ry` operation which rotates our state about the Y -axis.

```
class Qubit(metaclass=ABCMeta):
    @abstractmethod
    def h(self): pass

    @abstractmethod
    def x(self): pass

    @abstractmethod
    def ry(self, angle : float): pass      ①

    @abstractmethod
    def measure(self) -> bool: pass

    @abstractmethod
    def reset(self): pass
```

- ① This is the abstract method `ry` which takes an argument `angle` to specify how far to rotate the qubit around the Y -axis.

That should then cover all changes we need to make to our `Qubit` and `QuantumDevice` interface for playing CHSH with Eve. Now we need to address what changes we need to make to the `simulator.py` to actually allow it to allocate, operate, and measure multi-qubit states.

The main changes to our `Simulator` class which implements a `QuantumDevice` are that we now need some attributes to track how many qubits it has and the overall state of the register. 4.14 shows these changes as well as an update to allocation and deallocation methods.

Listing 4.14. The implementation of `Simulator`, a class that represents a device that represents a multi-qubit version of a `QuantumDevice` from our interface.

```
class Simulator(QuantumDevice):
    capacity : int
    available_qubits : List[SimulatedQubit] ② ③
```

```

register_state : qt.Qobj
def __init__(self, capacity=3):
    self.capacity = capacity
    self.available_qubits = [
        SimulatedQubit(self, idx)
        for idx in range(capacity)
    ]
    self.register_state = qt.tensor(
        *[
            qt.basis(2, 0)
            for _ in range(capacity)
        ]
    )
def allocate_qubit(self) -> SimulatedQubit:
    if self.available_qubits:
        return self.available_qubits.pop()

def deallocate_qubit(self, qubit : SimulatedQubit):
    self.available_qubits.append(qubit)

```

- ➊ We have changed the name from `SingleQubitSimulator` to `Simulator` to indicate it is more generalized. Here, that means we can actually simulate multiple qubits with it.
- ➋ The more general `Simulator` class now needs a few attributes, the first being `capacity` which represents the number of qubits it is capable of simulating.
- ➌ `available_qubits` is a list containing the qubits the `Simulator` is using.
- ➍ `register_state` uses the new QuTiP `Qobj` to represent the state of the entire simulator.
- ➎ A list comprehension allows us to make a list of available qubits by calling `SimulatedQubit` with the indices from the range of `capacity`.
- ➏ `register_state` is initialized by taking the tensor product of a number of copies of the $|0\rangle$ state equal to the capacity of the simulator. The `*[...]` notation will take the generated list and turn it into a sequence of arguments for `qt.tensor`.
- ➐ The `allocate_qubit` and `deallocate_qubit` methods are the same as our previous simulator from Chapter 3.

IMPORTANT**Peer not into the box, mortal!**

In just the same way that we used NumPy to represent the state of a simulator, the `register_state` property of our newly upgraded simulator uses QuTiP to predict how each instruction has transformed the state of our register. When we write quantum programs, though, we do so against the interface in [4.13](#), which doesn't have any way to let us access `register_state`.

We can think of the simulator as being a kind of black box that *encapsulates* the notion of a state. If our quantum programs were able to look inside that box, then they would be able to cheat by copying that information in ways forbidden by the no-cloning theorem. This means that for a quantum program to be correct, we cannot look inside the simulator to see its state.

In this Chapter, we'll cheat a little bit, but in the next Chapter, we'll see how to fix that up to make sure our programs can be run on actual quantum hardware.

We also will add a new *private* method to our `Simulator` that allows us to apply

operations to specific qubits in our device. This will let us write methods on our qubits that send operations back to the simulator to be applied to the state of an entire register of qubits.

TIP

Python is not strict about keeping methods or attributes private, but we will prefix this method name with an underscore to indicate it is meant for use in the class only.

Listing 4.15. One additional method for the Simulator class that allows us to apply operations to specific qubits in our simulator.

```
def _apply(self, unitary : qt.Qobj, ids : List[int]):  
    if len(ids) == 1:  
        matrix = qt.circuit.gate_expand_1toN(  
            unitary, self.capacity, ids[0]  
        )  
    else:  
        raise ValueError("Only single-qubit unitary matrices are supported.")  
  
    self.register_state = matrix * self.register_state
```

- ➊ The private method `_apply` takes an input unitary of type `Qobj` that represents a unitary operation to be applied, and a list of `int` to indicate the indices of the `available_qubits` list where you want to apply the operation. For now, that list will only ever contain one element, since we're only implementing single-qubit gates in our simulator. We'll relax this in the next Chapter, though.
- ➋ If the operation is only to be applied to one qubit, we can use QuTiP to generate the matrix that we would need to apply the operation to the right qubit given the representation of the entire register by applying `1` everywhere else. This is done for us automatically by the `gate_expand_1toN` function.
- ➌ Now that we have the right matrix to multiply our entire `register_state` by, we can update the value of that register accordingly.

Now let's get to the implementation of `SimulatedQubit`, the class that represents how we simulate a single qubit, given that we know it is part of a device that has multiple qubits. The main difference between the single and multi-qubit versions of `SimulatedQubit` is that we need each qubit to remember their "parent" device and their location or `id` in that device so that we can associate the state with the register and not each qubit. This is important as we will see in the next section when we want to measure qubits in a multi-qubit device.

Listing 4.16. The start of the class SimulatedQubit which implements a qubit from a multi-qubit device.

```
class SimulatedQubit(Qubit):  
    qubit_id : int  
    parent : "Simulator"  
  
    def __init__(self, parent_simulator : "Simulator", id : int):  
        self.qubit_id = id
```

```

    self.parent = parent_simulator

    def h(self) -> None:
        self.parent._apply(H, [self.qubit_id])          ②

    def ry(self, angle : float) -> None:
        self.parent._apply(qt.ry(angle), [self.qubit_id]) ③

    def x(self) -> None:
        self.parent._apply(qt.sigmax(), [self.qubit_id])

```

- ➊ Now to initialize a qubit we will need the name of the parent simulator (so we can easily associate it) and the index of the qubit in the simulator's register. `__init__` then sets those attributes and resets the qubit to the $|0\rangle$ state.
- ➋ To implement the `h` operation now, we ask the parent of our `SimulatedQubit` (which is an instance `Simulator`) to use the `_apply` method to generate the right matrix that would represent the operation on the complete register, and then update the `register_state`.
- ➌ We can also pass the parameterized `qt.ry` operation from QuTiP to `_apply` to rotate our qubit about the Y -axis by an angle `angle`.

Great! Now we are almost done upgrading our simulator to use QuTiP and support multiple qubits. We will tackle simulating measurement on multi-qubit states, in the next section.

4.3.3 Measuring up: How can we measure multiple qubits?

TIP

This section is one of the hardest sections in the book, please don't worry if things don't make a lot of sense the first time around.

In some sense, measuring multiple qubits works the same way we're used to from measuring single-qubit systems. We can still use Born's rule to predict the probability of any particular measurement outcome. For example, let's return to the $(|00\rangle + |11\rangle) / \sqrt{2}$ state that we've seen a few times now. If we were to measure a pair of qubits in that state, we would get either "00" or "11" as our classical outcomes with equal probability, since both have the same amplitude, namely $1 / \sqrt{2}$.

Similarly, we'll still demand that if we measure the same register twice in a row, we get the same answer. If we get the "00" outcome, for instance, we know that qubits are left in the $|00\rangle = |0\rangle \otimes |0\rangle$ state.

Where this gets a little bit trickier, however, is if we measure *part* of a quantum register without measuring the whole thing. Let's look at a couple examples to see how that could work. Again taking $(|00\rangle + |11\rangle) / \sqrt{2}$ as an example, if we measure *only* the first qubit and we get a "0", we know that we need to get the same answer again the next time we measure. The only way this can happen is if the state transforms to $|00\rangle$ as a result of having observed "0" on the first qubit.

On the other hand, what happens if we measure the first qubit out of a pair of qubits in the $|++\rangle$ state? It's helpful to first refresh our memory as to what $|++\rangle$ looks like when written as a vector.

```
>>> import qutip as qt
>>> ket_plus = qt.hadamard_transform() * qt.basis(2, 0)           ①
>>> ket_plus
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket      ②
Qobj data =
[[0.70710678]
 [0.70710678]]
>>> register_state = qt.tensor(ket_plus, ket_plus)                 ③
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.5]
 [0.5]
 [0.5]
 [0.5]]
```

- ① Let's start again by writing down $|+\rangle$ as $H|0\rangle$. In QuTiP, we'll use the `hadamard_transform` function to get a `Qobj` instance to represent H , and we'll use `basis(2, 0)` to get a `Qobj` representing $|0\rangle$.
- ② We can print out `ket_plus` to get a list of the elements in that vector; as before, we call each of these elements an *amplitude*.
- ③ To represent the state $|++\rangle$, we use that $|++\rangle = |+\rangle \otimes |+\rangle$.
- ④ This output tells us that $|++\rangle$ has the same amplitude each of the four computational basis states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, just as `ket_plus` has the same amplitude for each of the computational basis states $|0\rangle$ and $|1\rangle$.

Suppose that we measure the first qubit and get a "1" outcome. To make sure we get the same result the next time we measure, the state after measurement can't have any amplitude on $|00\rangle$ or $|01\rangle$. If we only keep the amplitudes on $|10\rangle$ and $|11\rangle$ (the third and four rows of the vector we calculated above), then we get that the state of our two qubits becomes $(|10\rangle + |11\rangle) / \sqrt{2}$.

IMPORTANT

Where did the $\sqrt{2}$ come from?

We included a $\sqrt{2}$ above to make sure that all our measurement probabilities still sum to one when we measure the second qubit. In order for Born's rule to make any sense, we always need the sum of the squares of each amplitude to sum to one.

There's another way to write this state, though, that we can check using QuTiP:

```
>>> import qutip as qt
>>> ket_0 = qt.basis(2, 0)
>>> ket_1 = qt.basis(2, 1)
>>> ket_plus = qt.hadamard_transform() * ket_0                  ①
>>> qt.tensor(ket_1, ket_plus)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
```

```
[[0.      ]
 [0.      ]
 [0.70710678]
 [0.70710678]]
```

- ➊ Recall that we can write $|+\rangle$ as $H|0\rangle$.

This tells us that if we only keep the parts of the state $|++\rangle$ that are consistent with getting a "1" outcome from measuring the first qubit, then we get $|1+\rangle = |1\rangle \otimes |+\rangle$. That is, nothing happens at all to the second qubit in this case!

Exercise 4.5: Measuring the other qubit

In the example where our two qubits start off in the $|++\rangle$ state, suppose we measured the second qubit instead. Check that no matter what result we get, nothing happens to the state of the first qubit.

To work out measuring part of a register more generally, we can use another concept from linear algebra called *projectors*. A projector is the product of a state vector (the "ket" or $|$ part of a bra-ket) and a measurement (the "bra" or $\langle |$ part of a bra-ket), and represents our requirement that *if* a certain measurement outcome occurs, *then* we must transform to a state consistent with that measurement. See [4.6](#) for a quick example of a single-qubit projector; defining projectors on multiple qubits works in exactly the same way.

Figure 4.6. An example of a projector acting on a single-qubit state.

This is an example of a projector; that is, a matrix that squares to itself.

In this example, we get our projector by multiplying a ket with a bra. When we compute inner products such as for Born's rule, you've generally done the opposite: multiplied a bra with a ket.

When you do so, you get that the bra is represented by a row vector, so that the projector is a product of a column with a row.

$$|0\rangle\langle 0| = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix} \right)^\dagger = \begin{bmatrix} 1 \\ 0 \end{bmatrix} [1 \quad 0] = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Recall that you can turn a ket like $|0\rangle$ into a bra like $\langle 0|$ by taking the conjugate transpose (written as a dagger).

Working through the matrix multiplication, we get a 1 in the $|0\rangle$ row and the $|0\rangle$ column, and zeroes everywhere else.

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \end{bmatrix}$$

When we multiply an arbitrary state vector by this projector, it projects out or filters out all but the part corresponding to the $|0\rangle$ computational basis state.

What you're left with might no longer be a valid state, as it might not have the right length. The length tells us about how much of the original state vector was picked out by our projector. In your simulator, you'll use this to find the probability for each measurement outcome, giving us another way to compute Born's rule.

In QuTiP, we write the bra corresponding to a ket by using the `.dag()` method (short for dagger, a call back to mathematical notation we saw in 4.6). Thankfully, even if the math isn't that straightforward, it winds up not being that bad to write in Python, as we can see in 4.16.

Listing 4.17. A method for the Simulator class that allows us to measure individual qubits in our register.

```
def measure(self) -> bool:
    projectors = [
        qt.circuit.gate_expand_1toN(
            qt.basis(2, outcome) * qt.basis(2, outcome).dag(),
            self.parent.capacity,
            self.qubit_id
        )
        for outcome in (0, 1)
```

1
2

```

        ]
post_measurement_states = [
    projector * self.parent.register_state
    for projector in projectors
]
probabilities = [
    post_measurement_state.norm() ** 2
    for post_measurement_state in post_measurement_states
]
sample = np.random.choice([0, 1], p=probabilities)
self.parent.register_state = post_measurement_states[sample].unit()
return int(sample)

def reset(self) -> None:
    if self.measure(): self.x()

```

- ➊ We start by using QuTiP to make a list of projectors, one for each possible measurement outcome. Don't worry too much about this for now, we'll get more practice with this concept as we go along.
- ➋ Just as you did in [4.15](#), you can use the `gate_expand_1toN` function provided by QuTiP again here to expand each single-qubit projector into a projector that acts on the whole register.
- ➌ Next, we use each projector to pick out the parts of a state that are consistent with each measurement outcome.
- ➍ The length of what each projector picks out (written as the `.norm()` method in QuTiP) tells us about the probability of each measurement outcome.
- ➎ Once we have the probabilities for each outcome, we can pick an outcome by using NumPy.
- ➏ Next, you can use the `.unit()` method built-in to QuTiP to make sure that whatever state we get, its measurement probabilities still sum up to one so that you're ready for the next measurement.
- ➐ Finally, you can use the new measurement method to implement the `reset` method from Chapters 2 and 3. Here, if the result of a measurement is $|1\rangle$, then flipping with an `x` instruction resets back to $|0\rangle$.

4.4 CHSH: Quantum strategy

Now that we have expanded our simulator to handle multiple qubits, let's see how we can simulate a *quantum* based strategy for our players that will result in a win probability higher than could be possible with any classical strategy! See [4.7](#) for a reminder of how the CHSH game is played.

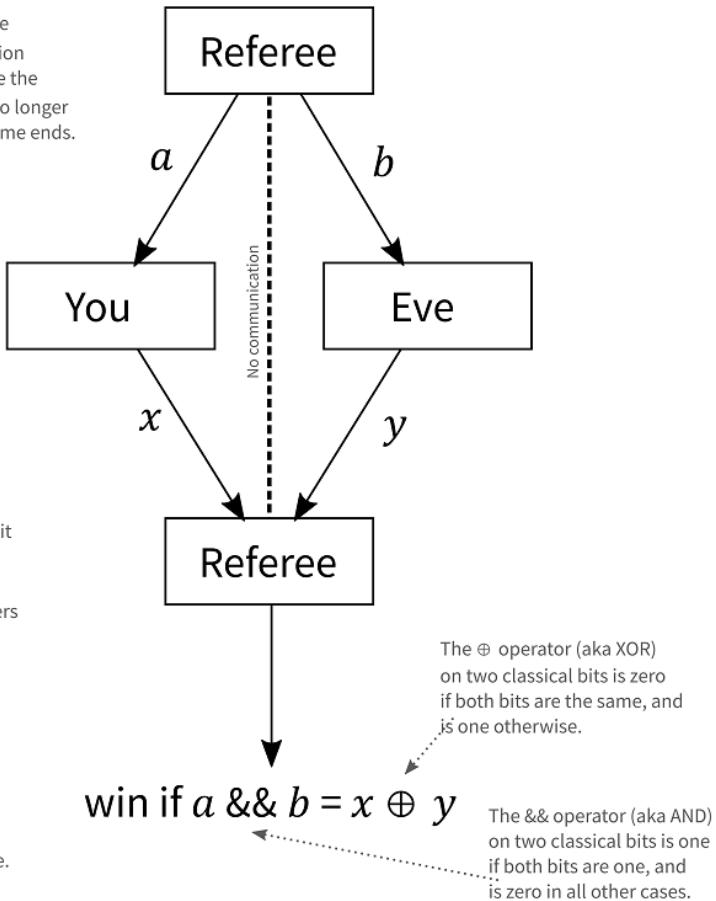
Figure 4.7. The CHSH game, a nonlocal game with two players and a referee.

- 1.** A referee sends you and Eve a one-bit question. Your question is labeled a and Eve's is b . Once the game starts, you and Eve can no longer communicate until after the game ends.

- 2.** The referee demands a one-bit answer from each player, here your answer is labeled x and Eve's is labeled y .

- 3.** Both players send a one-bit answer back to the referee. The referee then scores the game and declares if the players won.

- 4.** The scoring rules require that you and Eve answer with the correct parity given the input bits from the Referee.



You and Eve now have quantum resources, so let's start with the simplest option: you each have one qubit allocated from the same device. We'll use our simulator to implement this strategy, so this isn't really a test of quantum mechanics itself so much as that our simulator agrees with quantum mechanics.

NOTE

We can't simulate the players being truly nonlocal, as the parts of the simulator need to communicate in order to emulate quantum mechanics. Faithfully simulating quantum games and quantum networking protocols in this manner exposes a lot of interesting classical networking topology questions that are well beyond the scope of this book. If you're interested in simulators intended more for use in quantum networking than quantum computing, we recommend looking at the SimulaQron project (www.simulacron.org/) for more information.

Let's see how often you and Eve can win if you each start off with a single qubit, and if those qubits start off in the $(|00\rangle + |11\rangle) / \sqrt{2}$ state that we've seen a few times so far in this Chapter. Don't worry about how to prepare this state, we'll see how to do that in

the next Chapter. For now, let's just see what you can do with qubits in that state once you have them.

Using these qubits, we can form a new quantum strategy for the CHSH game we saw at the start of the chapter. The trick is that you and Eve can each apply operations to each of your qubits once you get your respective messages from the referee. As it turns out, ry is a very useful operation for this strategy. It lets you and Eve trade off slightly between how often you win when the referee asks for you both to output the same answers (the 00, 01, and 10 cases), in order to do slightly better when you need to output different answers (the 11 case), as shown in [4.8](#) and in [4.2](#).

Figure 4.8. Rotating qubits to win at CHSH

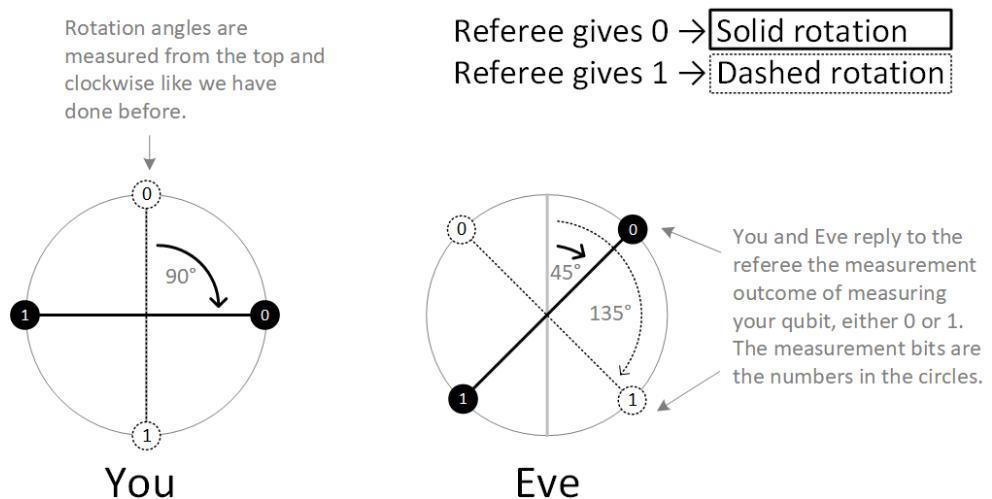


Table 4.5. Rotations you and Eve will do to your qubits as a function of the input bit you receive from the referee. Note they are all ry rotations, just by different angles (converted to radian for ry).

Input from referee	Your Rotation	Eve's Rotation
0	$ry(90 * np.pi / 180)$	$ry(45 * np.pi / 180)$
1	$ry(0)$	$ry(135 * np.pi / 180)$

Don't worry if these angles look random, we can check to see that they work using our new simulator! In [4.17](#), we've used the new features we added to the simulator to write out a quantum strategy.

Listing 4.18. A new quantum strategy for playing the CHSH game where you and Eve start with a pair of qubits.

```

import qutip as qt
def quantum_strategy(initial_state : qt.Qobj) -> Strategy:
    shared_system = Simulator(capacity=2)           ①
    shared_system.register_state = initial_state
    your_qubit = shared_system.allocate_qubit()      ②
    eve_qubit = shared_system.allocate_qubit()

    shared_system.register_state = qt.bell_state()    ③
    your_angles = [90 * np.pi / 180, 0]               ④
    eve_angles = [45 * np.pi / 180, 135 * np.pi / 180]

    def you(your_input : int) -> int:
        your_qubit.ry(your_angles[your_input])          ⑤
        return your_qubit.measure()                     ⑥

    def eve(eve_input: int) -> int :
        eve_qubit.ry(eve_angles[eve_input])             ⑦
        return eve_qubit.measure()

    return you, eve                                   ⑧

```

- ① To start the quantum strategy, we need to create a `QuantumDevice` instance where we will simulate our qubits.
- ② Labels can be assigned to each qubit as we allocate them to the `shared_system`.
- ③ We will cheat a little here to just set the state of our qubits to the entangled state $(|00\rangle + |11\rangle)/\sqrt{2}$, we will see in the next chapter how to prepare this state from scratch, and why the function to prepare this state is called `bell_state`.
- ④ The angles for the rotations you and Eve need to do based on your input from the referee.
- ⑤ Your strategy for playing the CHSH game starts with you rotating your qubit based on the input classical bit from the referee.
- ⑥ The classical bit value your strategy returns is the bit value you get when you measure your qubit.
- ⑦ Eve's strategy is similar to yours, just uses different angles for her initial rotation.
- ⑧ Just like our classical strategy, `quantum_strategy` returns a tuple of functions that represent your and Eve's individual actions.

Now that we have implemented a Python version of the `quantum_strategy`, let's see how often we can win with using our CHSH game `est_win_probability` function in 4.18.

Listing 4.19. We can run our CHSH game win estimator with the new quantum_strategy.

Note that this is a higher win probability than with the classical strategy!

```

>>> est_win_probability(quantum_strategy) ①
0.832

```

- 1 Note that you may get slightly more or less than 85% when you try this. This is because the win probability is estimated under the hood using a binomial distribution. For this example, we'd expect error bars of about 1.5%.

Awesome, you and Eve can start winning the CHSH game more frequently than any other classical players! What this strategy shows us, though, is an example of how states like $(|00\rangle + |11\rangle) / \sqrt{2}$ are an important resource provided by quantum mechanics.

NOTE

States like $(|00\rangle + |11\rangle) / \sqrt{2}$ are called *entangled*, as they can't be written as the tensor product of single-qubit states. We'll see many more examples of entanglement as we go along, but entanglement is of the most amazing and fun things that we get to use in writing quantum programs!

As we saw in this example, entanglement allows us to create correlations in our data that can be used to our advantage when we want to get useful information out of our quantum systems.

NOTE

The speed of light is still a thing

As it turns out, entanglement can **never** be used to communicate a message of your choosing all on its own, though. You always need to send something else along with using your entanglement. This means that the speed of light still constrains how fast information can travel through the universe.

Far from being strange or weird, though, entanglement is a direct result of what we've already learned about quantum computing so far: it is a direct consequence of quantum mechanics being **linear**. If we can prepare a two qubit register in the $|00\rangle$ state and in the $|11\rangle$ state, then we can also prepare a state in a linear combination of the two, such as $(|00\rangle + |11\rangle) / \sqrt{2}$.

Since entanglement is an direct result of the linearity of quantum mechanics, the CHSH game also gives us a great way to check that quantum mechanics is really correct (or to the best our data can show). Let's go back to that win probability in 4.18 again. If you do an *experiment*, and you see something like an 83.2% win probability, that tells you that your experiment couldn't have been purely classical, since we know a classical strategy can *at most* win 75% of the time. This experiment has been done many times throughout history, and is part of how we know that our universe isn't just classical — that we need quantum mechanics to describe it.

NOTE

In 2015, one experiment even had the two players in the CHSH game separated by over a kilometer!

Self-testing: An application for nonlocal games

This hints at another application for nonlocal games: if we can play *and win* a nonlocal game with Eve, then that means along the way, we must have built something that we can use to send quantum data. This sort of insight leads to ideas known as *quantum self-testing*, where we make parts of a device play nonlocal games with other parts of the device in order to make sure that it works correctly.

The simulator that you wrote in this chapter gives you everything you need to see how those kinds of experiments work, so that we can plow ahead to use quantum mechanics and our qubits to do awesome stuff, armed with the knowledge that quantum mechanics really is how our universe works.

4.5 Summary

In this chapter you learned:

- Simulate state preparation, operations, and measurement results for multiple qubits,
- Program a simulator for multiple qubits leveraging the QuTiP Python package,
- Recognize the proof quantum mechanics is consistent with our observations of the universe by simulating experimental results.



Teleportation and entanglement: Moving quantum data around

This chapter covers:

- Implement a quantum program to move data around a quantum computer using classical and quantum control
- Recognize a new way of visualizing single qubit operations called the Bloch sphere
- Predict the output of two-qubit operations, and Pauli operations

5.1 Moving quantum data

Just as with classical computing, sometimes in a quantum computer you have some data *here* that you would very much appreciate being somewhere over *there*. Classically, this is an easy problem to solve by copying data around, but as we saw in Chapters 3 and 4, the No-Cloning Theorem means we in general **can't** copy data stored in qubits.

Moving data classically

In some parts of classical computing, we run into the same problem of not being able to copy information for very different reasons. Copying data in a multi-threaded application can introduce subtle race conditions, while performance considerations can prompt us to reduce the amount that we copy data. The solution embraced by many classical languages (e.g.: C++11 and Rust) is to instead focus on *moving* data. Thinking in terms of moving data is helpful in quantum computing, though we'll implement moves in a very different way.

So what can you do if you want to move data around in a quantum device? Thankfully,

there's a number of different ways to move quantum data instead of copying it. In this chapter, we'll see a few of these approaches and will add the last couple features that we need to our simulator to implement them. Let's get to sharing quantum information then!

Suppose Eve has some qubits that encode some data she'd like to share with you.

Hey player! I have some quantum information I want to share with you, can I send it over to you?

-- Eve

Here, Eve is referring to the `swap` instruction — it's a bit different than the instructions we've seen so far in that it operates on *two* qubits at once. By contrast, every operation that you've seen so far only operates on one qubit at a time.

Looking at what `swap` does, the name is pretty descriptive, because it literally swaps the state of two qubits in the same register. For example, say you had two qubits in the $|01\rangle$ state. If you use the `swap` instruction on both qubits, the result would be your register is now in the $|10\rangle$ state. Let's look at [5.1](#) for an example of using the `swap` matrix built-in to QuTiP.

Listing 5.1. Using the built in swap function in QuTiP, and an example of using it on the $|0\rangle$ to get the $|0\rangle$ state.

```
>>> import qutip as qt
>>> ket_0 = qt.basis(2, 0)
>>> ket_plus = qt.hadamard_transform() * ket_0
>>> initial_state = qt.tensor(ket_plus, ket_0)
>>> initial_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678] ①
 [0.          ]
[0.70710678]
[0.          ]]
>>> swap_matrix = qt.swap()
>>> swap_matrix * initial_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678] ②
 [0.70710678]
[0.          ]
[0.          ]]
>>> qt.tensor(ket_0, ket_plus)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678] ③
 [0.70710678]
[0.          ]
[0.          ]]
```

- ➊ We'll start by using `qt.basis`, `qt.hadamard_transform`, and `qt.tensor` together to define a variable for old friend from last chapter, the state vector $|+0\rangle$. This state makes a nice example for seeing what the swap instruction does.
- ➋ As we saw in Chapter 4, this state has equal amplitudes on the $|00\rangle$ and $|10\rangle$ computational basis states.
- ➌ We can get a copy of the unitary matrix for the swap instruction by calling `qt.swap`.
- ➍ The same way that we've simulated single-qubit operations, we can multiply our state by the unitary matrix for swap to find the state of our two-qubit register after applying a swap instruction.
- ➎ When we do so, we see that we end up in a superposition between $|00\rangle$ and $|01\rangle$ instead of between $|00\rangle$ and $|10\rangle$.
- ➏ You can quickly check that the result of using the swap instruction on a register of two qubits that start off in the $|+0\rangle$ state is $|0+\rangle$.

Looking at [5.1](#), you can see that the swap instruction did pretty much what its name suggests. In particular, swap took two qubits that started off in the state $|+0\rangle$ to the $|0+\rangle$ state. More generally, you can read off what the swap instruction does by looking at the unitary matrix we used to simulate it above (see [5.2](#)).

Listing 5.2. Looking at the unitary matrix for the swap instruction

```
>>> import qutip as qt
>>> qt.swap()
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
➊
Qobj data =
[[1. 0. 0. 0.]
➋
[0. 0. 1. 0.]
➌
[0. 1. 0. 0.]
[0. 0. 0. 1.]]
➍
```

- ➊ Note that the unitary matrix we use to simulate the swap instruction is a 4×4 matrix. This is because it acts on two-qubit states, meaning that it has to define what happens to each of the four possible computational basis states that a two-qubit register can be in.
- ➋ Each column of this unitary matrix tells us what happens to one of the computational basis states; for instance, the first column tells us that the $|00\rangle$ state is mapped to the vector $[[1], [0], [0], [0]]$, which we recognize as $|00\rangle$. That is, the swap instruction does nothing to qubits that start off in the $|00\rangle$ state.
- ➌ On the other hand, we can see that the $|01\rangle$ and $|10\rangle$ states are swapped by the swap instruction.
- ➍ Finally, the swap instruction also leaves $|11\rangle$ alone.

IMPORTANT

The unitary matrix for the swap instruction **cannot** be written as the tensor product of any two single-qubit unitary matrices. That is, you can't understand what swap does by considering one qubit at a time—you need to work out what it does to state of the pair of qubits that the swap instruction acts on.

As shown in [5.1](#), you can see what swap does in general, no matter what state our two qubits start off in.

Figure 5.1. The two qubit operation swap exchanges the states of two qubits in a register.

Since the swap instruction acts on two-qubit register, the state of each such register is represented by a four-element vector. Just as you might use x or y as variables in algebra, we tend to use Greek letters like α and β to stand for arbitrary amplitudes. Here, α , β , γ , and δ are the amplitudes for each of the $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ computational basis states, respectively.

$$U_{\text{SWAP}} \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} = \begin{bmatrix} \alpha \\ \gamma \\ \beta \\ \delta \end{bmatrix}$$

After running the swap instruction, the amplitudes for $|01\rangle$ and $|10\rangle$ are swapped, while the amplitudes for $|00\rangle$ and $|11\rangle$ stay the same, since swapping $0 \rightleftharpoons 0$ or $1 \rightleftharpoons 1$ doesn't do anything.

We can simulate how the swap instruction changes the state of a register by multiplying the state of that register by a *unitary matrix*. As always, this matrix isn't the swap instruction, but a tool we can use to simulate swap.

Remember that in Chapter 2, you saw that a unitary matrix is a lot like a truth table. That is, unitary matrices like what we get back from `qt.swap` are useful in that they help us simulate what the swap instruction does. Just as a classical adder isn't its truth table, though, it's helpful to remember that these unitary matrices aren't quantum programs, but tools that we use to simulate how quantum programs work.

Exercise 5.1: swap the second and third qubits in a register.

Say you had a register with 3 qubits, in the state $|01+\rangle$. Using your simulator, prepare this state, and then swap the second and third qubits so your register is in the $|0+\rangle$ state.

Hint: since nothing will happen to the first qubit, make sure to tensor an identity matrix to `qt.swap` to build up the correct operation for your register.

At this point, though, Eve is positively *dying* waiting to send us her qubits. Let's go on and add what we need to our simulator, so as to not keep her waiting any longer!

5.1.1 Swapping out our simulator

The simulator you worked on in Chapter 4 needs just a couple of tweaks to be able to use two qubit operations like `swap`. The changes we will need to make are:

- modify `_apply` to work with two qubit operations,
- add the `swap` and other two qubit instructions, and
- add the rest of the single qubit rotation instructions.

As you saw in Chapter 4, if you have a matrix that acts on a single-qubit register, you can use QuTiP to apply that to a register with an arbitrary number of qubits by using the `gate_expand_1toN` function. This makes sure to tensor in the identity operators everywhere else *but* the qubits you're working with.

In the same way, you can call QuTiP's `gate_expand_2toN` function to turn two-qubit unitary matrices into matrices that you can use to simulate how two-qubit operations like `swap` transform the state of a whole register. Let's go on and add that into our simulator now:

TIP

We've made a couple small changes to the code in this Chapter to help make printed outputs look a little nicer. All of these changes, along with all the samples for this and other chapters, are on the GitHub repo for this book at github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

Listing 5.3. Using `gate_expand_2toN` to apply two-qubit unitary matrices to register states.

```
def _apply(self, unitary : qt.Qobj, ids : List[int]):
    if len(ids) == 1:
        matrix = qt.circuit.gate_expand_1toN(unitary,
                                              self.capacity, ids[0])
    elif len(ids) == 2:
        matrix = qt.circuit.gate_expand_2toN(unitary,
                                              self.capacity, *ids)
    else:
        raise ValueError("Only one- or two-qubit unitary matrices supported.")

    self.register_state = matrix * self.register_state
```

①
②

- ① To simulate two-qubit operations, we need two indices for qubits in the register; one for each qubit that our instruction acts on.

- ② The call to `gate_expand_2toN` looks very similar to our call to `gate_expand_1toN`, except that we pass a 4×4 matrix instead of a $2 \times$ matrix.

We saw that QuTiP provides the function `swap` to give us a copy of the unitary matrix that simulates the `swap` instruction. This can be used to pretty quickly add the `swap` instruction to your simulator using the changes you made to `Simulator._apply`; see [5.4](#).

Listing 5.4. Adding a swap instruction to your simulator.

```
def swap(self, target : Qubit) -> None:
    self.parent._apply(
        qt.swap(),
        [self.qubit_id, target.qubit_id]
    )
```

1
2

- ① To get the 4×4 unitary matrix we need to pass to `_apply`, we just use the `qt.swap` function we've seen a few times so far in this Chapter.
- ② Next, we just need to make sure we pass the indices for both qubits that we want to swap between. This will make it so that `gate_expand_2toN` correctly applies the unitary matrix for our new `swap` instruction to the state of an entire register.

While we're working with the simulator anyway, let's add one more instruction to let us print out the simulator's state more easily without having to access its internals, see [5.5](#).

Listing 5.5. Adding a dump instruction to the simulator allows you to more conveniently print the state of the register.

```
def dump(self) -> None:
    print(self.register_state)
```

This way, you can ask the simulator to help you out in debugging out quantum programs, but in a way that can be safely stripped out for devices that don't support that (e.g.: actual quantum hardware).

TIP

Remember that a qubit is not a state, a state is just a convenient way of representing how the quantum system will behave.

With both these changes in place, you're now all set to use the `swap` instruction. Let's use it to repeat our experiment before, where we swapped two qubits starting off in the $|0+\rangle$ state to transform them into the $|+0\rangle$ state, see [5.6](#).

Listing 5.6. Test out your simulator's new swap instruction on the $|0+\rangle$ state.

```
>>> from simulator import Simulator
```

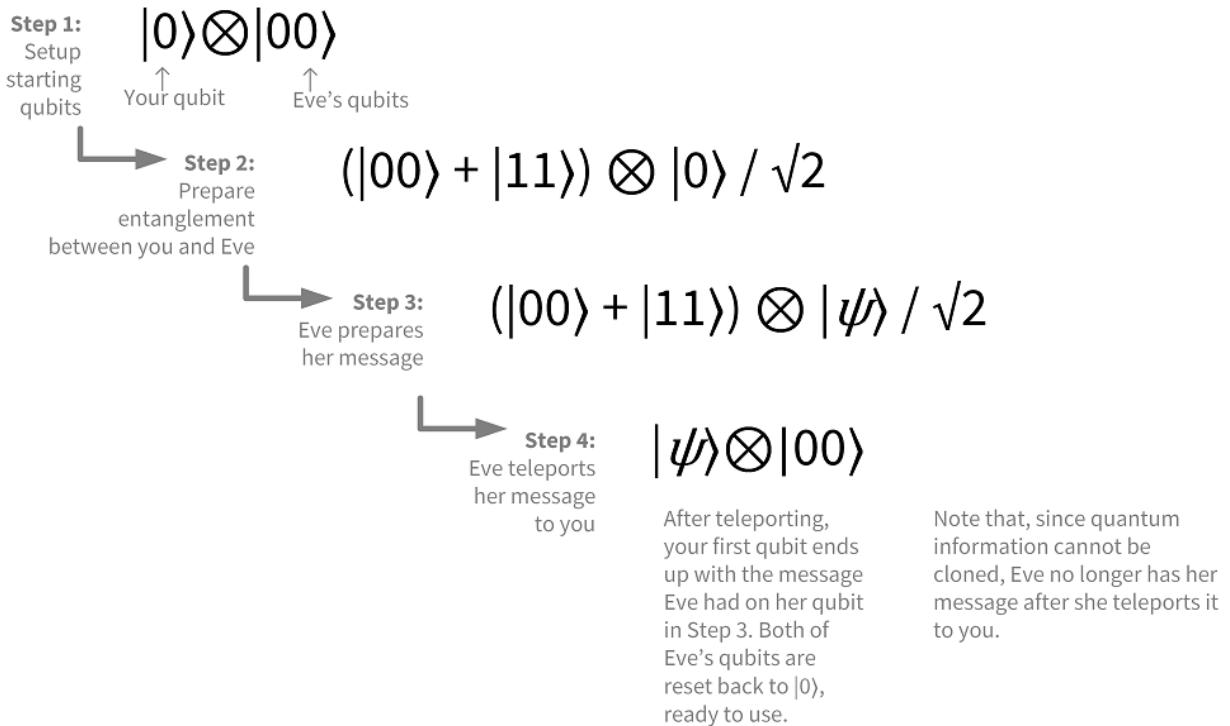
```
>>> sim = Simulator(capacity=2)
>>> with sim.using_register(n_qubits=2) as (you, eve):
...     eve.h()
...     sim.dump()
...     you.swap(eve)
...     sim.dump()
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket ❶
Qobj data =
[[0.70710678]
 [0.70710678]
[0.
 ]
[0.
 ]]
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket ❷
Qobj data =
[[0.70710678]
 [0.
 ]
[0.70710678]
[0.
 ]]

```

- ❶ Since we'll be working a lot with multiple-qubit registers in this Chapter, we've added a new convenience method that lets us allocate several qubits at once. See the code samples for this book for details.
- ❷ The first dump comes from our first call to `sim.dump`, and confirms that `eve.h()` prepared your qubits in the $|0+\rangle$ state.
- ❸ After calling `you.swap(eve)`, the next dump confirms for us that we ended up with the information Eve had prepared on her qubit, so that your qubit ends up in the $|+\rangle$ state, with Eve's qubit ending the way yours started: in the $|0\rangle$ state.

Great, you now have a way to share quantum data with Eve! Well, at least *as long as you're sharing a single quantum device*, so that you can apply the `swap` instruction to both your qubits at the same time.

What happens if we want to share quantum information *between* devices? Thankfully, quantum computing gives us a way for you to send each other your qubits by only communicating classical data, so long as you both start with some entanglement between your qubits. Like much in quantum computing, this technique is given a whimsical name, in this case, "quantum teleportation." Don't let the name fool you, however. When you get right down to it, teleportation uses what we learned in Chapter 4 to let you share quantum data in a useful way. [5.2](#) shows a list of the steps in a teleportation program.

Figure 5.2. The steps to the teleportation program.

What's really neat about teleportation is that, while you and Eve still need to do some two-qubit operations between your respective qubits, Eve can decide what data she wants to send you *after* you've done those operations. This means you could prepare the entangled qubits before you needed to exchange quantum data, and just use them as as needed.

Using the simulator that you've developed throughout the past several chapters, you might write up teleportation with a quantum program like that shown in [5.7](#).

Listing 5.7. A teleportation program in Python.

```
def teleport(msg : Qubit, here : Qubit, there : Qubit) -> None:
    here.h()
    here.cnot(there)

    msg.cnot(here)
    msg.h()

    if msg.measure(): there.z()
    if here.measure(): there.x()

    msg.reset()
```

```
here.reset()
```

You may notice a few new instructions in this program, though. In the rest of this Chapter, you'll see the other pieces you need to get up and running with quantum teleportation using your simulator.

5.1.2 What other two-qubit gates are there?

As you may guess, `swap` is not the only two-qubit operation. Indeed, as you can see from 5.7, in order to get teleportation working, we need to add another two-qubit instruction to your simulator, called `cnot`. The `cnot` instruction does something similar to `swap`, except that it switches the $|10\rangle$ and the $|11\rangle$ computational basis states instead of the $|01\rangle$ and $|10\rangle$ states. Another way to think of this is that `cnot` flips the second qubit *controlled* on the state of the first qubit being $|1\rangle$. Indeed, this is where the name `cnot` comes from: it's shorthand for "controlled-NOT."

TIP

We often call the first qubit passed to a `cnot` instruction the *control qubit*, and the second qubit the *target qubit*. As we will see in Chapter 7, though, there is a bit of subtlety to these names.

Let's jump right in and see how the `cnot` instruction works by applying it to that lovely example, the $|+0\rangle$ state.

Listing 5.8. cnot-intro

```
>>> import qutip as qt
>>> ket_0 = qt.basis(2, 0)
>>> ket_plus = qt.hadamard_transform() * ket_0
>>> initial_state = qt.tensor(ket_plus, ket_0)
>>> qt.cnot() * initial_state
❶
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
❷
Qobj data =
[[0.70710678]
 [0.        ]
 [0.        ]
 [0.70710678]]
>>> qt.cnot()
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
❸
Qobj data =
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]]
```

- ❶ QuTiP provides the unitary matrix for the `cnot` instruction as the `qt.cnot` function.
- ❷ Applying the `cnot` instruction to two qubits that start off in the $|+0\rangle = (|00\rangle + |10\rangle) / \sqrt{2}$, we see that `cnot` transformed the $|10\rangle$ part of our input state to $|11\rangle$ and did nothing to the first qubit in the $|1\rangle$ state, leaving our qubits in the state $(|00\rangle + |11\rangle) / \sqrt{2}$.

- ③ Looking at the matrix for cnot, we can see that it maps the $|10\rangle$ computational basis state to $|11\rangle$ and vice versa, just as we expected from the description.

IMPORTANT

The **cnot** instruction is **not** the same as **if** statements in classical programming languages, in that a cnot instruction preserves superposition. If we wanted to use an **if** statement, we'd have to measure the control qubit, such that it would collapse any superposition on the control qubit. We'll actually use both cnot instructions and **if** statements conditioned on measurement results when we write out our teleportation program at the end of this Chapter — both can be useful!

In Chapters 8 and 9, we'll see more about how controlled operations differ from **if** statements.

In 5.3, we can see how the **cnot** instruction acts on two-qubit states in general. For now, though, we recognize the output state that we got in 5.8 by acting **cnot** on two qubits in the $|+0\rangle$ state as the *entangled* state we needed to play the CHSH game in Chapter 4, $(|00\rangle + |11\rangle)/\sqrt{2}$.

Figure 5.3. The two qubit operation cnot exchanges the states of two qubits in a register.

The state of each register acted on by a **cnot** instruction is represented by a four-element vector. These elements tell us the amplitudes for each of the $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ computational basis states, respectively.

$$U_{\text{CNOT}} \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ \delta \\ \gamma \end{bmatrix}$$

After running the **cnot** instruction, the amplitudes for $|10\rangle$ and $|11\rangle$ are swapped, while the amplitudes for $|00\rangle$ and $|01\rangle$ stay the same.

Just as we simulated swap by using a unitary matrix, we can also simulate the controlled-NOT (**cnot**) instruction by using a unitary matrix.

We can think of this instruction as doing nothing when the first qubit is in the $|0\rangle$ state, and as flipping the second qubit when the first one is in $|1\rangle$. Unlike a classical **if** statement, which requires measurement, the **cnot** instruction is *linear*, and thus preserves superposition.

This means we have everything we need to write a quantum program that entangles two qubits that start off in the $|00\rangle$ state. All we need to do is add the **cnot** instruction to our simulator, the same way as we added **swap** above.

```

def cnot(self, target : Qubit) -> None:
    self.parent._apply(
        qt.cnot(),
        [self.qubit_id, target.qubit_id]
)

```

Now we can write a program to prepare two qubits in an entangled pair.

```

>>> from simulator import Simulator
>>> sim = Simulator(capacity=2)
>>> with sim.using_register(2) as (you, eve):
...     eve.h()
...     eve.cnot(you)
...     sim.dump()
...
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678]
 [0.
 [0.
 [0.70710678]]

```

At this point, it's helpful to pause a moment (sorry, Eve!) and reflect on what you just did. In Chapter 4, when you simulated playing the CHSH game with Eve, you had to "cheat" by assuming that you and Eve could have access to two qubits that magically start off in the entangled state $(|00\rangle + |11\rangle) / \sqrt{2}$. Now, though, we see exactly how you and Eve can prepare that entanglement by running another quantum program before playing CHSH. The `h` instruction prepares the superposition that you need, while the new `cnot` instruction allows you to prepare entanglement with Eve. This entanglement "shares" that superposition across your two qubits. (Sharing is caring, after all.)

Just like preparing entanglement between you and Eve was how you needed to get ready for the CHSH game in Chapter 4, it will be the first step you need for Eve to teleport her quantum data to you. This makes `cnot` a very important instruction for us going forward.

Getting back to Eve, though, the next step that you'll need in order for her to teleport her data to you is that you'll need to do one of four different single-qubit operations to *decode* the quantum data that she sends you (recall [5.2](#)), so let's look at those next.

5.2 All the single (qubit) rotations

The last thing you will need to program quantum teleportation is to apply a correction based on some classical data that Eve sends you. To do so, you'll need couple of new single-qubit instructions. For that, it's helpful to revisit the pictures we've been using to depict quantum instructions as rotations, because we may have been cheating a little . We have depicted our qubits so far as any position on a circle, but in reality we are missing a dimension for our model of a qubit. The state of a *single* qubit is represented any point on the surface of a sphere.

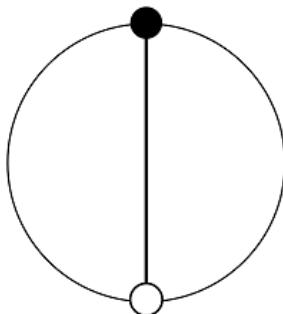
IMPORTANT**Single qubits only!**

This (and the previous) way of visualizing the state of a qubit *only* works if that qubit is not entangled with any other qubits. Another way of saying this is we cannot easily visualize a multi-qubit state. Even trying to visualize the state of a two-qubit register with entanglement would take drawing pictures in 7 dimensions. While "7D" might be nice for advertising your ride at Niagara Falls, it is much harder draw useful pictures that way.

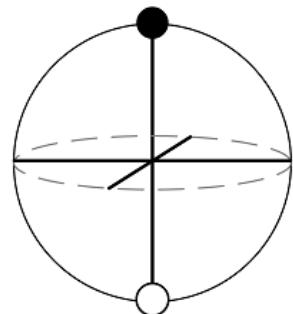
The circle you are familiar with was really just a slice through the sphere, and all the rotations we were doing resulted in states still on that circle.

Figure 5.4. Comparison of our previous model for a qubit and the Bloch sphere, a better general model for a single qubit.

In Chapters 2, 3, and 4, you saw that the state of a single-qubit can be thought of as a point on a circle.



In this Chapter, we'll see that single-qubit states can also be rotated "out of the page," giving you an entire sphere instead of just the circle.



You might have inferred from the fact that we had shown the Z -axis and the X -axis that the Y -axis was probably hiding somewhere!

You may recall that when we first introduced the vector representation of a qubit state in Chapter 2, that the amplitudes in each vector were *complex numbers*. In the rest of this Chapter, we'll see that when you use rotation instructions to transform the state of a single qubit, we'll get complex numbers in general. Complex numbers are an incredibly useful tool for keeping track of rotations, and thus play a large role in quantum computing. Primarily, they help us understand the angles and phases between different quantum states. Don't worry if you're a little rusty with complex numbers, as you'll get plenty of chance practice with them throughout the rest of the book. If you need a refresher, please check out Appendix B.

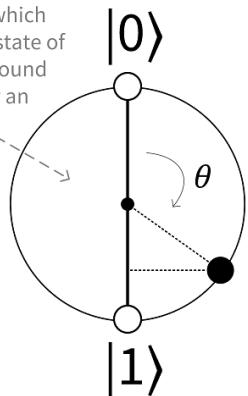
5.2.1 Relating rotations to coordinates: The Pauli operations

Let's do a quick review of a couple of the single qubit operations we have seen so

far, x and ry.

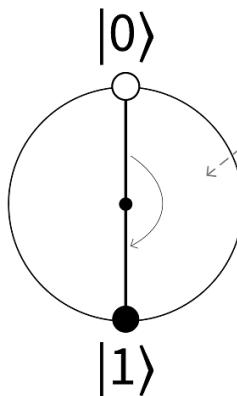
Figure 5.5. Illustrations of what x and ry do to a qubit, both operations you have seen in previous chapters.

Recall the ry operation, which rotates the state of our qubit around the circle by an angle θ .



ry operation

You also saw the x operation, which rotates from the $|0\rangle$ to the $|1\rangle$ state and vice-versa, giving you the quantum analogue to the classical NOT gate.



x operation

Now that we know the state of our qubit actually can be rotated on the surface of a sphere, what other rotations can help us rotate the state out of the plane? We can add a rotation about the line between the $|+\rangle$ and $|-\rangle$ states. This line is conventionally called the X—axis to distinguish it from the Z -axis that connects the $|0\rangle$ and $|1\rangle$ states. The rx function in QuTiP gives us a Qobj encapsulating the rotation matrix for a X—axis rotation:

Listing 5.9. Using the QuTiP built-in function qt.sigmax.

```
>>> qt.rx(np.pi).tidyup()          ①
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = False
Qobj data =
[[ 0.+0.j  0.-1.j]               ②
 [ 0.-1.j  0.+0.j]]
```

- ➊ Due to how floating-point numbers work in classical computers, sometimes simulations of rotations can result in very small numbers like 10^{-17} where we would expect 0. QuTiP Qobj instances have a method tidyup to help make matrices more readable when this happens.
- ➋ Up to a coefficient of $-i$ (written in Python as $-1j$), rotating by 180° about the X - axis results in the x (NOT) instruction that we first saw in Chapter 2.

TIP

In Python, the complex number i is represented by $1.0j$ which is 1 times j which is sometimes what the imaginary number i is called in other fields.

This snippet illustrates something very important: the X operation is precisely what we get by rotating around the X axis by an angle of 180° (π).

NOTE**Global phase**

As noted in the callouts for the above snippet, you can check that `qt.rz(np.pi)` is actually off by a factor of $-i$ from `qt.sigmax()`. That factor is an example of something called a **global phase**. As we will see shortly, global phases *cannot* affect the results of measurement. Thus, `qt.rz(np.pi)` and `qt.sigmax()` are different unitary matrices that represent the same operation. We'll get more practice with global and local phases in Chapter 7.

By analogy, we call rotating by 180° about the Z -axis a z operation. In chapter 3, QuTiP provided you with the `qt.sigmax` function which allow you to simulate the x instruction. Similarly, `qt.sigmaz` provides us with the unitary matrices we need to simulate the z instructions. See [5.10](#) for an example of using `qt.sigmaz`.

Listing 5.10. Using the QuTiP built-in functions `qt.rz` and `qt.sigmaz`.

```
>>> 1j * qt.rz(np.pi).tidyup()           ①
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]
>>> qt.sigmaz()                         ②
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]
```

- ➊ Note that we've just included the coefficient (that is, the global phase) of $-i$ by multiplying by i right away; this works since $-i \times i = -(-1) = 1$. Cancelling out the global phase in this way will make it easier to read the output below.
- ➋ As promised, up to the coefficient of $-i$, the z instruction applies a 180° rotation about the Z -axis.

In the same way that the X operation flips between $|0\rangle$ and $|1\rangle$ while leaving $|+\rangle$ and $|-\rangle$ alone, the z operation flips between $|+\rangle$ and $|-\rangle$ while leaving $|0\rangle$ and $|1\rangle$ alone.

[5.1](#) shows a truth table like the ones we have made before for the Hadamard operation in Chapter 2. By looking at the truth table, you can confirm that, for any input state if you do a Z operation twice on it you will be back to where you started. Another way to say that is, Z squares to the identity operation 1 , in the same way that $X^2 = 1$.

Table 5.1. Table Representing the z instruction as a table.

Input state	Output state
$ 0\rangle$	$ 0\rangle$
$ 1\rangle$	$- 1\rangle$
$ +\rangle$	$ -\rangle$
$ -\rangle$	$- +\rangle$

NOTE

We listed four rows in 5.1, but we only need two rows to completely specify how Z acts for any input. The other two rows serve to emphasize that we can choose between defining Z by its action on $|0\rangle$ and $|1\rangle$ or by its action on $|+\rangle$ and $|-\rangle$.

Exercise 5.2: Practice using rz and z

Suppose you prepare a qubit in the $|-\rangle$ state and apply a z rotation. If you measure along the $X -$ axis, what would you get? What would you measure if you apply two z rotations? If you had to implement those same two rotations with rz , what angles would you use?

We can define one more rotation in the same way, namely the rotation about an axis coming "out of the page." This axis connects the states $(|0\rangle + i|1\rangle) / \sqrt{2} = R_{x(\pi)}|0\rangle$ and $(|0\rangle - i|1\rangle) / \sqrt{2} = R_{x(\pi)}|1\rangle$, and is conventionally called the Y -axis. A 180° rotation about the Y -axis both flips bit labels ($|0\rangle \leftrightarrow |1\rangle$) and phases ($|+\rangle \leftrightarrow |-\rangle$), but leaves alone the two states along the Y -axis.

Exercise 5.3: Truth table for sigmay.

Use the `qt.sigmay()` function to make a table similar to 5.1, but for the y instruction.

Definition 5.1: Pauli matrices

Together, the three matrices X , Y , and Z representing the x , y , and z operations are called the Pauli matrices in honor of physicist Wolfgang Pauli. The identity matrix 1 is sometimes included as well, representing the "do nothing" or identity operation.

Playing rock-paper-scissors with the Pauli matrices

The Pauli matrices have a number of useful properties that we'll make use of throughout the rest of the book. Many of these properties make it easy to work out different equations involving the Pauli operators.

For example, if you multiply X and Y together, you get iZ , but if you multiply YX instead, you get $-Z$ back:

```
>>> qt.sigmax() * qt.sigmay()
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = False
Qobj data =
[[0.+1.j 0.+0.j]
 [0.+0.j 0.-1.j]]
>>> qt.sigmay() * qt.sigmax()
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = False
Qobj data =
[[0.-1.j 0.+0.j]
 [0.+0.j 0.+1.j]]
```

Similarly, $YZ = iX$ and $ZX = iY$, but $ZY = -iX$ and $XZ = -iY$. One way to remember this is as thinking of X , Y , and Z as playing a little game of rock–paper–scissors: X "beats" Y , Y "beats" Z , and Z "beats" X in turn.

We can think of these matrices as establishing a kind of coordinate system for qubit states, called the *Bloch sphere*. As shown in 2.27, the X and Z -axes form the circle that you've seen in the book thus far, while the Y -axis comes out of the page.

NOTE

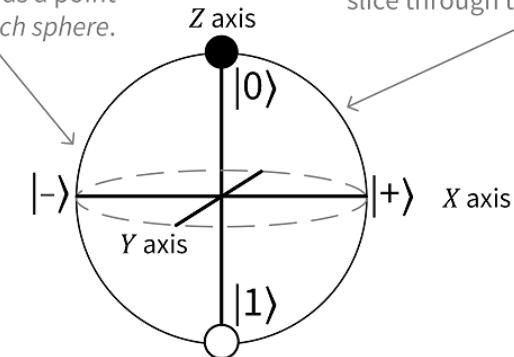
Describing states with Pauli measurements

Any single-qubit state can be entirely specified up to a global phase by the measurement probabilities for X , Y , and Z measurements. That is, if I tell you the probability of getting a "1" outcome for each of the three Pauli measurements that I could perform, then you can use that information to write down a state vector that is identical to mine, up to global phase. This makes the analogy to points in three dimensions a very good analogy for thinking about single-qubit states.

Figure 5.6. The Bloch sphere, in all its spherical glory.

The state of the qubit now is represented as a point anywhere on the *Bloch sphere*.

The circle that we are used to from Chapters 2 through 4 is a slice through the Bloch sphere.

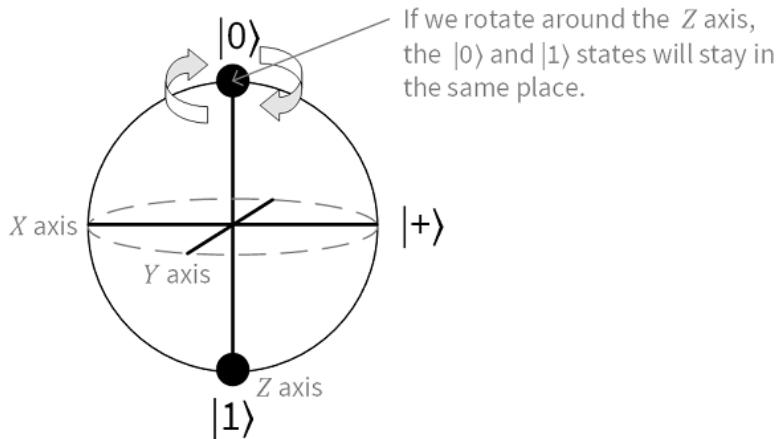


NOTE**The *i*s have it**

The states at the ends of the *Y*-axis are usually labeled $|i\rangle$ and $|{-i}\rangle$, but are not often used on their own. We will just stick to the labeled states we were using before: $|0\rangle$, $|1\rangle$, $|+\rangle$, and $|-\rangle$.

With this picture in mind, it's easier to see why some rotations don't affect the results of measurements. For instance, as we illustrate in [5.7](#), the Bloch sphere picture helps us understand what happens if we rotate $|0\rangle$ about the *Z*-axis.

Figure 5.7. The Bloch sphere illustrating how an r_z rotation leaves the $|0\rangle$ state unchanged.



In the same way that the North Pole on a globe stays in the same spot no matter how much we spin the globe, if we rotate a state about an axis parallel to that state, there is no observable effect on our qubit. We can also see this effect come out of the math as well, see [5.11](#).

Listing 5.11. An example showing how the $|0\rangle$ state is unaffected by r_z rotations.

```
>>> ket0 = qt.basis(2, 0)          ❶
>>> ket_psi = qt.rz(np.pi / 3) * ket0 ❷
>>> ket_psi
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[ 0.8660254-0.5j]                      ❸
 [ 0.0000000+0.j ]]
>>> bra0 = ket0.dag()                ❹
>>> bra0
Quantum object: dims = [[1], [2]], shape = (1, 2), type = bra
Qobj data =
[[ 1.  0.]]                                ❺
>>> np.abs((bra0 * ket_psi)[0, 0]) ** 2 ❻
1.0
```

- ❶ We start by defining a variable to represent the state $|0\rangle$.

- 2 Next, we introduce a new state $|\psi\rangle$ that's a 60° ($\pi/3$ in radians) rotation of $|0\rangle$ about the Z axis.
- 3 The resulting state is $|\psi\rangle = [\cos(60^\circ/2) - i \sin(60^\circ/2)] |0\rangle = [1/2 + i\sqrt{3}/2] |0\rangle$.
- 4 To check that this rotation doesn't do anything to our measurement results, we can check what we get from Born's rule. Let's start by writing down a measurement $\langle 0| = |0\rangle^+$. Recall that in QuTiP, we write out the "dagger" operator $^+$ by calling the `.dag` method of `Qobj` instances.
- 5 Since $|0\rangle$ is represented by a column vector, taking the conjugate transpose to get $\langle 0|$ gives us a row vector.
- 6 Taking the inner product $\langle 0 | \psi\rangle$, we can compute Born's rule $\text{Pr}(0 | \psi) = |\langle 0 | \psi\rangle|^2$. Note that we need to index by $[0, 0]$, since QuTiP represents the inner product of $|0\rangle$ with $|\psi\rangle$ as a 1×1 matrix.
- 7 As before, we note that the probability of observing a "0" when measuring along the Z axis hasn't changed.

TIP

When writing down states, $|\psi\rangle$ is often used as just an arbitrary name, similar to how x is often used to represent an arbitrary variable in algebra.

Exercise 5.4: verify that applying `rz` doesn't change $|0\rangle$.

We've only checked that one measurement probability is still the same, but maybe the probabilities have changed for X or Y measurements. To be fully check that the global phase doesn't change anything, prepare the same state and rotation as in 5.11 and check that the probabilities of measuring the state along the X or Y axis aren't changed by applying an `rz` instruction.

In general, you can always multiply a state by a complex number whose absolute value is 1 without changing the probabilities of any measurement. Any complex number $z = a + bi$ can be written as $z = re^{i\theta}$ for real numbers r and θ , where r is the absolute value of z and where θ is an angle. When $r = 1$, we have a number of the form $e^{i\theta}$, which we call a **phase**. We then say that multiplying the state by a phase applies a *global phase* to that state.

IMPORTANT

No global phase can ever be detected by any measurement.

The states $|\psi\rangle$ and $e^{i\theta}|\psi\rangle$ are in every conceivable way two different ways of describing the exact same state. There is no measurement that one can do *even in principle* to learn about global phases. On the other hand, we've seen that we can tell apart states like $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ and $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$ that differ only in the *local* phase of the $|1\rangle$ computational basis state.

Taking a step back, let's summarize what you've learned about the x , y , and z instructions so far, and about the Pauli matrices we use to simulate those instructions. We've seen that the x instruction flips us between $|0\rangle$ and $|1\rangle$, while the z instruction flips us between the $|+\rangle$ and $|-\rangle$ states. Put differently, the x instruction flips bits, while the z instruction flips phases.

Looking at the the Bloch sphere, you can see that rotating about the Y -axis should do both of these. You can also see this by using that $Y = -iXZ$, as is straightforward to check with QuTiP. We summarize what each Pauli instruction does in [5.2](#).

Table 5.2. Table Pauli matrices as bit and phase flips.

Instruction	Pauli matrix	Flips bits ($ 0\rangle \leftrightarrow 1\rangle$)?	Flips phases ($ +\rangle \leftrightarrow -\rangle$)?
(no instruction)	$\mathbb{1}$	No	No
x	X	Yes	No
y	Y	Yes	Yes
z	Z	No	Yes

Exercise 5.5: Hmm, that rings a Bell...

The $(|00\rangle + |11\rangle) / \sqrt{2}$ state that we've seen a few times now isn't the only example of an entangled state. In fact, if you pick a two-qubit state at random, it is almost certainly going to be entangled. Just as the computational basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ is a particularly useful set of unentangled states, there's a set of four particular entangled states known as the *Bell basis* after physicist John Stewart Bell.

Table 5.3. The four Bell states

Name	Expansion in computational basis
$ \beta_{00}\rangle$	$(00\rangle + 11\rangle) / \sqrt{2}$
$ \beta_{01}\rangle$	$(00\rangle - 11\rangle) / \sqrt{2}$
$ \beta_{10}\rangle$	$(01\rangle + 10\rangle) / \sqrt{2}$
$ \beta_{11}\rangle$	$(01\rangle - 10\rangle) / \sqrt{2}$

Using what you've learned about the `cnot` instruction and the Pauli instructions (x, y, and z), write programs to prepare each of the four Bell states in the table above.

Hint: [5.2](#) should be very helpful in this exercise.

We finish our discussion of single-qubit operations by adding instructions to our Qubit interface and simulator for the x, Y and Z operations.

Listing 5.12. simulator.py

```
def rx(self, theta : float) -> None:
    self.parent._apply(qt.rx(theta), [self.qubit_id]) ①

def ry(self, theta : float) -> None:
    self.parent._apply(qt.ry(theta), [self.qubit_id])

def rz(self, theta : float) -> None:
    self.parent._apply(qt.rz(theta), [self.qubit_id])

def x(self) -> None:
    self.parent._apply(qt.sigmax(), [self.qubit_id]) ②
```

```

def y(self) -> None:
    self.parent._apply(qt.sigmay(), [self.qubit_id])

def z(self) -> None:
    self.parent._apply(qt.sigmax(), [self.qubit_id])

```

- ➊ We can implement the rotation instructions `rx`, `ry`, and `rz` using the corresponding QuTiP functions `qt.rx`, `qt.ry`, and `qt.rz` to get copies of the unitary matrices we need to simulate each instruction.
- ➋ QuTiP uses the notation σ_x instead of X for the Pauli matrices. Using this notation, the function `sigmax()` returns a new Qobj representing the Pauli matrix X . This way, we can implement the `x`, `y`, and `z` instructions corresponding to each Pauli matrix.

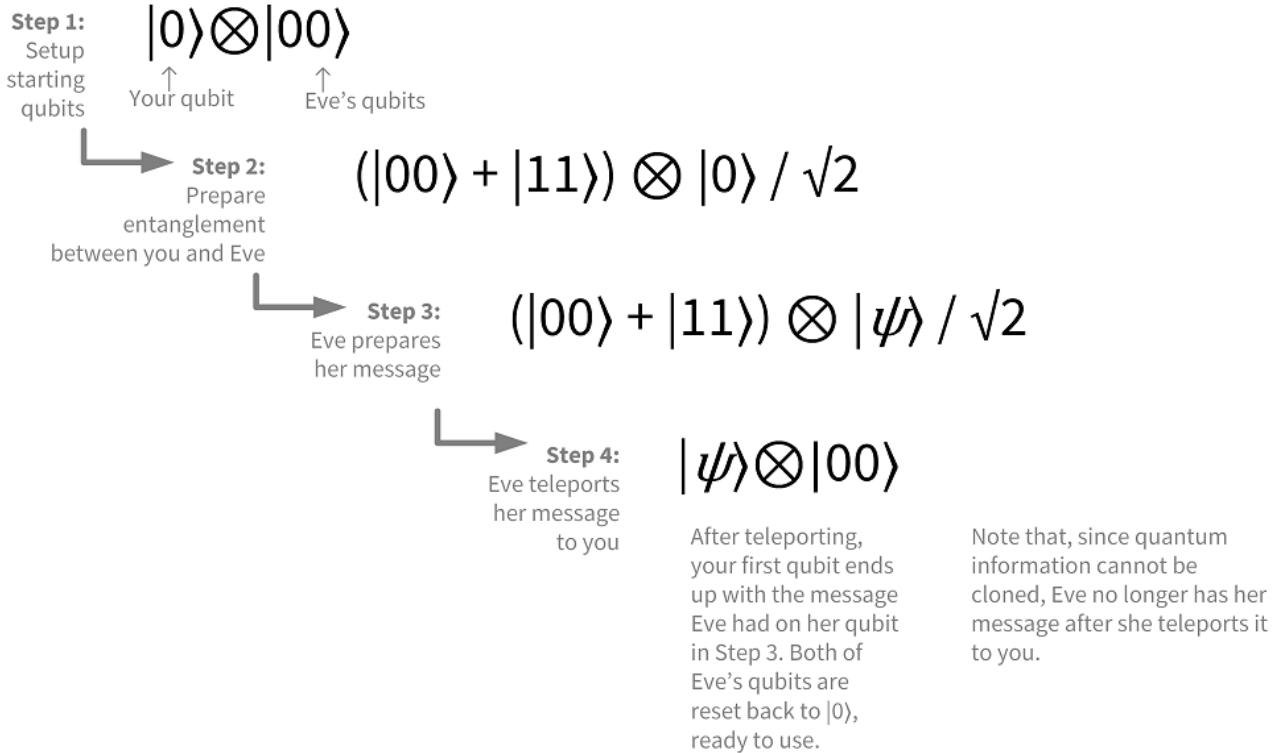
No one can be told what the matrix is. You have to see it for yourself.

We've talked a lot about matrices thus far in Part I. A lot. It's tempting to say that quantum programming is all about matrices, and that qubits are really just vectors. In reality, though, matrices are how we **simulate** what a quantum device does. We'll see more in Part II, but quantum programs don't manipulate matrices and vectors at all—they manipulate classical data such as what instructions to send to a quantum device, and what to do with the data we get back from devices. For instance, if we have an instruction that we run on a device, there's no simple way to see what matrix we should use to simulate that instruction—rather, you have to reconstruct that matrix from many repeated measurements using a technique called *process tomography*.

When you write down a matrix, whether in code or on a piece of paper, you're implicitly simulating a quantum system. If that really bakes your noodle, don't worry, this will make a lot more sense as you go through the rest of the book.

5.3 Teleportation

OK, now we have everything we need to write out what teleportation looks like as a quantum program. As a quick review, [5.8](#) is what we want this program to do.

Figure 5.8. Recall the steps to the teleportation program.

We will assume you can prepare some entangled qubits while they are in the same device, and that you and Eve have a means of classical communication that you can use to signal the correct correction to use.

We can now use the features we added to our simulator in this chapter to implement the teleportation program, see [5.13](#).

Listing 5.13. Quantum teleportation, all in just a few lines of Python.

```
from interface import QuantumDevice, Qubit
from simulator import Simulator

def teleport(msg : Qubit, here : Qubit, there : Qubit) -> None:
    here.h()
    here.cnot(there)

    # ...
    msg.cnot(here)
    msg.h()

    if msg.measure(): there.z()
```

```
if here.measure(): there.x()
msg.reset()
here.reset()
```

7

- ➊ Your teleport function will take two qubits as input: the qubit that you want to move (`msg`), and where you want it to be moved to (`there`). You'll also need one temporary qubit, which we call `here`. We'll presume by convention that both `here` and `there` start in the $|0\rangle$ state.
- ➋ We need to start off with some entanglement between `here` and `there`. We can use our old friend, the `h` instruction, together with our *new* friend, the `cnot` instruction.
- ➌ This is the only instruction that we'll use in this program that needs to act on both `here` and `there`; after running this, you can send Eve your qubit, and both of you can run the rest of the program with only classical communication.
- ➍ At this point in the program, `here` and `there` are in the $(|00\rangle + |11\rangle) / \sqrt{2}$ state that we first saw in Chapter 4.
- ➎ The next step is to run the program we used to prepare the $(|00\rangle + |11\rangle) / \sqrt{2}$ state backwards, but on the `msg` and `here` qubits that live entirely on your device instead. We can think of running a preparation backwards as a kind of measurement, such that these steps set us up to measure the quantum message you're trying to send Eve in an entangled basis.
- ➏ When we actually do that measurement, we get *classical* data back out that we can use to send Eve. Once she has that data, she can use the `x` and `z` instructions to decode the quantum message.
- ➐ Now that you're done with your qubits, it's good to put each of them back into $|0\rangle$ so that they're ready to be used again. This doesn't affect the state of `there`, though, as you've only reset your qubits, not the one you gave to Eve!

IMPORTANT***c what we did there?***

If we didn't need to send Eve our classical measurement results as a part of teleportation, then we could use teleportation to send both classical and quantum data faster than the speed of light. Just like you couldn't communicate with Eve when you played the CHSH game in the previous chapter, the speed of light means that you need to communicate with Eve classically in order to use entanglement to send quantum data. In both cases, entanglement can help us communicate, but it doesn't let you communicate all on its own: you always need some other kind of communication as well.

To see that this actually works, Eve can prepare something on your qubit, send it to Eve, and then she can undo your preparation on her qubit. Thus far, the messages you and Eve have been sending have been classical, but here the message is *quantum*. You can and will measure the quantum message to get a classical bit out, but you can also use the quantum message you get from Eve like any other quantum data, for example, you can apply whatever rotations and other instructions you like.

What is this good for?

Sending quantum data may not seem that much more useful than sending classical data; after all, sending classical data has gotten us a lot of neat things thus far. By contrast, applications for sending quantum data tend to be a bit more niche at the moment.

That said, that we can move quantum data around is a really useful example to help us understand how quantum computers work. The ideas that you developed in this Chapter aren't often that *directly* useful, but will help you build up really great stuff going forward.

Let's say Eve prepares a *quantum* message to you by using the operation `msg.ry(0.123)`, see [5.14](#) for how she can teleport this message to you.

Listing 5.14. A sample program that uses teleportation to move quantum data around, sending the message 0.123 to Eve.

```
if __name__ == "__main__":
    sim = Simulator(capacity=3)
    with sim.using_register(3) as (msg, here, there):
        msg.ry(0.123)
        teleport(msg, here, there)

        there.ry(-0.123)
    sim.dump()
```

1
2
3
4

- ➊ As before, you'll allocate a register of qubits, and give each qubit a name.
- ➋ Next, you prepare a message to send to Even. Here, we've shown using a particular angle as the message, but it really could be anything.
- ➌ You can then call the teleportation program that you wrote earlier to move the message you prepared onto Eve's qubit.
- ➍ If Eve then undoes your rotation by rotating by the opposite angle, you can check that the output of the `dump` instruction that we added above tells us that the register we allocated is back in the $|000\rangle$ state. This shows that your teleportation worked!

When you run this program, you'll get output similar to that in [5.15](#).

NOTE

Your output may differ by a global phase, depending on what measurement outcomes you got.

To verify that the teleportation worked, if you undo the instruction that Eve did on her qubit (`there.ry(0.123)`), you should get back the $|0\rangle$ state that you started with. With teleportation, Eve was able to send the quantum information stored in her qubit to you, using entanglement and classical communication.

Listing 5.15

```
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = (8, 1), type = ket
Qobj data =
[[1.]]
```

```
[0.]  
[0.]  
[0.]  
[0.]  
[0.]  
[0.]  
[0.]
```

Exercise 5.6: What if it just didn't do anything?

Try changing your operation or Eve's operation to convince yourself that you only get a $|000\rangle$ state at the end if you undo the same operation that Eve applied to her qubit.

Now you can brag to all your friends (and make whatever sci-fi references you want) that you can do teleportation. Hopefully you see why this is *not* the same as getting beamed down to a planet from orbit, and that when you teleport a message it does not communicate faster than the speed of light.

In this chapter you learned:

- Implement a quantum program to move data around a quantum computer using classical and quantum control
- Recognize a new way of visualizing single qubit operations called the Bloch sphere
- Predict the output of two-qubit operations, and Pauli operations

5.4 Part I: Conclusion

You have made it to the end of Part I, but sadly our qubits are in another castle \square . Getting through this Part is no mean feat, as we were refreshing a number of topics such as linear algebra and complex numbers, all while introducing a whole host of new quantum concepts. Certainly there will probably still be some questions or parts you are a bit shaky on, and that's ok. We will be using and practicing these skills to start developing more complicated quantum programs for cool applications like chemistry and cryptography.

Before we move onto that, though, give yourself a pat on the back, you've done a lot so far! Let's summarize some of what you've already accomplished:

- Refreshed your linear algebra and complex number skills
- Learned about what a qubit is as well as what you can do with one
- Built up a multiple-qubit simulator in Python
- Wrote a number of quantum programs for tasks like quantum key distribution (QKD), playing nonlocal games, and even quantum teleportation
- Learned braket notation for the states of quantum systems

Your Python simulator will continue to be a useful tool when trying to understand what

is happening when we get to some of the larger applications. For Part II, we will switch to primarily using Q# as our tool of choice to write our quantum programs. There are a number of reasons we will talk about in the next chapter for why we will choose to write these more advanced quantum programs in Q# over Python, but the main reasons are speed and extensibility. Plus, you can even use Q# from Python or with a Q# kernel for Jupyter, so you can use whatever development environment you like best!

See you in Part II!

Changing the odds: An introduction to Q#

This chapter covers:

- Using the Quantum Development Kit to write quantum programs in Q#,
- How to use Jupyter Notebook to work with Q#,
- How to run Q# programs using a classical simulator.

Up to this point, we've used Python to implement our own software stack to simulate quantum programs. Moving forward, though, we'll be writing more intricate quantum programs that will benefit from specialized language features that are hard to implement by embedding our software stack inside Python. Especially as we explore quantum algorithms, it's helpful to have a language tailor-made for quantum programming at our disposal. In this chapter, we'll get started with Q#, Microsoft's domain-specific language for quantum programming, included with the Quantum Development Kit.

6.1 *Introducing the Quantum Development Kit*

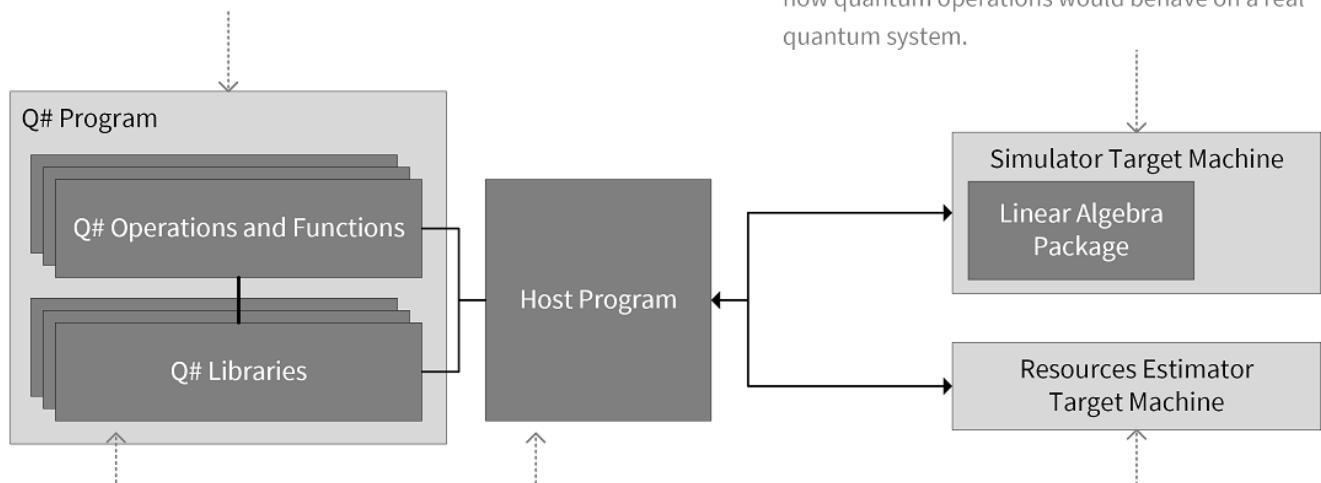
The Quantum Development Kit provides a new language, Q#, for writing quantum programs and simulating them using classical resources. Quantum programs written in Q# are run by thinking of quantum devices as a kind of accelerator, similar to how you might run code on a graphics card.

TIP

If you've ever used a graphics card programming framework like CUDA or OpenCL, this is a very similar model.

Figure 6.1. Q# software stack on a classical computer.

A Q# program can include both classical and quantum instructions.



A Q# program can also reference the Q# standard libraries, which provide a number of useful pre-made operations, or any other Q# libraries available on NuGet.org.

A classical program can then send Q# applications to a *target machine* such as a simulator for execution. After a Q# task is executed, the classical driver can collect the output of that task for further processing.

The QuantumSimulator target machine acts very similarly to the simulator that we implemented in Part I, and uses matrix multiplication to simulate how quantum operations would behave on a real quantum system.

Alternatively, the ResourcesEstimator target machine counts the resources needed to execute a quantum program, even if the program is too large to simulate on a classical computer.

Let's take a look at this software stack for Q#.

Our Q# program itself consists of operations and functions that instruct quantum and classical hardware to do certain things. There are also a number of libraries that are provided with Q# that have helpful, pre-made operations and functions to use in our programs.

Once the Q# program is written, we need a way for it to pass instructions to the hardware. A classical program, sometimes called a "driver" or a "host program," is responsible for allocating a target machine and running a Q# operation on that machine.

The Quantum Development Kit provides a plugin for Jupyter Notebook called IQ# that makes it easy to get started with Q# by providing host programs automatically for us. In Chapter 8, we'll see how to write host programs using Python and C#, but for now we'll focus on Q# itself. See Appendix B for instructions on setting up your Q#

environment to work with Jupyter Notebook.

Using the IQ# plugin for Jupyter Notebook, we can use one of two different target machines to run Q# code. The first is the `QuantumSimulator` target machine, which is very similar to the Python simulator that we have been developing. It will be a lot faster than our Python code at simulating our qubits.

The second is the `ResourcesEstimator` target machine which will allow us to estimate how many qubits and quantum instructions we would need to run it, without having to fully simulate it. This is especially useful for getting an idea of the resources you would need to run a Q# program for your application, as we'll see when we look at larger Q# programs later on in the book.

Figure 6.2. Getting started with IQ# and Jupyter Notebook

Jupyter Notebooks can contain text, headings, figures and other content alongside your code cells. Here, we've used a text cell to give a title to our notebook.

Classical Hello

The currently selected text `In [1]:` or code cell is indicated with a border.

```
function HelloWorld() : Unit {
    Message("Hello, classical world!");
}
```

`Out[1]:`
When we run a code cell, the output for that cell is shown below.

- `HelloWorld` Code in Jupyter Notebooks is divided into cells, each of which can be run independently. Here, we've used a code cell to define a new Q# function called `HelloWorld`.

`In [2]:` `%simulate HelloWorld`

`Hello, classical world!`

`Out[2]:` `()`

To get a sense for how everything works, let's start by writing out a purely classical Q# "hello, world" application. First, start Jupyter Notebook by running the following in a terminal:

```
jupyter notebook
```

This will automatically open a new tab in your browser with the home page for your Jupyter Notebook session. From the `New` ↓ menu, select "Q#" to make a new Q# notebook. Type the following into the first empty cell in the notebook and press

Control + Enter or ⌘ + Enter to run it.

```
function HelloWorld() : Unit {  
    Message("Hello, classical world!"); ①  
}
```

- ① This line defines a new function which takes no arguments, and returns the empty tuple, whose type is written as `Unit`.
- ② The `Message` function tells the target machine to collect a diagnostic message.
The `QuantumSimulator` target machine prints all diagnostics to the screen, so we can use `Message` in the same way as `print` in Python.

TIP
Watch out for semicolons!

Unlike Python, Q# uses semicolons rather than newlines to end statements. If you get a lot of compiler errors, make sure you remembered your semicolons.

You should get a response back listing that the `HelloWorld` function was successfully compiled. To run our new function, we can use the `%simulate` command in a new cell.

```
%simulate HelloWorld
```

TIP
A bit of classical magic

The `%simulate` command we used above is an example of a *magic command*, in that it's not actually a part of Q# itself, but is an instruction to the Jupyter Notebook environment. If you're familiar with the IPython plugin for Jupyter, you may have used similar magic commands to tell Jupyter how to handle Python plotting functionality. The magic commands we use in this book all start with % to make them easy to tell apart from Q# code.

In this example, `%simulate` allocates a target machine for us and sends a Q# function or operation to that new target machine. In Chapter 8, we'll see how to accomplish something similar using Python and C# host programs, instead of using Jupyter Notebook.

The Q# program is sent to the simulator, but in this case, the simulator just runs the classical logic, since there's no quantum instructions to worry about yet.

6.2 Functions and Operations in Q#

Now that we have the Quantum Development Kit up and running with Jupyter Notebook, let's use Q# to write some quantum programs. Back in Chapter 2, we saw that one useful thing to do with a qubit is to generate random numbers one classical bit at a time. Revisiting that application makes a great place to start with Q#, especially since random numbers are useful if you want to play games.

Long ago in Camelot, Morgana le Fay shared our love for playing games. Being a clever mathematician with skills well beyond her own day, Morgana was even known to use qubits from time to time as a part of her games. One day, as Sir Lancelot lay

sleeping under a tree, Morgana trapped him and challenged him to a little game: each of them must try to guess the outcome of measuring one of Morgana's qubits.

Two sides of the same... qubit?

In Chapter 2, we saw how we can generate random numbers one bit at a time by preparing and measuring qubits. That is, qubits can be used to implement *coins*. We'll use the same kind of idea in this Chapter as well, thinking of a coin as a kind of interface that allows its user to "flip" it and get out a random bit. That is, we can implement the coin interface by preparing and measuring qubits.

If the result of measuring along the Z axis is a 0, then Lancelot wins their game and gets to return to Genevieve. If the result is a 1, though, Morgana wins and Lancelot has to stay and play again. Notice the similarity to our QRNG program from before. Just as in chapter 2, we'll measure a qubit to generate random numbers, this time for the purpose of playing a game. Of course, Morgana and Lancelot could have also flipped a more traditional coin, but where is the fun in that?

Morgana's side game

1. Prepare a qubit in the $|0\rangle$ state
2. Apply the Hadamard operation (recall that the unitary operator H takes $|0\rangle$ to $|+\rangle$ and vice versa)
3. Measure the qubit in the Z axis. If the measurement result is a 0, then Lancelot can go home. Otherwise, he has to stay and play again!

Sitting at a coffee shop watching the world go by, we can use our laptops to predict what will happen in Morgana's game with Lancelot by writing a quantum program in Q#. Unlike the `ClassicalHello` function that we wrote above, our new program will need to work with qubits, so let's take a moment to see how to do so with the Quantum Development Kit.

The primary way that we interact with qubits in Q# is by calling *operations* that represent quantum instructions. For instance, the `H` operation in Q# represents the Hadamard instruction we saw in Chapter 2. To understand how these operations work, it's helpful to understand the difference between Q# operations and the functions that we saw in the `ClassicalHello` example above.

- **Functions** in Q# represent *predictable* classical logic, things like mathematical functions (`Sin`, `Log`). Functions always return the same output when given the same input.
- **Operations** in Q# represent code that can have *side effects*, such as sampling random numbers, or issuing quantum instructions which modify the state of one or more qubits.

This separation helps the compiler figure out how to automatically transform your code as a part of larger quantum programs; we'll see more about this later.

Another perspective on functions versus operations

Another way of thinking of the difference between functions and operations is that functions compute things, but cannot cause anything to *happen*. No matter how many times we call the square root function `Sqrt`, nothing about our Q# program has changed. By contrast, if we run the `X` operation, then an `X` instruction is sent to our quantum device, which causes a change in the state of the device. Depending on the initial state of the qubit that the `X` instruction was applied to, we can then tell that the `X` instruction has been applied by measuring the qubit. Because functions don't do anything in this sense, we can always predict their output exactly given the same input.

One important consequence is that functions cannot call operations, but operations can call functions. This is because you can have an operation which is not necessarily predictable call a predictable function and you still have something that may or may not be predictable. However, a predictable function cannot call a potentially unpredictable operation and still be predictable.

We'll see more about the difference between Q# functions and operations as we use them throughout the rest of the book.

Since we want quantum instructions to have an effect on our quantum devices (and on Lancelot's fate), all quantum operations in Q# are defined as operations (hence the name). For instance, suppose that Morgana and Lancelot prepare their qubit in the $|+\rangle$ state using the Hadamard instruction. Then we can predict the outcome of their game by writing out the quantum random number generator (QRNG) example from Chapter 2 as a Q# operation.

NOTE

There may be side effects to this operation...

When we want to send instructions to our target machine to do something with our qubits, we need to do so from an operation, since sending an instruction is a kind of *side effect*. That is, when we run an operation, we aren't just computing something, we're *doing something*. Running an operation twice isn't the same as running it once, even if we get the same output both times. Side effects aren't deterministic or predictable, and so we can't use functions to send instructions on how to manipulate our qubits.

In Listing 6.1, we'll do just that, starting by writing an operation called `NextRandomBit` to simulate each round of Morgana's game. Note that since `NextRandomBit` needs to work with qubits, it has to be an operation and not a function. We can ask the target machine for one or more fresh qubits with the `using` block.

NOTE Allocating qubits in Q#

The `using` statement is one of the only two ways we can ask the target machine for qubits. There's no limit to the number of `using` statements that we can have in our Q# programs, other than the number of qubits that each target machine can allocate. At the end of each `using` block, the qubits then go back to the target machine, so that one way to think of `using` blocks is to make sure that each qubit that is allocated is "owned" by a particular operation. This makes it impossible to "leak" qubits within a Q# program, which is very helpful given that qubits are likely to be very expensive resources on actual quantum hardware.

Q# offers one other way to allocate qubits, known as *borrowing*. Unlike when we allocate qubits with `using` statements, the borrowing statement lets us borrow qubits that are owned by different operations without knowing what state they start in. We won't see much of borrowing in this book, but the borrowing statement works very similarly to the `using` statement in that it makes it impossible for us to forget that we've borrowed a qubit.

By convention, all qubits start off in the $|0\rangle$ state right after we get them, and we promise the target machine that we'll put them back into the $|0\rangle$ state at the end of the block so that they're ready for the target machine to give to the next operation that needs them.

Listing 6.1. Simulating one round of Morgana's game using Q#

```
operation NextRandomBit() : Result {
    using (qubit = Qubit()) {
        H(qubit);
        let result = M(qubit);
        Reset(qubit);
        return result;
    }
}
```

1
2
3
4
5
6

- ➊ This time, because we want to use a qubit, we declare an operation instead of a function. Since our operation needs to return a result to its caller, we denote by changing the return type to the Q# type `Result`.
- ➋ The `using` keyword in Q# asks the target machine for one or more qubits. Here, we ask for a single value of type `Qubit`, which we store in the new variable `qubit`.
- ➌ Quantum operations such as the Hadamard operation can be found in the `Microsoft.Quantum.Intrinsic` namespace. For instance, we can call Hadamard using the `Microsoft.Quantum.Intrinsic.H` operation. After calling `H`, `qubit` is in the $H|0\rangle = |+\rangle$ state.
- ➍ Next, we use the `M` operation to measure our qubit in the Z basis, saving the result to the `result` variable we declared earlier. Since we are in an equal superposition of $|0\rangle$ and $|1\rangle$, `result` will be either Zero or One with equal probability.
- ➎ Before returning our qubit to the target machine, we use the `Microsoft.Quantum.Intrinsic.Reset` operation to return it to the $|0\rangle$ state. Since we've already stored the classical data we got from our measurement into the `result` variable, we can safely reset the qubit without losing any information that we care about.
- ➏ We finish our operation by returning the measurement result back to the caller.

Next, we need to see how many rounds it takes for Lancelot to get the Zero he needs to go home. Let's write an operation to play rounds until we get a Zero. Since this operation simulates playing Morgana's game, we'll call it `PlayMorganasGame`.

Listing 6.2. Simulating many rounds of Morgana's game using Q#

```
operation PlayMorganasGame() : Unit {
    mutable nRounds = 0;                                ①
    mutable done = false;
    repeat {
        set nRounds = nRounds + 1;                      ②
        set done = (NextRandomBit() == Zero);            ③
    }
    until (done)
    fixup {}                                         ④

    Message($"It took Lancelot {nRounds} turns to get home."); ⑤
}
```

- ➊ All Q# variables are immutable by default—we can use the `mutable` keyword to declare a variable that we can change later with the `set` keyword. Here, we start by initializing a mutable variable indicating how many rounds have already passed, and a mutable variable we'll use to exit the loop.
- ➋ Q# allows operations to use a kind of loop called a "repeat-until-success" (RUS) loop. Unlike a `while`-loop, RUS loops allow us to specify a "fixup" that runs if the condition to exit the loop isn't met. Note that the `fixup` block is required, even if it is empty.
- ➌ Inside our loop, we call the QRNG that we wrote above as the `NextRandomBit` operation. We check to see if the result is a Zero (that is, if Lancelot wins and can leave), and if so, set `done` to be true.
- ➍ If we got a Zero, then we can stop the loop.
- ➎ Finally, we use `Message` again to print the number of rounds to the screen. To do so, we use `$""` strings which, similar to `""` strings in C# and `f""` strings in Python, let us include variables in the diagnostic message by using `{}` placeholders inside the string.

Why Do We Need to Reset Qubits?

In Q#, when we allocate a new qubit with `using`, we promise the target machine that we will put it back in the $|0\rangle$ state before we deallocate it. At first glance, this seems rather unnecessary, as the target machine could just reset the state of qubits when they are deallocated—after all, we will often simply call the `Reset` operation at the end of a `using` block.

It is important to note, though, that the `Reset` operation works by making a measurement in the `Z` basis and flipping the qubit with an `X` operation if the measurement returns One. **In many quantum devices, measurement is much more expensive than other operations**, such that if we can avoid calling `Reset` we can reduce the cost of our quantum programs. Especially given the limitations of medium-term devices, this kind of optimization can be critical in making a quantum program practically useful.

Later in the chapter, we will see examples of where we know the state of a qubit when it needs to be deallocated, such that we can "unprepare" the qubit instead of measuring it.

We can run this new operation with the `%simulate` command in a very similar fashion

as the `ClassicalHello` example. When we do so, we can see how long Lancelot has to stay:

Listing 6.3. Output from running the Qrng application

```
In []: %simulate PlayMorganasGame
It took Lancelot 1 turns to get home.
Out[]: ()
```

Looks like Lancelot got lucky that time! Or perhaps unlucky, if he was bored of hanging 'round the table in Camelot.

6.3 Passing Operations as Arguments

Suppose in Morgana's game above, we were interested in sampling random bits with non-uniform probability. After all, Morgana didn't promise Lancelot *how* she prepared the qubit that they measure; she can keep him playing longer if she makes a biased coin with their qubit instead of a fair coin.

The easiest way to modify Morgana's game is to, instead of calling `H` directly, take as an input an operation representing what Morgana does to prepare for their game. To take an operation as input, we need to write down the *type* of the input, just as can write down `qubit : Qubit` to declare an input `qubit` of type `Qubit`. Operation types are indicated by thick arrows (`=>`) from their input type to their output type. For instance, `H` has type `Qubit => Unit` since `H` takes a single qubit as input and returns an empty tuple as its output.

Listing 6.4. Using operations as inputs in order to predict Morgana's game.

```
operation PrepareFairCoin(qubit : Qubit) : Unit {
    H(qubit);
}

operation NextRandomBit(
    statePreparation : (Qubit => Unit)           ①
) : Result {
    using (qubit = Qubit()) {
        statePreparation(qubit);
        return result = MResetZ(qubit);            ②
    }
}                                              ③
```

- ① This time, we've added a new input called `statePreparation` to `NextRandomBit` that represents the operation we want to use to prepare the state we use as a coin. In this case, `Qubit => Unit` is the type of any operation which takes a single qubit and returns the empty tuple type `Unit`.
- ② Within `NextRandomBit`, the operation passed as `statePreparation` can be called in the same way as any other operation.

- ③ The Q# standard libraries provide `MResetZ` as a convenience for measuring and resetting a qubit in one step. This is equivalent to the `set result = M(qubit); Reset(qubit);` statements we saw in the previous example, but requires one less measurement to perform.

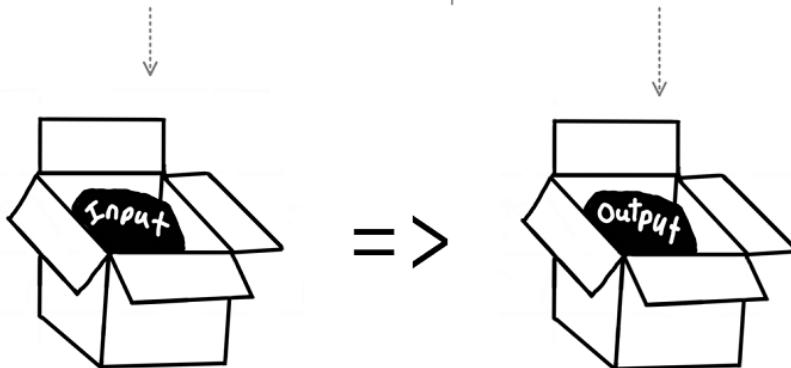
Tuple-In Tuple-Out

All functions and operations in Q# take a single *tuple* as an input and return a single *tuple* as an output. For instance, a function declared as `function Pow(x : Double, y : Double) : Double { ... }` takes as input a tuple (`Double, Double`), and returns a tuple (`Double`) as its output. This works because of a property known as *singleton–tuple equivalence*. For any type '`T`', the tuple (`'T`) containing a single '`T`' is equivalent to '`T`' itself. In the example of `Pow`, this means that we can think of the output as a tuple (`Double`) that is equivalent to `Double`.

Figure 6.3. Representing operations with a single input and a single output

No matter how many inputs an operation takes, we can always think of that operation as taking exactly one input: a tuple containing all of the inputs.

Similarly, every operation can be thought of as returning exactly one output.



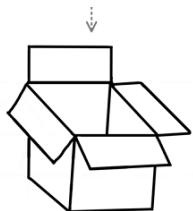
With this in mind, a function or operation that returns no outputs can be thought of as returning a tuple with no elements, `()`. The type of such tuples is called `Unit`, similar to other tuple-based languages such as F#. If we think of a tuple as a kind of box, then this is distinct from `void` as used in C, C++, or C# because there still is *something* there, namely a box with nothing in it.

In Q#, we always return a box, even if that box is empty.

There's no meaning in Q# to a function or operation that returns "nothing." For more details, see Section 7.2 of Get Programming with F#.

Figure 6.4. Unit versus void

If an operation that doesn't take any inputs or that doesn't return any outputs, then we represent that with an empty tuple, written in Q# as () .



$() : \text{Unit}$



void

Just like any other value in Q#, the empty tuple has a type. The type of () is called **Unit**.

Most quantum operations that don't involve a measurement will return **Unit**.

In other languages, like C, C++, C#, or Java, you'll sometimes see the keyword **void** used to indicate that a function or method doesn't return anything. Unlike **Unit**, **void** represents that there isn't a value at all, not even an empty box.

In this example, we see that `NextRandomBit` treats its input `statePreparation` as a "black box." The only way to learn anything about Morgana's preparation strategy is to *run* it.

Put differently, we don't want to do anything with `statePreparation` that implies we know what it does or what it is. The only way that `NextRandomBit` can interact with `statePreparation` is by calling it, passing it a `Qubit` to act on.

This allows us to reuse the logic in `NextRandomBit` for many different kinds of state preparation procedures that Morgana might use to cause Lancelot a bit of trouble. For example, suppose she wants a biased coin that returns a One $\frac{3}{4}$ of the time and a Zero $\frac{1}{4}$ of the time. Then, we might run something like the following to predict this new strategy:

Listing 6.5. Passing different state preparation strategies to the `PlayMorganasGame` example.

```
open Microsoft.Quantum.Math;           ①
operation PrepareQuarterCoin(qubit : Qubit) : Unit {
    Ry(2.0 * PI() / 3.0, qubit);        ②
```

}

- ➊ Classical math functions such as Sin, Cos, Sqrt, and ArcCos, as well as constants like PI() are provided by the Microsoft.Quantum.Math namespace, so we open it as well as the intrinsics.
- ➋ The Ry operation implements the Y -axis rotation that we saw in Chapter 2. Q# uses radians rather than degrees to express rotations, so this is a rotation of 120° about the Y -axis. Thus, if qubit starts in $|0\rangle$, this prepares qubit in the state $R_y(-120^\circ)|0\rangle = \sqrt{3}/4|0\rangle + \sqrt{1}/4|1\rangle$, such that the probability of observing 1 when we measure is $\sqrt{3}/4^2 = 3/4$.

We can make this example even more general, allowing Morgana to specify an arbitrary bias for her coin (which is implemented by their shared qubit):

Listing 6.6. Passing operations to implement PlayMorganasGame with arbitrary coin biases.

```
operation PrepareBiasedCoin(morganaWinProbability : Double, qubit : Qubit) : Unit {
    let rotationAngle = -2.0 * ArcCos(Sqrt(morganaWinProbability));           ➊
    Ry(rotationAngle, qubit);
}

operation PrepareMorganasCoin(qubit : Qubit) : Unit {                         ➋
    PrepareBiasedCoin(0.62, qubit);
}
```

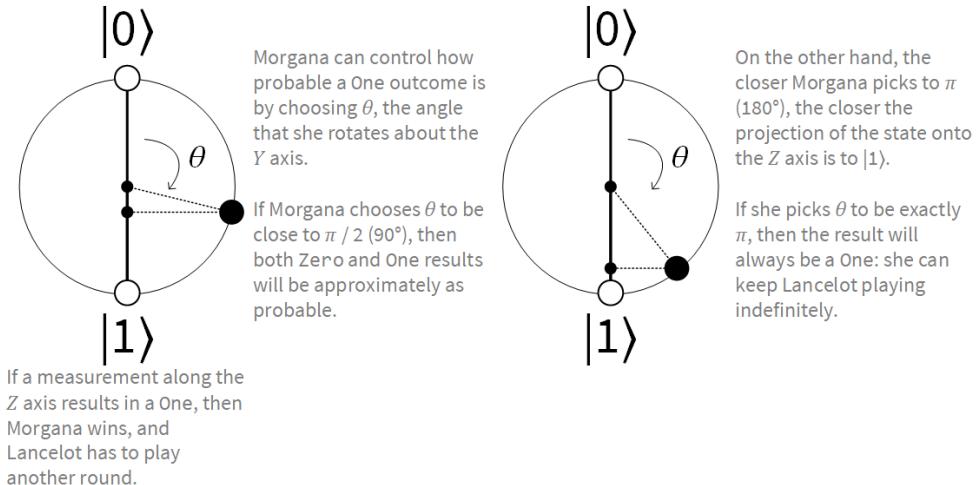
- ➊ We need to find out what angle we rotate the input qubit by in order to get the right probability of seeing a Zero as our result. This takes a little bit of trigonometry, see the sidebar below for the details.
- ➋ This operation has the right type signature (`Qubit => Unit`) and we can see that the probability Morgana will win each round is 62%.

Working out the trigonometry

As we've seen a number of times, quantum computing deals extensively with *rotations*. To figure out what angles we need for our rotations, we need to rely on a little bit on a branch of mathematics for describing rotation angles, known as trigonometry (literally, the study of triangles). For instance, as we saw in Chapter 2, rotating $|0\rangle$ by an angle θ about the Y axis results in a state $\cos(-\theta/2)|0\rangle + \sin(-\theta/2)|1\rangle$. We know we want to choose θ such that $\cos(-\theta/2) = \sqrt{62\%}$, so that we get a 62% probability of getting a Zero result. That means we need to "undo" the cosine function to figure out what θ needs to be. In trigonometry, the inverse of the cosine function is called the arccosine function, and is written `arccos`. Taking the arccosine of both sides of $\cos(-\theta/2) = \sqrt{62\%}$ gives us $\text{arccos}(\cos(-\theta/2)) = \text{arccos}(\sqrt{62\%})$. We can cancel out the `arccos` and `cos` to find a rotation angle that gives us what we need, $-\theta/2 = \text{arccos}(\sqrt{62\%})$. Finally, we multiply both sides by -2 to get the equation we used in line ➊ of 6.1.

Figure 6.5. How Morgana can choose θ to control how her game plays out

If a measurement along the Z axis results in a Zero, then Lancelot wins and can go home.



This is somewhat unsatisfying, though, in that the operation `PrepareMorganasCoin` introduces a lot of boilerplate just to lock down the value of 0.62 for the input argument `headsProbability` to `PrepareBiasedCoin`. If Morgana changes her strategy to have a different bias, then using this approach, we'll need another new boilerplate operation to represent it. Taking a step back, let's look at what `PrepareMorganasCoin` actually *does*. It starts with an operation `PrepareBiasedCoin : (Double, Qubit) => Unit`, and wraps it into an operation of type `Qubit => Unit` by locking down the `Double` argument to 0.62. That is, it removes one of the arguments to `PrepareBiasedCoin` by fixing the value of that input to 0.62.

Thankfully, Q# provides a convenient shorthand for making new functions and operations by locking down some (but not all!) of the inputs. Using this shorthand, known as *partial application*, we can rewrite the above in a more readable form:

Listing 6.7. Using partial application to make it easier to vary Morgana's strategy.

```
let flip = NextRandomBit(PrepareBiasedCoin(0.62, _));
```

The `_` here indicates that a part of the input to `PrepareBiasedCoin` is **missing**. We say that `PrepareBiasedCoin` has been partially applied. Whereas `PrepareBiasedCoin` had type `(Double, Qubit) => Unit`, because we filled in the `Double` part of the input, `PrepareBiasedCoin(0.62, _)` has type `Qubit => Unit`, making it compatible with our modifications to `NextRandomBit`.

TIP

Partial application in Q# is similar to `functools.partial` in Python and the `_` keyword in Scala.

Another way to think of partial application is as a way to make new functions and operations by specializing existing functions and operations:

```
function BiasedPreparation(headsProbability : Double) : (Qubit => Unit) { ①
    return PrepareBiasedCoin(headsProbability, _); ②
}
```

- ➊ Here, the output type of `BiasedPreparation` is an operation that takes a `Qubit` and returns the empty tuple. That is, `BiasedPreparation` is a function that makes new operations!
- ➋ We make the new operation by passing along `headsProbability`, but leaving a blank `(_)` for the target qubit. This gives us an operation that takes a single `Qubit` and substitutes in the blank.

It may seem a bit confusing that `BiasedPreparation` returns an operation from a function, but this is completely consistent with the split between functions and operations described above, since `BiasedPreparation` is still predictable. In particular, `BiasedPreparation(p)` always returns the same operation for a given `p`, no matter how many times you call the function. We can assure ourselves that this is the case by noticing that `BiasedPreparation` only partially applies operations, but never calls them.

6.4 Playing Morgana's Game in Q#

With first-class operations and partial application at the ready, we can now make a more complete version of Morgana's game.

The Q# standard libraries

The Quantum Development Kit comes with a variety of different standard libraries that we'll see throughout the rest of the book. In 6.2, for example, we make use of an operation `MResetZ` that both measures a qubit (similar to `M`) and resets it (similar to `Reset`). This operation is offered by the `Microsoft.Quantum.Measurement` namespace, one of the main standard libraries that comes with the Quantum Development Kit. A full list of the operations and functions available in that namespace can be found at docs.microsoft.com/qsharp/api/qsharp/microsoft.quantum.measurement. For now, though, don't worry too much about it; we'll see more of the Q# standard libraries as we go.

Listing 6.8. Complete listing of Q# operations for the biased PlayMorganasGame example

```
open Microsoft.Quantum.Math; ①
open Microsoft.Quantum.Measurement;

operation PrepareBiasedCoin(winProbability : Double, qubit : Qubit) : Unit {
    let rotationAngle = 2.0 * ArcCos(Sqrt(1.0 - winProbability)); ②
```

```

    Ry(rotationAngle, qubit);
}

operation NextRandomBit(statePreparation : (Qubit => Unit)) : Result {
    using (qubit = Qubit()) {
        statePreparation(qubit);
        return MResetZ(qubit); ③
    }
}

operation PlayMorganasGame(winProbability : Double) : Unit {
    mutable nRounds = 0;
    mutable done = false;
    let prep = PrepareBiasedCoin(winProbability, _); ④
    repeat {
        set nRounds = nRounds + 1;
        set done = (NextRandomBit(prep) == Zero);
    }
    until (done)
    fixup {}

    Message($"It took Lancelot {nRounds} turns to get home.");
}

```

- ➊ We start by opening namespaces from the Q# standard library to help with classical math, and to help with measuring qubits.
- ➋ The rotation angle chooses the bias the coin has.
- ➌ Here we use the operation we passed in as `statePreparation` and apply it to the qubit.
- ➍ The `MResetZ` operation is defined in the `Microsoft.Quantum.Measurement` namespace that we open at the beginning of the sample. It measures the qubit in the `Z` basis and then applies what operations are needed to return the qubit to the `|0>` state.
- ➎ We use *partial application* to specify the bias for our state preparation procedure, but not the target qubit. While `PrepareBiasedCoin` has type `(Double, Qubit) => Unit`, `PrepareBiasedCoin(0.2, _)` "fills in" one of the two inputs, leaving an operation with type `Qubit => Unit`, as expected by `EstimateBias`.

Providing documentation for Q# functions and operations

Documentation can be provided for Q# functions and operations by writing small specially formatted text documents in triple-slash (///) comments before a function or operation declaration. These documents are written in Markdown, a simple text formatting language used on sites like GitHub, Azure DevOps, Reddit, and Stack Exchange, and by site generators like Jekyll. The information in /// comments is shown when hovering over calls to that function or operation, and can be used to make API references similar to those at docs.microsoft.com/quantum/.

Different parts of /// comments are indicated with section headers, for example /// # Summary. For example, we may document the `PrepareBiasedCoin` operation from 6.2 with the following:

```

/// # Summary
/// Prepares a state representing a coin with a given bias.
///
/// # Description
/// Given a qubit initially in the |0> state, applies operations

```

```

/// to that qubit such that it has the state  $\sqrt{p} |0\rangle + \sqrt{1-p} |1\rangle$ ,
/// where p is provided as an input.
/// Measurement of this state returns a One Result with probability p.
///
/// # Input
/// ## winProbability
/// The probability with which a measurement of the qubit should return One.
/// ## qubit
/// The qubit on which to prepare the state  $\sqrt{p} |0\rangle + \sqrt{1-p} |1\rangle$ .
operation PrepareBiasedCoin(
    winProbability : Double, qubit : Qubit
) : Unit {
    let rotationAngle = 2.0 * ArcCos(Sqrt(1.0 - winProbability));
    Ry(rotationAngle, qubit);
}

```

When using IQ#, you can look up documentation comments by using the ? command. For instance, you can look up the documentation for the X operation by running X? in an input cell.

For a full reference, see docs.microsoft.com/quantum/language/statements#documentation-comments.

To estimate the bias of a particular state preparation operation, we can run the PlayMorganasGame operation repeatedly and count how many times we get a Zero.

Let's pick a value for winProbability and run the PlayMorganasGame operation with to see how long it Lancelot will be stuck.

Listing 6.9. Running PlayMorganasGame to see how long Lancelot is stuck.

```

In []: operation Main() : Unit {
    PlayMorganasGame(0.9);
}
In []: %simulate Main
It took Lancelot 5 turns to get home.

```

Try playing around with different values of winProbability! Note that if Morgana really tips the scales, we can confirm that it will take Lancelot quite a long time to make it back to Genevieve.

Listing 6.10. Output of varying Morgana's strategy.

```

In []: operation Main() : Unit {
    PlayMorganasGame(0.999);
}
In []: %simulate Main
It took Lancelot 3255 turns to get home.

```

6.5 Summary

In this chapter you learned:

- how to use the Quantum Development Kit to write quantum programs in Q#, and

- how to use Jupyter Notebook to run your quantum programs with a quantum simulator.

In the next chapter, we'll build on these skills by going back to Camelot to find our first example of a quantum algorithm, the Deutsch–Jozsa algorithm.

What is a Quantum Algorithm?

This chapter covers:

- What is a quantum algorithm?
- How to design oracles to represent classical functions in quantum programs
- A first example of a quantum algorithm
- Several useful quantum programming techniques

One important application for quantum algorithms is in obtaining speedups for solving problems where we need to search over inputs to a function that we're trying to learn about. Such functions could be obfuscated (such as hash functions), or could be computationally difficult to evaluate (common in studying mathematical problems). In either case, applying quantum computers to such problems requires us to understand how we program and provide input to quantum algorithms. To learn how to do so, we'll program up and run an implementation of an algorithm known as the *Deutsch–Jozsa algorithm*, which will let us learn properties of unknown functions quickly using quantum devices.

7.1 Classical and quantum algorithms

Definition 7.1: Algorithm

noun: a step-by-step procedure for solving a problem or accomplishing some end. —Merriam–Webster Dictionary

When we talk about classical programming, we sometimes say that a program implements an *algorithm*; that is, a sequence of steps that can be used to solve some problem. If we want to sort a list, for example, we can talk about the quicksort algorithm independently of what language or operating system we are using. We often specify these steps at a high level. In the quicksort example, we might list the steps as something like the following.

Quicksort algorithm

1. If the list to be sorted is empty or only has one element, return it as-is.
2. Pick an element of the list to be sorted, called the *pivot*.
3. Separate all other elements of the list into those that are smaller than the pivot, and those that are larger.
4. Quicksort each new list recursively.
5. Return the first list, then the pivot, and finally the second list.

These steps then serve as a guide for writing an implementation in a particular language of interest. Say we want to write the quicksort algorithm in Python:

Listing 7.1. An example implementation of the quicksort algorithm.

```
def quicksort(xs):
    if len(xs) > 1:
        pivot = xs[0]
        left = [x in xs[1:] if x <= pivot]
        right = [x in xs[1:] if x > pivot]
        return quicksort(left) + [pivot] + quicksort(right)
    else:
        return xs
```

- ➊ We check for the base case by checking if there's at least two elements in the list.
- ➋ We pick the first element to be our pivot for step 2.
- ➌ Next, we write out Python code that builds two new lists as described in step 3.
- ➍ Finally, we concatenate everything back together as described in steps 4 and 5.

A well-written algorithm can help guide how to write implementations by making the steps that must be executed clear. Quantum algorithms are the same in this respect: they list the steps that we need to perform in any implementation.

Definition 7.2: Quantum program

A *quantum program* is an implementation of a quantum algorithm, consisting of a *classical program* that sends instructions to a *quantum device* in order to prepare a particular state or measurement result.

As we saw in Chapter 6, when we write a Q# program, we are writing a classical program which sends instructions to one of several different target machines on our behalf, as illustrated as [7.1](#), returning measurements back to our classical program.

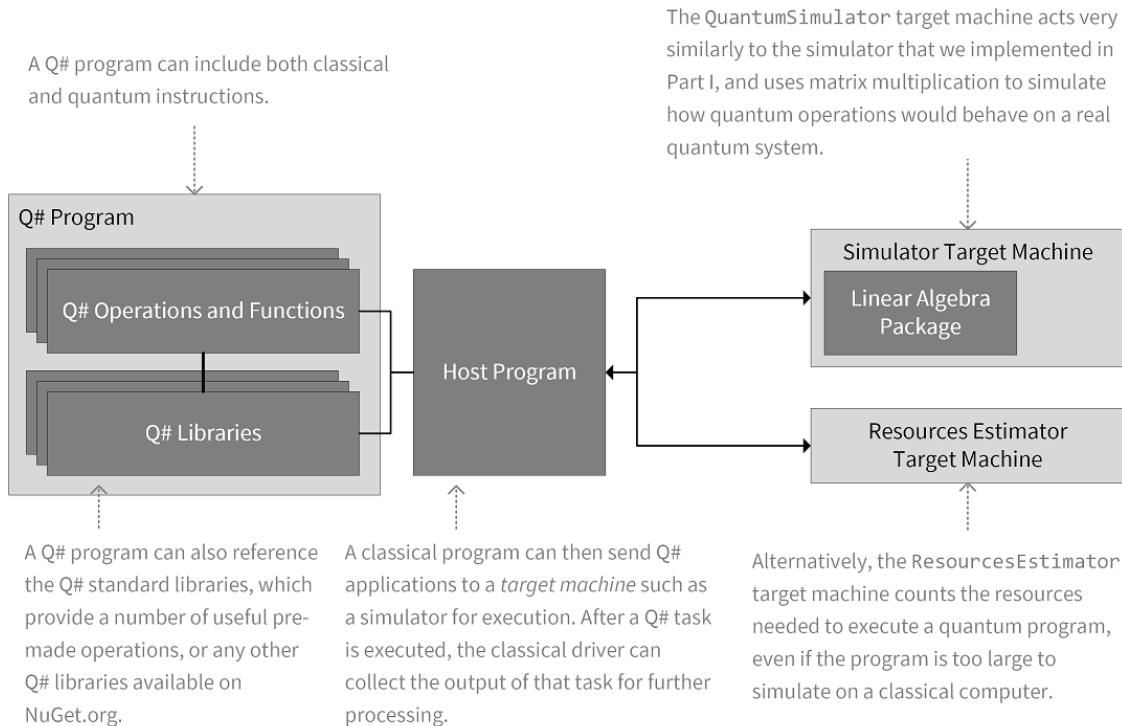
The art of quantum programming

We cannot copy quantum states, but if they resulted from running a program, we can tell someone else what steps they need to take to prepare the same states. As we saw above, quantum programs are a special kind of classical program, so we can copy them with reckless abandon. As we will see throughout the rest of the book, any quantum state can either be approximated or written out exactly by the output of a quantum program that starts with only copies of the $|0\rangle$ state. For example, in Chapter 2, we prepared the initial state $|+\rangle$ of a QRNG by a program consisting of a single H instruction.

Put differently, we can think of a program as being a recipe for how to prepare a qubit. Given a qubit, we cannot determine what recipe was used to prepare it, but we can copy the recipe itself as much as we like.

Whereas the steps in executing quicksort instruct the Python interpreter to compare values and to move values around in memory, the steps in a Q# program instruct our target machines to apply rotations and measurements to qubits in a device. As shown in [7.1](#), we can use a host program to send Q# applications to each different target machine to run. For now, we'll keep using the IQ# plugin for Jupyter Notebook as our host program; in the next chapter, we'll see how to use C# to write our own host programs as well.

Figure 7.1. The Quantum Development Kit software stack.



Most of the time in this book, we'll be interested in simulating our quantum programs, so we'll use the `QuantumSimulator` target machine. This simulator works very similarly to the ones we developed in Chapters 2 and 4, as it executes instructions such as the Hadamard instruction `H` by multiplying quantum states with unitary operators like H .

TIP

As in previous chapters, we can use fonts to distinguish instructions like `H` from the unitary matrices like H that we use to simulate those instructions.

The `ResourcesEstimator` target machine allows you to not run a quantum program, but to get estimates on how many qubits it *would* take to run it. This is useful for larger programs that cannot be classically simulated or run on available hardware, to help you learn how many qubits it will take; we'll see more about this target machine later.

Since Q# applications send instructions to the target machines that we use to run them, it's easy to reuse Q# code later across different target machines that share the same instruction set. The `QuantumSimulator` target machine, for instance, uses the same instructions that we expect actual quantum hardware to take once it becomes available, so that we can test Q# programs on simulators now using small instances of problems, and can then run the same programs on quantum hardware later.

What remains in common across these different target machines and applications is that we need to write the program that sends instructions to the target machine in order to accomplish some goal. Our task as quantum programmers is thus to make sure that these instructions have the effect of solving some useful problem.

TIP

The way that we use simulators to test Q# programs is a bit similar to the way we use simulators to test programs for other specialized hardware like field-programmable gate arrays (FPGAs), or the way we use emulators to test applications for mobile devices from our desktops and laptops. The main difference is that we can only use a classical computer to simulate a quantum computer for a very small numbers of qubits, or for restricted kinds of programs.

This is much easier to do when we have an algorithm guiding us to organize the steps that need to happen in both the classical and quantum devices. In developing new quantum algorithms, we can make use of the quantum effects, such as entanglement, which we saw in Chapter 4.

TIP

In fact, to get any advantage from our quantum hardware we *must* use the unique quantum properties of the hardware, or else we just have a more expensive and slower classical computer.

7.2 ***Deutsch–Jozsa Algorithm: moderate improvements for searching***

So what might make a good example of a *quantum* algorithm that takes advantage of

our shiny new quantum hardware? We learned in Chapters 4 and 6 that thinking about games can often help, and this is no exception. To find our game for this chapter, let's take a trip back to Camelot, where Merlin finds himself facing a test...

7.2.1 *Lady of the (Quantum) Lake*

Merlin, the famous and wise wizard, has just encountered *Nimue*, the lady of the lake. Nimue, seeking a capable mentor for the next King of England, has decided to test Merlin to see if he is up to the task. Two bitter rivals, Arthur and Mordred, are vying for the throne, such that if Merlin is to accept Nimue's task, he must choose whom to mentor as king.

For her part, Nimue does not care who becomes king, so long as Merlin can give them sage council. What Nimue **is** concerned about is whether Merlin, the appointed instructor for the new king, will be reliable and consistent in his leadership.

Since Nimue shares our love of games, she has decided to play a game with Merlin to test if he will be a good mentor or not. Nimue's game, Kingmaker, tests to see if Merlin is *consistent* in his role as advisor to the king. To play Kingmaker, Nimue gives Merlin the name of one of the two bitter rivals for the throne, to which Merlin must respond with whether Nimue's candidate should be the true heir to the throne or not.

Kingmaker game rules

- In each round, Nimue asks Merlin a single question of the form "Should *potential heir* be the king?"
- Merlin must answer either "yes" or "no," giving no additional information.

Each round gives Nimue more information about the realm of mortals, so her objective is to ask as few questions as is needed to catch Merlin out if he is not trustworthy.

Nimue's objectives

- Verify that Merlin will be a good mentor to the new King of England.
- Ask as few questions as possible to verify.
- Avoid learning who Merlin will say yes to mentoring.

At this point, Merlin has 4 possible strategies:

Merlin's strategies:

1. Say "yes" when asked if Arthur should be king, and "no" otherwise (good mentor)
2. Say "yes" when asked if Mordred should be king, and "no" otherwise (good mentor)
3. Say "yes" regardless of who Nimue asks about (bad mentor)
4. Say "no" regardless of who Nimue asks about (bad mentor)

One way to think of Merlin's strategies is by using the concept of a truth table once again. Suppose, for instance, Merlin has decided to be singularly unhelpful and deny

any candidates to the throne. Then we might write this down using the truth table in 7.1.

Table 7.1. Truth table for one possible strategy of Kingmaker: Merlin always says no.

Input (Nimue)	Output (Merlin)
"Should Mordred be king?"	"No."
"Should Arthur be king?"	"No."

At this point, Nimue would be right to complain about Merlin's wisdom as a mentor! Merlin has not been consistent with his charge to choose between Arthur and Mordred. While Nimue may not care whom Merlin picks, he surely must pick *someone* to mentor and prepare for the throne.

Nimue needs a strategy to determine if Merlin has either strategy 1 or 2 (good mentor) or if Merlin is playing according to 3 or 4 (bad mentor) in as few rounds of the game as possible. She could just ask both questions: "Should Mordred be king?" and "Should Arthur be king?" and then compare his answers, but this would result in Nimue knowing for sure who he chose to be king. After all, with each question Nimue learns more about the mortal affairs of the kingdom — how distasteful!

While it would seem Nimue's game is doomed to force her to learn his choice of heir, she is in luck. This being a quantum lake, we'll see throughout the rest of this chapter that Nimue can ask a *single* question that will tell her *only* if Merlin is committed to his role as mentor, and not whom he has chosen.

Since we don't have a quantum lake at our disposal, let's try to model what Nimue is doing with quantum instructions in Q# on our classical computer and then simulate it. Let's represent Merlin's strategy by a classical function f , which takes Nimue's question as an input x . That is, we'll write $f(\text{Arthur})$ to mean "what Merlin answers when asked if Arthur should be king." Note that since Nimue will only ask one of two questions, which question she asks is an example of a bit. Sometimes it's convenient to write out that bit using the labels "0" and "1", while sometimes it's helpful to label Nimue's input bit using the Boolean values "False" and "True." After all, "1" would be a pretty strange answer to a question like "should Mordred be king?"

Table 7.2. Encoding Nimue's question as a bit

Nimue's question	Representation as a bit	Representation as a Boolean
"Should Mordred be king?"	0	False
"Should Arthur be king?"	1	True

Using bits, we write $f(0) = 0$ to mean that Nimue if asks Merlin "should Mordred be king," his answer will be no.

If she didn't have any quantum resources, to be sure of what Merlin's strategy is,

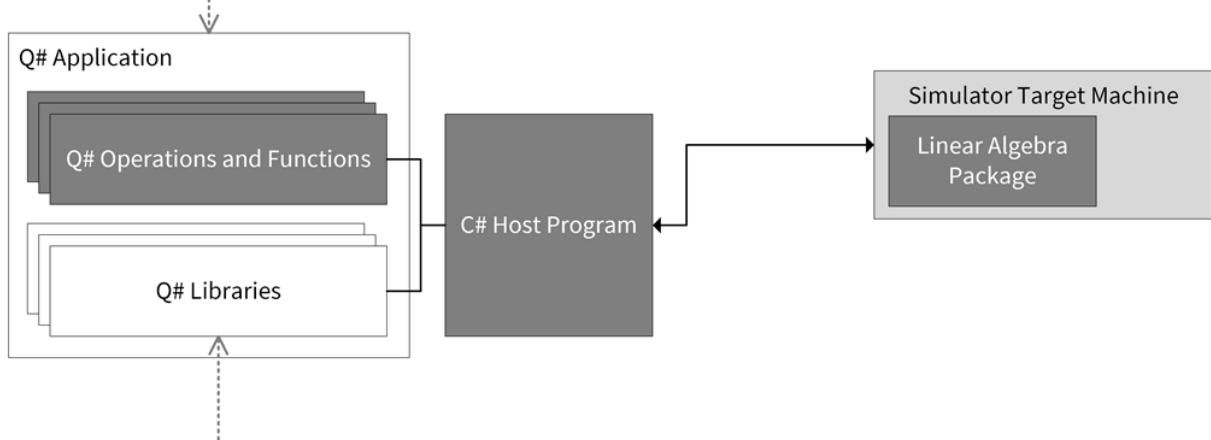
Nimue would have to try both inputs to f ; that is, she'd have to ask both questions to Merlin. Trying all the inputs would give us Merlin's full strategy, which as noted, Nimue is not really interested in.

To solve this, instead of having to ask Merlin about both Mordred and Arthur, we can implement a quantum algorithm in Q# that uses quantum effects to learn whether Merlin is a good mentor or not by asking him only **one** question. Using the simulators provided with the Quantum Development Kit, we can even run our new Q# program on our laptops or desktops!

In the rest of this chapter, we'll learn an example of how to write this quantum algorithm, called the Deutsch–Jozsa algorithm (see [7.1](#)).

Figure 7.2. Where we will be working in the Q# software stack for this chapter.

For the rest of Chapter 6, we'll be writing Q# applications that we can run on our classical computers using the simulator target machine.



We'll make use of the Q# libraries that come with the Quantum Development Kit, but we won't be writing our own in this chapter.

Let's try and sketch out what our quantum program will look like.

The possible inputs and outputs for f (Merlin's strategy) are True and False. We can write down a truth table for f using the inputs and outputs we get when we call f . For instance, if f is the classical NOT operation (often denoted \neg), then we will observe that $f(\text{True})$ is False and vice versa. As shown in [7.2](#), using a classical NOT operation as a strategy in our game corresponds to picking Mordred to be king.

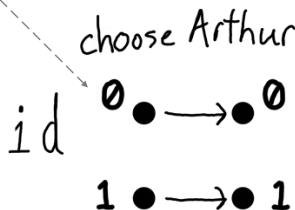
Table 7.3. Truth table for the classical NOT operation

Input	Output
True ("Should Arthur be king?")	False ("No.")
False ("Should Mordred be king?")	True ("Yes.")

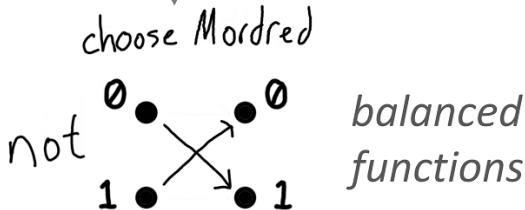
There are four possible options for the definition of our function f , each of which represents one of the four strategies available to Merlin, as summarized in [7.3](#).

Figure 7.3. Four different functions from one bit to one bit.

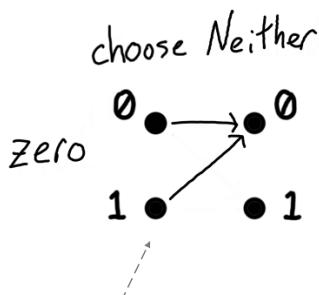
To help us write things down in programs, we'll label everything using bits. When Nimue asks about Mordred, for instance, we'll use a 0, and when Merlin answers with "no," we'll draw an arrow from 0 to 0.



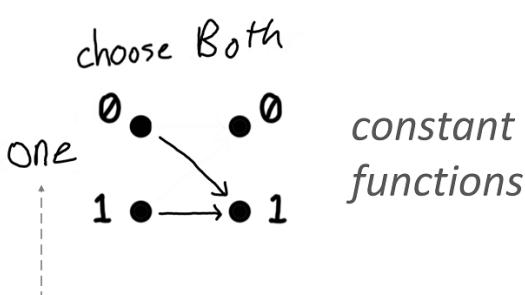
Each different function represents one of the different strategies that Merlin could use to respond to Nimue. If Merlin uses the not function as his strategy (using the classical \neg operation), then he'll say 1 ("yes") when asked 0 ("Mordred").



Two of the four strategies Merlin could use are *balanced functions*, meaning he picks exactly one candidate to mentor as king.



The left column of dots represents the inputs to each function, and the right column represents the outputs. Here, we see that zero maps the 0 and 1 inputs to the same output, 1.



By naming each of the four functions describing a possible strategy for Merlin, it's easier to later refer back to them.

The other two strategies Merlin could use are *constant functions*, meaning he gives the same answer no matter whom Nimue asks about.

Two of these functions, labeled *id* and *not* for convenience, send each of the 0 and 1 inputs to different outputs; we call these functions *balanced*. In our little game, they represent the cases in which Merlin picked exactly one person to be king.

Table 7.4. Classifying Merlin's strategies as either constant or balanced.

Merlin's strategy	Function	Type	Passes Nimue's challenge?
Choose Arthur	id	Balanced ($f(0) \neq f(1)$)	Yes
Choose Mordred	not	Balanced ($f(0) \neq f(1)$)	Yes
Choose neither	zero	Constant ($f(0) = f(1)$)	No
Choose both	one	Constant ($f(0) = f(1)$)	No

On the other hand, the functions that we label as zero and one are each *constant* functions, since they send both inputs to the same output. Constant functions then represent strategies in which Merlin is being decidedly useless, as he's either picked both to be king (a good way to start a bad war), or because he's picked neither.

Classically, if we want to know if a function is constant or balanced (whether Merlin is a bad or good mentor, respectively), we have to learn the entire function by building up its truth table. Remember, Nimue wants to ensure Merlin is a reliable mentor. If Merlin is following a strategy represented by a constant function, then he will not be a good mentor. Looking at the truth tables for the `id` and `one` functions, 7.3 and 7.4 respectively, we can see how these describe when Merlin is following a strategy that will let him be either a good or bad mentor.

Table 7.5. Truth table for the id function, an example of a balanced function.

Input	Output
True ("Should Arthur be king?")	True ("Yes")
False ("Should Mordred be king?")	False ("No")

Table 7.6. Truth table for the one function, an example of a constant function.

Input	Output
True ("Should Arthur be king?")	True ("Yes")
False ("Should Mordred be king?")	True ("Yes")

The difficulty that Nimue faces in trying to learn whether Merlin is a good or bad mentor (that is, whether f is balanced or constant) is that the quality of Merlin's mentorship is a kind of *global* property of his strategy. There's no way to look at a single output of f and conclude anything about what f would output for different inputs. If we only have access to f , then Nimue is stuck: she must reconstruct the entire truth table to decide whether Merlin's strategy is constant or balanced.

On the other hand, if we can represent Merlin's strategy as a part of a quantum program, then we can use the quantum effects we've learned about so far in the book. Using quantum computing, Nimue can learn *only* if his strategy is constant or balanced, without having to learn exactly which strategy he's using.

Since we are not interested in all the additional info the truth table provides beyond whether Merlin is a good or bad mentor, using quantum effects can help us learn what we care about more directly. With our quantum algorithm we can do this with one call of the function, and without needing to learn any additional information we are not interested in. By not asking for all of the details of the truth table, but only looking for more general properties of our function, we can best utilize our quantum resources.

The power of quantum computing

If we want to use a *classical* computer to learn whether a function is constant or balanced, we have to solve a harder problem first, namely of identifying exactly which function we have. By contrast, quantum mechanics lets us solve only the problem we care about (constant vs balanced) without solving the harder problem a classical computer has to solve.

This is an example of a pattern we'll see throughout the book, in which quantum mechanics lets us specify less powerful algorithms than we can express classically.

To do so, we will use the Deutsch–Jozsa algorithm, which uses a *single query* to our quantum representation of Merlin's strategy to learn he is a good or bad mentor. The advantage here isn't terribly practical (a savings of only one question) but that's OK, we'll see more practical algorithms later on in the book. For now, the Deutsch–Jozsa algorithm is a great place to start learning how to implement quantum algorithms, and even more importantly, to start learning what tools we can use to understand what quantum algorithms do.

7.2.2 Oracles: representing classical functions in quantum algorithms

Let's see what things look like from Nimue's quantum lake. As we plunge in for a swim, we face a somewhat immediate question: how can we implement the function f that represents Merlin's strategy with qubits? From the previous section, we saw that the classical function f is our description of a strategy that Merlin uses to play each round of Kingmaker. Since f is classical, it's easy to translate this back into a set of actions that Merlin will take: Nimue gives Merlin a single classical bit (her question), and Merlin gives Nimue a classical bit back (his answer).

To avoid meddling in the affairs of mortals, Nimue now wants to use the Deutsch–Jozsa algorithm instead. Since she lives in a quantum lake, Nimue can easily allocate a qubits to give Merlin instead. Lucky for us, Merlin knows how to communicate with qubits, but we still need to figure out what Merlin will do with Nimue's qubits in order to act on his strategy.

The trouble is that we can't pass qubits to the function f that we use to represent Merlin's strategy up above, but f takes and returns classical bits, not qubits. For Merlin to use his strategy to guide what he does with Nimue's qubits, we want to turn Merlin's strategy f into a kind of quantum program known as an *oracle*.

Conveniently for us, Merlin plays the role of an oracle pretty well.

IMPORTANT

Merlin and reversibility

From the T. H. White treatment of Merlin, we learn he lives life backwards in time. We'll represent that by making sure that everything Merlin does is *unitary*. As we saw in Chapter 2, one consequence of this is that the transformations Merlin applies are **reversible**. In particular, Merlin won't be able to measure Nimue's qubits, since measurement is not reversible. That privilege is Nimue's alone.

To understand what we need to do to model Merlin's actions as an oracle we have to figure out two things:

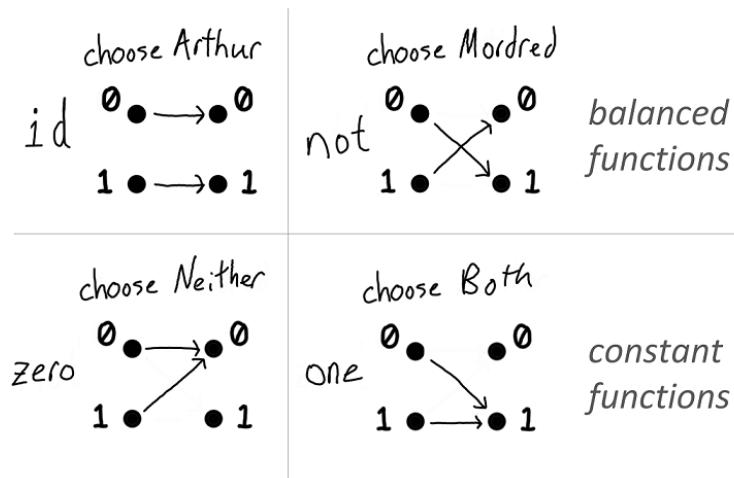
1. What transformation should Merlin apply to Nimue's qubits based on his strategy?
2. What quantum operations will Merlin need to apply to implement that transformation?

Unitary matrices and truth tables

Another way of saying what we need to do in step ① is that we need to find a *unitary matrix* that represents what Merlin does, similarly to how we used classical functions like f to represent what Merlin did when Nimue gave him classical bits. As we saw in Chapter 2, unitary matrices are to quantum computing as truth tables are to classical computing: they tell us what the effect of a quantum operation is for every possible input. Once we found the right unitary, step ② is where we'll figure out what sequence of quantum operations we can do that will be described by that unitary.

To complete step ①, we need to turn functions like f into unitary matrices, so let's start by recapping what f can be. The possible strategies Merlin could use are represented by each of the functions `id`, `not`, `zero`, and `one` (see [7.4](#)).

Figure 7.4. Four different functions from one bit to one bit.



For the two balanced functions, `id` and `not` in [7.4](#), we can answer the first question easily. Quantum programs for `id` and `not` can be implemented as rotation operations, so it is easy to turn them into quantum operations. The quantum NOT operation, for instance, is a rotation of 180° around the X axis, exchanging the $|0\rangle$ and $|1\rangle$ states with each other.

TIP

Recall from Chapter 4 that the quantum operation X , represented by the unitary matrix $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, is used to apply a rotation of 180° around the X axis. This operation implements a quantum NOT: since $X|0\rangle = |1\rangle$ and $X|1\rangle = |0\rangle$, we can write using the \neg (NOT) operator from Chapter 2 as $X|x\rangle = |\neg x\rangle$.

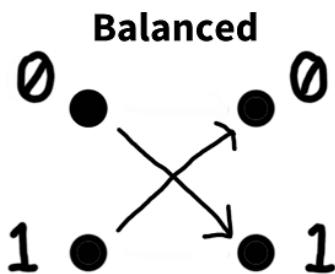
While any rotation can be undone by rotating the same amount in the opposite direction, we run into more problems with the constant functions zero and one. Neither zero nor one can be implemented directly as rotations, so we have a bit more work to do. For instance, if f is zero, then the outputs $f(0)$ and $f(1)$ are both 0. If we only have the output 0, we cannot tell whether we got that output from giving f a 0 or a 1 as input (see [7.5](#)).

NOTE

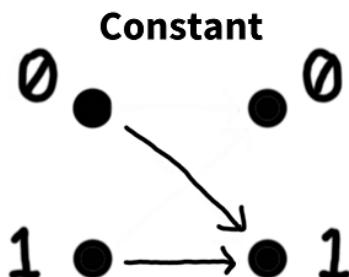
Once we apply zero or one, we have lost any information about the input.

Figure 7.5. Why can't we reverse the constant zero or one functions?

Example 1: We can see with the way the inputs and outputs are mapped that if we know what the output of the function, then we can trace it back to what the input was!



Example 2: We can see that with constant functions that though we know the output, it is ambiguous what was the input. It could have been either 0 or 1 but we can't know... ☺



Since one and zero are both irreversible, and valid operations on qubits are reversible,

Merlin will need another way of representing functions like f in quantum algorithms such as the one for Nimue's challenge. On the other hand, if we can represent Merlin's strategy by a reversible classical function instead of f , it will be much easier to write down a quantum representation of his strategy.

Our strategy for representing classical functions as quantum oracles

1. Find a way to represent our irreversible classical function by a reversible classical function
2. Write down a transformation on quantum states using our reversible classical function
3. Figure out what quantum operations we can do that result in that transformation

Let's use the tried and true approach of guessing and checking to see if we can design a valid reversible classical function! The easiest way to figure out whether we were given a 0 or a 1 as an input is to just record it somewhere, so let's make a new function that returns two bits instead of one:

First attempt: Record and keep the input

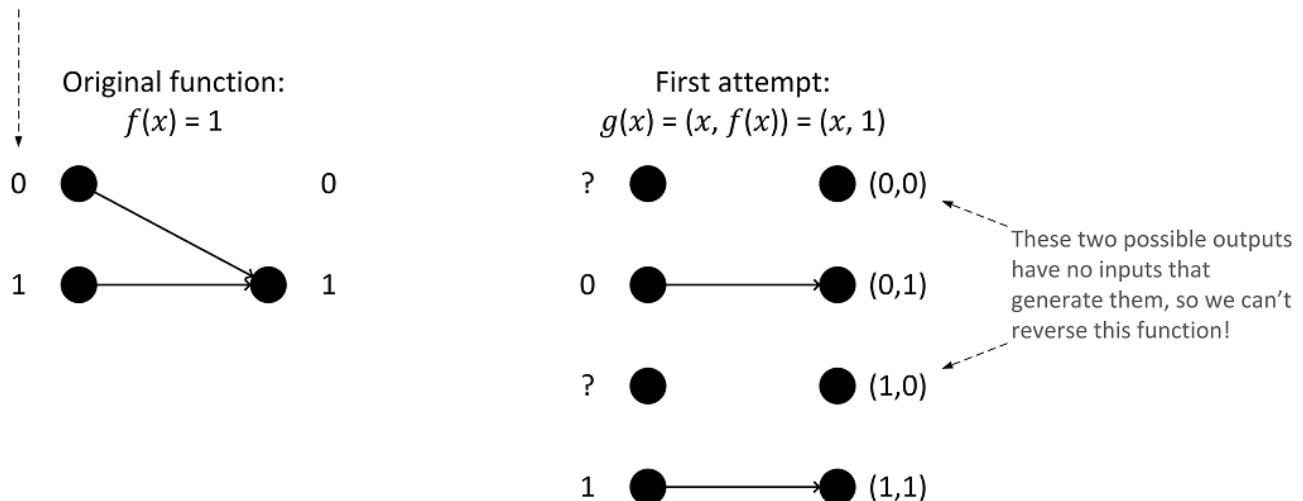
$$g(x) = (x, f(x))$$

For example, if Merlin uses the strategy one (that is, says "yes" to Nimue no matter what she asks), then $f(x) = 1$, and $g(x) = (x, f(x)) = (x, 1)$.

This gets a lot closer, since we can now tell whether we started with a 0 or 1 input, but we're not quite there, since g has two outputs and one input (see [7.6](#)).

Figure 7.6. First attempt: keeping input and output with $g(x)$.

As before, we'll represent functions by drawing arrows from possible inputs to the corresponding outputs. The one function, for example, sends both 0 and 1 to 1.



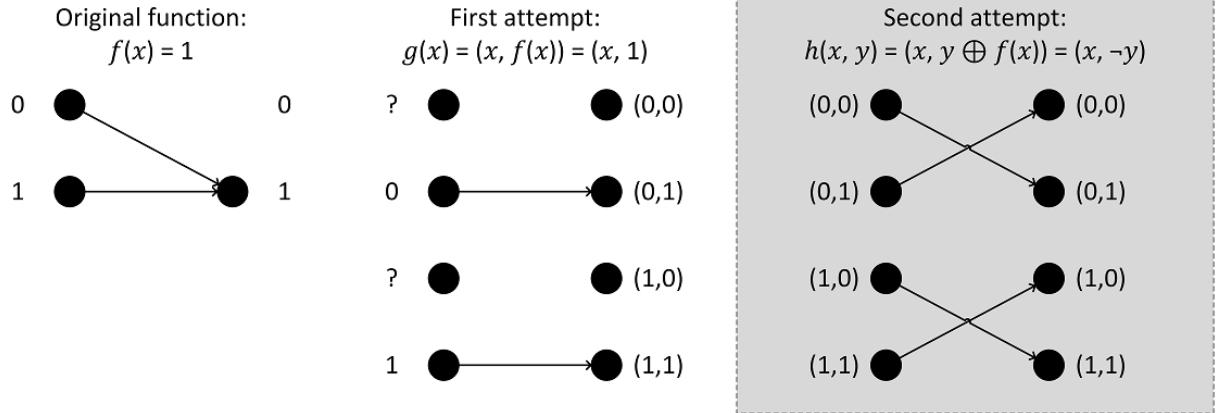
To use g as a strategy, Merlin would have to give back Nimue more qubits than she gave him, but she is the keeper of both swords and qubits. More technically, reversing g would thus destroy information, as it would take two inputs and return one output!

Trying one more time, let's define a new classical function h that takes two inputs and returns two outputs, $h(x, y)$. Let's again consider the example where we describe Merlin's strategy by the function $f(x) = 1$. Since g got us nearly there, we'll choose h such that $h(x, 0) = g(x)$. We saw from our first attempt that when Merlin uses the strategy $f(x) = 1$, then $g(x) = (x, 1)$, so we have that $h(x, 0) = (x, 1)$. Now we just need to define what happens when we pass $y = 1$ to h . If we want h to be reversible, we need that $h(x, 1)$ is assigned to something other than $(x, 1)$. One way to do this is to let $h(x, y) = (x, \neg y)$, so that $h(x, 1) = (x, 0) \neq (x, 1)$.

This choice is especially convenient since applying h twice gets us back our original input, $h(h(x, y)) = h(x, \neg y) = (x, \neg \neg y) = (x, y)$.

Figure 7.7. Second attempt: $h(x, y)$ which is reversible and has the same number of inputs and outputs!

Hurray! By having 2 input bits and 2 output bits our function is reversible. Here, that means we can map every output to a unique input, and vice versa.



Now that know how to make a *reversible* classical function from each strategy, let's finish by making a quantum program from our reversible function. In the case of one, we saw that h flips its second input, $h(x, y) = (x, \neg y)$. Thus, we can write a quantum program that does the same thing as our reversible classical function simply by flipping the second of two input qubits. As we saw in Chapter 4, we can do this using the X instruction, since $X|x\rangle = |\neg x\rangle$.

Generalizing our results

More generally, we can make a reversible quantum operation, in precisely the same way we made reversible classical functions h by flipping an output bit based on the output of the irreversible function f . We can define the unitary matrix (that is, the quantum analogue of a truth table) U_f for f for each input state in exactly the same way.

Figure 7.8. Constructing reversible classical functions and unitary matrices from irreversible classical functions

$$\underline{h(x, y) = (x, y \oplus f(x))}$$

We can make a new reversible classical function h from an irreversible function f by flipping a bit based on the output of f .

To define h , we specify what it does for arbitrary classical bits x and y .

$$\underline{U_f |x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle}$$

In exactly the same way, we can define a unitary matrix U_f .

Much like we defined h by saying what it did for each classical bit x and y , we can say what U_f does for input qubits in the labeled by classical bits; that is, in the $|0\rangle$ and $|1\rangle$ states.

Defining U_f in this way makes it easy to undo the call to f , since applying U_f twice gives us the identity $\mathbb{1}$ (that is, the unitary matrix for the "do nothing" instruction). When we define a unitary in this way by applying a function f conditionally to the labels for qubit states, we call this new operation an *oracle*.

Definition 7.3: Oracle

An oracle is a quantum operation represented by a unitary matrix U_f that transforms its input state as

$$U_f |x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle.$$

The symbol \oplus represents the exclusive or operator from regular boolean logic.

Now all that's left is to figure out what sequence of instructions we need to send to implement each unitary U_f . We've seen above how what instructions we need to implement an oracle for one, namely an X instruction on the second qubit. Now let's look at how to write oracles for other possible functions f . That way, Merlin will know what he should do no matter what his strategy is.

DEEP DIVE: Why is it called an oracle?

So far, we've seen a few different examples of the kind of whimsical naming that quantum computing owes to its physics history, like "bra," "ket," and "teleportation." It's not just physicists who like to have a bit of fun, though! One branch of theoretical computer science called *complexity theory* explores what is possible to do efficiently even in principle given different kinds of computing machines. You may have heard, for instance, of the " P versus NP " problem, a classic conundrum in complexity theory that asks whether problems in P are as difficult to solve as those NP or not. The complexity class P is the group of questions for which there exists a way to answer them with an algorithm that takes polynomial time. By contrast, NP is the group of questions for which we can check a potential answer in polynomial time, but we don't know if we could come up with an answer from scratch in polynomial time.

Many other problems in complexity theory are posed by introducing small games or stories to help researchers remember what definitions to use where. Our own little story about Merlin and Nimue is a nod to this tradition. In fact, one of the most celebrated stories in quantum computing is called *MA* for "Merlin–Arthur." Problems in the class *MA* are thought of using a story in which Arthur gets to ask an Merlin, an all-powerful but untrustworthy wizard, some set of questions. A yes/no decision problem is in *MA* if, whenever the answer is "yes," there exists a proof that Merlin can give Arthur and that Arthur can easily check using a *P* machine and a random number generator.

The name "oracle" fits into this kind of storytelling, in that any complexity class *A* can be turned into a new complexity class *A*^b by allowing *A* machines to solve a **B** problem in a single step, as though they were consulting an oracle. Much of the history of problems like the Deutsch–Jozsa problem stems from trying to understand how quantum computing affects computational complexity, so many of the naming conventions and much of the terminology has been adopted into quantum computing itself.

For more on complexity theory and how it relates to quantum computing, black holes, free will, and Greek philosophy, check out *Quantum Computing Since Democritus* (ISBN: 978-0521199568).

In general, finding a sequence of instructions by starting from a unitary matrix is a mathematically difficult problem known as *unitary synthesis*. That said, in this case, we can figure it out by substituting each of Merlin's strategies *f* into our definition for U_f and identifying what instructions will have that effect — we can guess and check in the same way that we did to turn the one function into an oracle.

Let's try this out for the zero function:

Exercise 7.1: Try out writing an oracle!

What would the oracle operation (U_f) be for *f* if *f* was zero?

Let's work it out one step at a time.

- From the definition for U_f , we know that $U_f |xy\rangle = |x\rangle |y \oplus f(x)\rangle$.
- Substituting in zero for *f*, $f(x) = 0$, we get that $U_f |xy\rangle = |x\rangle |y \oplus 0\rangle$.
- We can use that $y \oplus 0 = y$ to simplify this a little bit further, getting that $U_f |xy\rangle = |x\rangle |y\rangle$.
- At this point, we notice U_f does nothing to its input state, so we can implement it by... doing nothing.

The function $f = \text{id}$ is slightly more subtle than the zero and one cases because $y \oplus f(x)$ cannot be simplified to not depend on *x*. As summarized in 7.5, we need $U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle = |x\rangle |y \oplus x\rangle$. That is to say, we need the action of the oracle on the input state ($|x\rangle |y\rangle$) to leave *x* alone and replace *y* with the exclusive or of *x* and *y*.

Another way to think of this output is to recall that $y \oplus 1 = \neg y$, so that when $x = 1$, we need to flip *y*. This is precisely how we defined the controlled-NOT (CNOT) instruction in Chapter 5, so we recognize that when *f* is *id*, U_f can be implemented by applying a CNOT.

This leaves us with how to define the oracle for $f = \text{not}$. Just as the oracle for id flips the output (target) qubit when the input (control) qubit is in the $|1\rangle$ state, the same argument gives us that we need our oracle for not to flip the second qubit when the input qubit is in $|0\rangle$. The easiest way to do this is to first flip the input qubit with an X instruction, apply a CNOT instruction, and then undo the first flip with another X .

To review all of the oracles we have learned how to define, we collected all our work in the above section in [7.5](#).

Table 7.7. Table Oracle outputs for each one-bit function f

Function Name	Function	Output of Oracle
zero	$f(x) = 0$	$ x\rangle y\oplus 0\rangle = x\rangle y\rangle$
one	$f(x) = 1$	$ x\rangle y\oplus 1\rangle = x\rangle \neg y\rangle$
id	$f(x) = x$	$ x\rangle y\oplus x\rangle$
not	$f(x) = \neg x$	$ x\rangle y\oplus \neg x\rangle$

The uncompute trick: Turning functions into quantum oracles

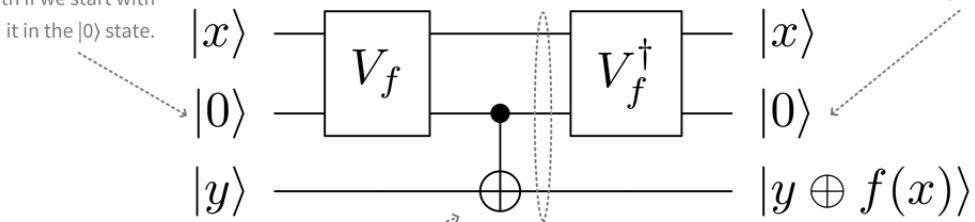
As it stands, it may seem as though it takes a lot of work to design U_f for each function f . Thankfully, there's a nice trick that lets us build an oracle, starting with a somewhat simpler requirement.

Recall that above, we attempted to define a reversible version of f by returning $(x, f(x))$ as output when given $(x, 0)$ as input. Similarly, suppose that we are given a quantum operation V_f that correctly transforms $|x\rangle|0\rangle$ to $|x\rangle|f(x)\rangle$. Then, we can always make an oracle U_f by using one additional qubit, and by calling V_f twice using a technique called the "uncompute trick," as shown in [7.9](#).

Figure 7.9. Using the "uncompute trick" to turn an operation that only works when you add an extra $|0\rangle$ input qubit to an operation that can be used as an oracle.

The *uncompute trick*: If we add another input $|0\rangle$, then an operation which gives us a way to make the operation from $|x\rangle|y\rangle$ to $|x\rangle|f(x)\rangle$ is reversible! This works for any function $f(x)$.

The first step in the uncompute trick is to use a temporary qubit along with the circuit from attempt 2. It is easiest for the math if we start with it in the $|0\rangle$ state.



Since the information $f(x)$ is classical, we can then use a CNOT to copy this information onto the $|y\rangle$ register.

We can apply the inverse of the circuit above to get rid of the memory leak, cleaning up the temporary register we used.

The state here would be: $|x\rangle|f(x)\rangle|y \otimes f(x)\rangle$. This is a problem because the second qubit leaks information about our system (it's an extra qubit we will recycle after this, so like a hard drive we should return it blank).

At the end of the uncompute trick, our temporary register doesn't "remember" anything about x , so we don't have a memory leak!

It's straightforward at this point to check that our new circuit is reversible — it's actually its own inverse, and is of the same form as the other oracles in this chapter.

While this doesn't especially help in the case of Deutsch–Jozsa, it shows us that the concept of an oracle is a very general one, as it's often much easier to start with an operation of the form V_f .

NOTE

The oracle construction also works for multiple qubit functions.

As a thought exercise, if we have a function $f(x_0, x_1) = x_0$ and x_1 , how would the oracle U_f transform an input state $|x_0 x_1 y\rangle$? We'll see in later chapters how to code up this oracle.

We've thus used the oracle representation to solve the problem that functions like zero and one cannot be represented as rotations. With that dealt with, we can continue on to actually write the rest of the algorithm that Nimue uses to challenge Merlin.

Deep Dive: Other ways to represent functions as oracles

This isn't the only way we could have defined U_f . Merlin could have also flipped the sign of Nimue's input x when $f(x)$ is one,

$$U_f|x\rangle = (-1)^{f(x)}|x\rangle$$

This turns out to be a more useful representation in some cases, such as in gradient descent algorithms. These algorithms are common in machine learning, and minimize functions by searching along directions in which a function changes the fastest. For more information, see Chapter 4.10 of *Groking Deep Learning*.

Picking the right way for a particular application to represent classical information such as subroutine calls within a quantum algorithm is a part of the art of quantum programming. For now, we will use the definition of "oracle" introduced above.

With the oracle representation of f in hand, the first few steps of the Deutsch–Jozsa algorithm can be written out in the same sort of pseudocode that we used to write quicksort earlier:

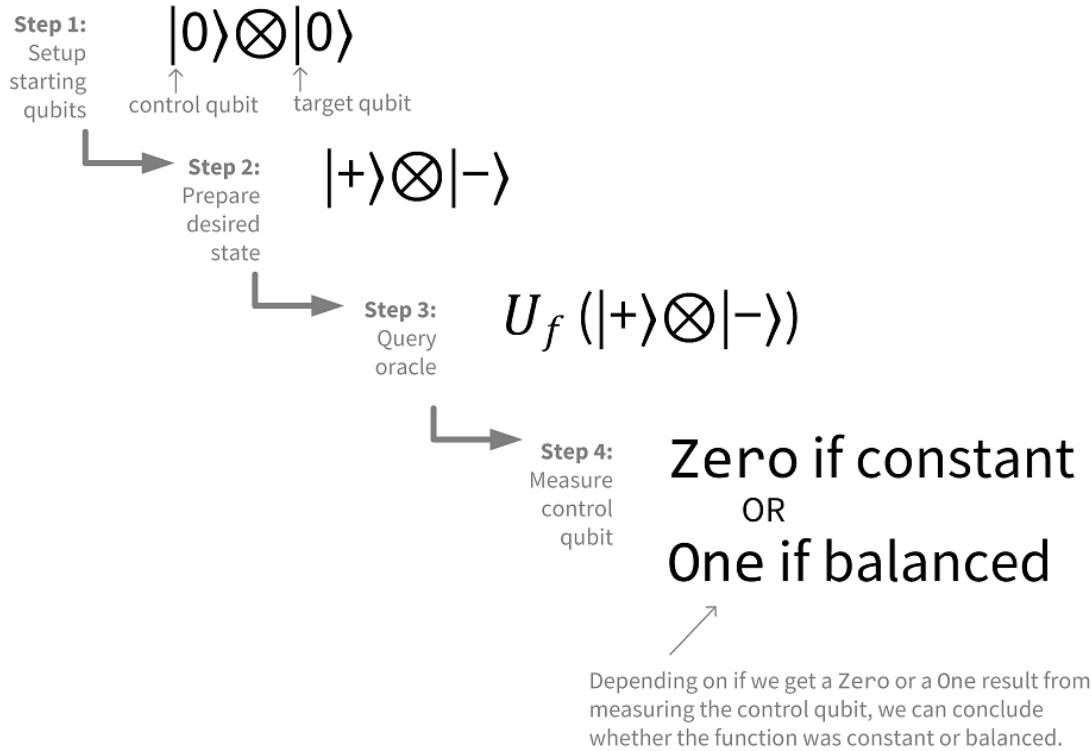
Deutsch–Jozsa Algorithm

1. Prepare two qubits labeled **control** and **target** in the $|0\rangle \otimes |0\rangle$ state.
2. Apply operations to the **control** and **target** qubits to prepare the following state: $|+\rangle \otimes |-\rangle$.
3. Apply the oracle U_f to the input state $|+\rangle \otimes |-\rangle$. Recall that $U_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$.
4. Measure the **control** qubit in the X basis; if we observe a 0, then the function is constant, otherwise the function is balanced.

TIP

Measuring a qubit in the X basis always returns a 0 or 1 just like if we measured in the Z basis. Recall from Chapter 3 that if the state of the qubit is $|+\rangle$, we always get a 0 when we measure in the X basis, while we always get a 1 if the qubit is in $|-\rangle$.

Let's look at a figure illustrating these steps:

Figure 7.10. Steps to the Deutsch–Jozsa Algorithm.

We'll see at the end of the chapter **why** this algorithm works, but let's jump right in start programming it up. To do so, we'll use the Q# language provided by the Quantum Development Kit, since this will make it much easier for us to see the structure of a quantum algorithm from its source code.

7.3 Simulating the Deutsch–Jozsa algorithm in Q#

We tried out in chapter 6 passing operations as arguments in Q# programs. We can use the same approach of passing operations as inputs to pass oracles to help us predict how Nimue's challenge will turn out. To do so, recall that there are four possible functions that we can consider for this problem, each representing a possible strategy that Merlin could use, as shown in 7.6.

Table 7.8. Representing one-bit functions as two-qubit oracles.

Function Name	Function	Output of Oracle	Q# Operation
zero	$f(x) = 0$	$ x\rangle y\rangle \oplus 0\rangle = x\rangle y\rangle$	NoOp(control, target);
one	$f(x) = 1$	$ x\rangle y\rangle \oplus 1\rangle = x\rangle \neg y\rangle$	X(target);
id	$f(x) = x$	$ x\rangle y\rangle \oplus x\rangle$	CNOT(control, target)
not	$f(x) = \neg x$	$ x\rangle y\rangle \oplus \neg x\rangle$	X(control), CNOT(control, target); X(control);

If we represent each function $f(x)$ by an oracle (quantum operation) that maps $|x\rangle|y\rangle$ to $|x\rangle|y\rangle \oplus f(x)$, then we can identify each of the functions zero, one, id, and not from [7.4](#).

Each of the four oracles above translates immediately into Q#:

Listing 7.3. Oracles.qs: Q# operations for each Deutsch–Jozsa oracle

```
namespace DeutschJozsa {
    open Microsoft.Quantum.Intrinsic;

    operation ZeroOracle(control : Qubit, target : Qubit) : Unit {
    }

    operation OneOracle(control : Qubit, target : Qubit) : Unit {
        X(target);
    }

    operation IdOracle(control : Qubit, target : Qubit) : Unit {
        CNOT(control, target);
    }

    operation NotOracle(control : Qubit, target : Qubit) : Unit {
        X(control);
        CNOT(control, target);
        X(control);
    }
}
```

Can't We Just Look at the Source Code?

In `Oracles.qs`, we wrote the source code out for each of the four single-qubit oracles `ZeroOracle`, `OneOracle`, `IdOracle`, and `NotOracle`. Looking at that source code, we can tell whether each is constant or balanced without having to call it at all, so why should we worry with the Deutsch–Jozsa algorithm? Thinking from Nimue's perspective, she doesn't necessarily have the source code that Merlin uses to apply operations to her qubits. Even if she does, Merlin's ways are inscrutable, such that she may not be able to easily predict what Merlin does even given the source code that he uses.

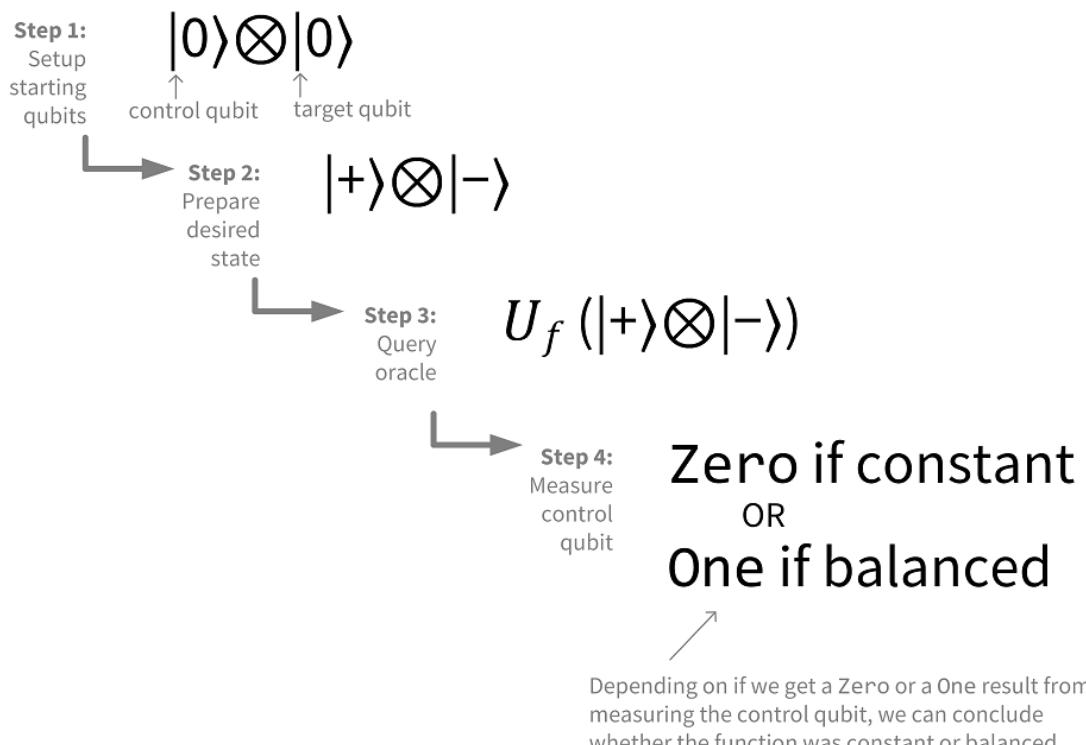
Practically speaking, while it's hard to obfuscate a two-qubit oracle all that much, the Deutsch–Jozsa algorithm demonstrates a technique that is useful more generally. For example, we might have access to the source code for some operation, but it is a mathematically or computationally difficult problem to extract the answer to a question about that operation. All cryptographic hash functions have this property

by design, whether they're used to ensure that a file has been downloaded correctly, an application has been signed by a developer, or as a part of growing a blockchain through mining for collisions.

We'll see an example in Chapter 10 where we can use techniques such as those developed in the Deutsch–Jozsa algorithm to more quickly ask questions about such functions.

With these oracles now implemented in Q#, we can write the entirety of the Deutsch–Jozsa algorithm (as well Nimue's strategy for Kingmaker)!

Figure 7.11. Steps to the Deutsch–Jozsa Algorithm.



Listing 7.4. Algorithm.qs: Q# operation implementing the Deutsch–Jozsa algorithm

```
operation IsOracleBalanced(
    oracle : ((Qubit, Qubit) -> Unit)
) : Bool {
    using ((control, target) = (Qubit(), Qubit())) {
        H(control);
        X(target);
        H(target);

        oracle(control, target);
    }
}
```

```

    H(target);
    X(target);

    return MResetX(control) == One;
}
}

```

④ ⑤

- ① This using statement asks the target machine to give us two qubits, control and target, each starting in the $|0\rangle$ state.
- ② These lines prepare the input state $|+-\rangle = (|00\rangle - |01\rangle + |10\rangle - |11\rangle) / \sqrt{2}$ on control and target, as shown in Step ① of 7.11.
- ③ Here, we call the oracle that we were given as the input argument oracle. Note that the oracle is only called once!
- ④ We know that the target qubit is still in $|-\rangle$, so we can undo the `X(target); H(target);` sequence of operations above to reset the target qubit.
- ⑤ Finally, we can measure whether the control qubit is in $|+\rangle$ or $|-\rangle$, corresponding to Zero or One results in the X basis. Similarly to the `MResetZ` operation provided by the Q# standard libraries, the `MResetX` operation performs the desired X -basis measurement and resets the measured qubit to $|0\rangle$.

Now as we want to make sure our implementation is good, so let's test it!

Measurement results in Q#

We've now seen both the `MResetX` and `MResetZ` operations in Q#, which measure and reset a qubit in the X and Z bases, respectively. Both of these operations return a `Result` value, which seems a little bit confusing at first. After all, an X basis measurement tells us whether we were in the $|+\rangle$ or the $|-\rangle$ state, so why does Q# use the labels Zero and One?

Table 7.9. Conventions used for X – and Z – basis measurement results in Q#.

Result value	X basis	Z basis
Zero	$\langle + $	$\langle 0 $
One	$\langle - $	$\langle 1 $

We'll see in more detail on this later, but the short version is that a value of type `Result` tells us how many about how many phases of (-1) are applied to a state by different instructions. For example, $Z|1\rangle = -|1\rangle = (-1)^1 |1\rangle$, while $X|-\rangle = (-1)^1 |-\rangle$. Since in both cases, we raise (-1) to the power of 1, $|1\rangle$ and $|-\rangle$ are assigned to the One result when we measure in the Z and X bases, respectively. Similarly, since $Z|0\rangle = (-1)^0 |0\rangle$, we assign $|0\rangle$ to the Zero result when we measure Z .

We said earlier that Nimue would like to learn as little as she can about the affairs of humankind. Thankfully, she only asks Merlin to do something with her qubits once, where we call `oracle(control, target)`. Nimue only gets one classical bit of information out, from the call to `MResetX`, which isn't enough for her to tell the

difference between the `id` strategy (Merlin selects Arthur to mentor as king) and the `not` strategy (Merlin selects Mordred to mentor).

To make sure that she can still learn what she actually cares about, whether Merlin's strategy is constant or balanced, we can use the `Fact` function provided with the Q# standard libraries to test that our implementation works. `Fact` takes two boolean variables as the first two arguments, checks to see if they are equal, and if not issues a message.

TIP Later, we'll see how to use these assertions to write unit tests for quantum libraries.

Listing 7.5. `Algorithm.qs`: Q# operation testing that the Deutsch–Jozsa algorithm works as intended

```
operation RunDeutschJozsaAlgorithm() : Unit {
    Fact(not IsOracleBalanced(ZeroOracle), "Test failed for zero oracle."); ①
    Fact(not IsOracleBalanced(OneOracle), "Test failed for one oracle."); ②
    Fact(IsOracleBalanced(IdOracle), "Test failed for id oracle.");
    Fact(IsOracleBalanced(NotOracle), "Test failed for not oracle.");

    Message("All tests passed!"); ③
}
```

- ① This line runs the Deutsch–Jozsa algorithm for the case in which Merlin uses the zero strategy. We pass the oracle for zero as the `ZeroOracle` operation that we wrote above. Since zero isn't a balanced function, we expect `IsOracleBalanced(ZeroOracle)` to return `false` as its output; this expectation can be checked using the `Fact` function.
- ② We can do exactly the same thing for the one strategy, this time calling `IsOracleBalanced(OneOracle)` instead.
- ③ If all four assertions passed, then we can be sure that our program for the Deutsch–Jozsa algorithm works regardless of which strategy Merlin uses.

If we run this using the `%simulate` magic command, we can confirm that by using the Deutsch–Jozsa algorithm, Nimue is able to learn exactly what she wanted to learn about Merlin's strategy.

```
In [ ]: %simulate RunDeutschJozsaAlgorithm
All tests passed!
```

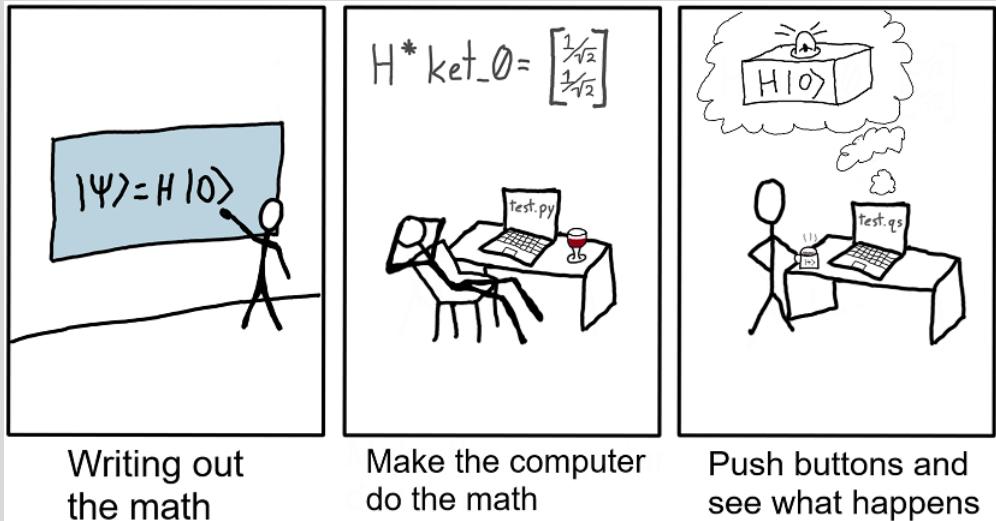
7.4 Exploring the Deutsch–Jozsa Algorithm by example

Now that we've implemented the Deutsch–Jozsa algorithm for ourselves, it's helpful to take a step back and see how it works, as we'll learn some valuable quantum programming techniques along the way.

Trying things out!

In Chapters 2 and 4, we learned to use NumPy and QuTiP to simulate how the states of qubits are transformed when we send instructions to a quantum computer. We effectively used these packages to do math for us to find out what would happen to our quantum states. This is like the "make the computer do the math" approach in [7.12](#).

Figure 7.12. Three different approaches of learning how a quantum program or algorithm works.



Now that we're programming larger algorithms in Q#, we can use both the what we learned before and a bit of the right box (push all the buttons!) to help us predict what a particular operation will do.

These three approaches when used together are powerful problem solving tools when learning quantum programming! If you get stuck using one approach you can always try another to see if that helps.

Recall the four steps to Deutsch–Jozsa algorithm in [7.1](#) as we just programmed in the previous section.

Listing 7.6. The four steps in the Deutsch–Jozsa algorithm.

```
// 1. Prepare the input state |+->.
H(control);
X(target);
H(target);

// 2. Apply the oracle.
oracle(control, target);

// 3. Undo the preparation on the target qubit.
H(target);
X(target);

// 4. Finally, we measure in the X basis.
```

```
set result = MResetX(control);
```

The key to understanding how the Deutsch–Jozsa algorithm works is understanding the step where we call the oracle, `oracle(control, target)`. Before we can get to that, though, we need to understand step 1, where we prepare our input to `oracle`.

7.4.1 Step 1. Preparing the input state for Deutsch–Jozsa

Let's use Python to try and understand what is happening when we prepare our $|+-\rangle$ state. The operations we used to prep the input state in Q# were:

Listing 7.7. State prep for the Deutsch–Jozsa algorithm.

```
H(control);
X(target);
H(target);
```

Each of the operations applied above are single qubit gates, so we can consider what happens to each qubit independently. Let's look at what happens to the control qubit after the Hadamard operation.

Listing 7.8. Using QuTiP to model the control qubit state preparation.

```
>>> import qutip as qt
>>> H = qt.hadamard_transform()                                     ①
>>> H
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 0.70710678  0.70710678]
 [ 0.70710678 -0.70710678]]                                         ②
>>> control_state = H * qt.basis(2, 0)                            ③
>>> control_state
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket          ④
Qobj data =
[[0.70710678]
 [0.70710678]]
```

- ➊ While `H` in Q# is an instruction, `hadamard_transform` in QuTiP gives us a unitary matrix that we can use to simulate how the `H` instruction transforms states.
- ➋ $1/\sqrt{2} \approx 0.707$, so this output tells us that $H = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} / \sqrt{2}$.
- ➌ In QuTiP, we can get the vector for the $|0\rangle$ state by calling `basis(2, 0)`. The 2 tells QuTiP we want a qubit (necessary dimension of $|0\rangle$), while the 0 tells us we want the state to have the value $|0\rangle$. Since $|+\rangle = H|0\rangle$, this sets `control_state` to be $|+\rangle$.
- ➍ Using that $1/\sqrt{2} \approx 0.707$, we read this as telling us that $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$.

So that's a pretty easy one, the control qubit is now in the $|+\rangle$ state. Now let's look at preparing the target qubit in [7.2](#).

Listing 7.9. Using QuTiP to model the target qubit state preparation.

```
>>> target_state = H * (qt.sigmax() * qt.basis(2, 0))           ①
>>> target_state
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket      ②
Qobj data =
[[ 0.70710678]
 [-0.70710678]]
```

- ① We repeat the same H operation as before, but this time on $X|0\rangle = |1\rangle$.
- ② QuTiP tells us that $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$. That is, the same as $|+\rangle$, but with the sign of $|1\rangle$ flipped.

Now that we have seen how we prepare each qubit, let's have QuTiP help us write the state of our input *register*.

Listing 7.10. Using QuTiP to model the combined input state.

```
>>> register_state = qt.tensor(control_state, target_state)        ①
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket ②
Qobj data =
[[ 0.5]
 [-0.5]
 [ 0.5]
 [-0.5]]
```

- ① As we saw in Chapter 5, we combine the states of different qubits to get the state of an entire register of qubits using the `tensor` function.
- ② QuTiP tells us that $|+\rangle \otimes |-\rangle = |+-\rangle = (|00\rangle - |01\rangle + |10\rangle - |11\rangle) / 2$. That is, we have an equal superposition over all four possible computational basis states, with a minus sign in front of computational basis states where the target qubit is in the $|1\rangle$ state.

NOTE

As we saw in Chapter 5, when writing down the state of a multi-qubit system, tensor products can get a little bit verbose. Thus, we'll often write down multi-qubit states like $|0\rangle \otimes |1\rangle$ by concatenating their labels inside a single ket, as in $|01\rangle$. Similarly, $|+-\rangle$ is the same as $|+\rangle \otimes |-\rangle$.

7.4.2 Step 2. Applying the oracle

Having prepared our input, let's get back to the core of the Deutsch–Jozsa algorithm, where we call into our oracle [7.3](#).

Listing 7.11. Applying the oracle in the Deutsch–Jozsa algorithm.

```
oracle(control, target);
```

In the same way that we can understand operations like $H(\text{control})$ by writing down the state of the control qubit and applying the unitary operator H to that state, we can

understand what the oracle U_f does by analyzing its action on the state we pass to it.

Our quantum oracle operates on two qubits, which raises a question of how we should interpret each of those qubits. In the original classical case, the interpretation of the input and output classical bits from f was clear: Nimue asked a one-bit question and got a one-bit answer. To understand what each qubit does for us, recall that we saw that when we went to a reversible classical function that we also needed two inputs: the first input acted like the question we asked in the irreversible case, while the second input gave us somewhere to put the answer (see [7.13](#) for a reminder).

Figure 7.13. Constructing reversible classical functions and unitary matrices from irreversible classical functions

$$h(x, y) = (x, y \oplus f(x))$$

We can make a new reversible classical function h from an irreversible function f by flipping a bit based on the output of f .

To define h , we specify what it does for arbitrary classical bits x and y .

$$U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$$

In exactly the same way, we can define a unitary matrix U_f .

Much like we defined h by saying what it did for each classical bit x and y , we can say what U_f does for input qubits in the labeled by classical bits; that is, in the $|0\rangle$ and $|1\rangle$ states.

We can roughly think of the oracle in the same way: the first qubit (control in the snippet above) represents our question, while the second qubit (target) gives us somewhere for Merlin to apply his answer. This interpretation makes sense when control starts off in either the $|0\rangle$ or $|1\rangle$ state, but how can we interpret the case above where we pass qubits in the $|+\rangle$ state to the oracle? In this case, our control qubit starts off in the $|+\rangle$ state, but $f(+)$ doesn't make any sense. Since f is a classical function, its input has to be either 0 or 1... we can't pass + to the classical function f . It may seem like we would be at a dead end, but thankfully there's a way to figure it out.

Quantum mechanics is linear, which means we can always understand what a quantum operation does by breaking it down into its action on a representative set of states.

TIP

As we saw in Chapter 2, a set of states that can be used this way is called a *basis*.

To understand what our oracle does when the control qubit is in the $|+\rangle$ state, we can use the fact that $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ to break the oracle's action down into what it does to $|0\rangle$ plus its action on $|1\rangle$, then sum both parts back together (making sure to divide by $\sqrt{2}$ at the end). This helps because instead of being confused trying to understand what " $f(+) =$ " means, we can reduce the action of U_f to cases we do know how to compute

like $f(0)$ and $f(1)$!

TIP
Computational basis states

Expanding the action of a quantum operation in terms of how it acts on $|0\rangle$ and $|1\rangle$ is very common in quantum programming. Given how useful this is, we often give a special name to these two input states, and call $|0\rangle$ and $|1\rangle$ the **computational basis** to set it apart from other bases we might use, like $|+\rangle$ and $|-\rangle$.

Using linearity to understand quantum operations isn't limited to a single qubit, as we'll see in the rest of the chapter. For two qubits, for instance, the computational basis for two qubits then consists of the states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$.

If we have even more (say, five) qubits, we can write out states like

$|1\rangle \otimes |0\rangle \otimes |0\rangle \otimes |1\rangle \otimes |0\rangle$ as strings in the same way, getting $|10010\rangle$. We can write down the computational basis for five qubits in the same way as $\{|00000\rangle, |00001\rangle, |00010\rangle, \dots, |11110\rangle, |11111\rangle\}$.

More generally, if we have n qubits, then the computational basis consists of all strings of n classical bits, each as the label of a ket. Put differently, the computational basis for a multiple qubit system is made up of all tensor products of $|0\rangle$ and $|1\rangle$; that is, all states labelled by a string of classical bits.

With this approach of breaking down how the oracle works, let's look at some examples of the oracles we have implemented earlier.

Example 1: id oracle

Suppose we were given an oracle that implemented the strategy where Merlin chooses Arthur as king. Recall the classical one bit function that represents this strategy is `id`.

Table 7.10. Representing one-bit functions as two-qubit oracles

Function Name	Function	Output of Oracle	Q# Operation
id	$f(x) = x$	$ x\rangle y \oplus x\rangle$	CNOT(control, target)

From [7.7](#), we know that this means that U_f is implemented by the CNOT instruction, so let's see what that does to `register_state`.

TIP

Recall that the controlled NOT instruction flips its second qubit if the first qubit is in $|1\rangle$.

Listing 7.12. Using QuTiP to compute how the id oracle transforms its input state.

```
>>> cnot = qt.cnot(2, 0, 1)                                     1
>>> cnot
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
Qobj data =
[[1. 0. 0. 0.]                                                 2
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]]                                                 3
```

```
[0. 0. 1. 0.]]
>>> register_state = cnot * register_state
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.5]
 [-0.5]
 [-0.5]
 [ 0.5]]
```

④

- ➊ We can ask QuTiP for a matrix that will let us simulate the CNOT instruction by using the `cnot` function. Here, the `2` indicates that we want to simulate CNOT on a two-qubit register, the `0` indicates that the 0th qubit is our control, and the `1` indicates that the 1st qubit is our target.
- ➋ Remember that unitary operators are for quantum computing what truth tables are for classical logic. Each row in this table tells us what happens to a computational basis state.
- ➌ For example, this row `2` (zero-indexed), which we write as `10` in binary. Thus, this row is the vector that we'll get out if our input is $|10\rangle$, and tells us that the CNOT instruction leave our qubits in $|11\rangle$ (3 in decimal, hence there's a `1` in the third column).
- ➍ QuTiP tells us that the register with our control and target qubits is now in the state $(|00\rangle - |01\rangle - |10\rangle + |11\rangle) / \sqrt{2}$.

Now that we have worked out the action of the `id` oracle, let's look at what the `not` oracle does to our input state.

Example 2: the `not` oracle

Let's repeat the analysis using the `not` oracle, the other balanced function. The oracle representing Merlin choosing Mordred is implemented with a series of `X` and `CNOT` operations as in 7.8.

Table 7.11. Representing the one-bit function `not` as a two-qubit oracle.

Function Name	Function	Output of Oracle	Q# Operation
<code>not</code>	$f(x) = \neg x$	$ x\rangle y\rangle \oplus \neg x$	<code>X(control); CNOT(control, target); X(control);</code>

Let's now jump to Python to see how to break down the operation of the `not` oracle.

Listing 7.13. Using QuTiP again, now with the `not` oracle.

```
>>> control_state = H * qt.basis(2, 0)
>>> target_state = H * qt.basis(2, 1)
>>> register_state = qt.tensor(control_state, target_state)
>>> I = qt.qeye(2)
>>> X = qt.sigmax()
>>> oracle = qt.tensor(X, I) * qt.cnot(2, 0, 1) * qt.tensor(X, I)
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
Qobj data =
[[ 0.  1.  0.  0.]]
```

①

②

③

④

⑤

```
[1. 0. 0. 0.]
[0. 0. 1. 0.]
[0. 0. 0. 1.]
>>> register_state = oracle * register_state
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket ⑦
Qobj data =
[[[-0.5]
 [ 0.5]
 [ 0.5]
 [-0.5]]]
```

- ➊ We prepare the control and target qubits in the $|+-\rangle$ state in exactly the same way as we did before.
- ➋ As in Chapter 4, it's helpful to define some variables I and X as shorthand for the identity matrix (`qt.qeye`) and the matrix representing the X operation, respectively.
- ➌ This time our oracle is the not oracle, which we implement with the sequence of instructions `X(control); CNOT(control, target); X(control)`; as per [7.8](#).
- ➍ The unitary operator for the oracle operation looks a bit different this time: it flips the target qubit when the control qubit is a $|0\rangle$.
- ➎ For instance, row 0 (00 in binary) tells us that $|00\rangle$ is transformed to $|01\rangle$.
- ➏ Similarly, row 2 (10 in binary) tells us that $|10\rangle$ is transformed to $|10\rangle$; the oracle leaves that input alone.
- ➐ The state after applying the oracle is $(-|00\rangle + |01\rangle + |10\rangle - |11\rangle) / \sqrt{2} = (-1) |+-\rangle$, precisely the same as before, aside from a global phase of -1 .

Looking at these two examples, we got the exact same output state, except the signs are all flipped. This means that if we multiplied one of the state vectors by a -1 , they would both be the same. Multiplying an entire vector by a constant can be referred to as adding a global phase. Since global phases cannot be observed through any measurements, this means that we got *exactly* the same information out from applying the `id` and not oracles. We learned nothing about whether we applied `id` or not, and if we were able to compare the vectors right here we would only know that we applied a balanced oracle.

Now let's see for comparison what the register looks like after we apply an oracle representing a *constant* function.

Example 3: the zero oracle

Once more with feeling, let's use Python to break down how an oracle representing the constant function zero works. We want to do this with the zero oracle here so that we can show what goes differently when we apply an oracle representing a constant function. This oracle is especially easy to apply, since it consists of applying no instructions at all.

Table 7.12. Representing the one-bit function zero as two-qubit oracles

Function Name	Function	Output of Oracle	Q# Operation
not	$f(x) = 0$	$ x\rangle y\rangle \oplus 0\rangle = x\rangle y\rangle$	(empty)

Listing 7.14. Computing how the zero oracle transforms its input state.

```
>>> control_state = H * qt.basis(2, 0)
>>> target_state = H * qt.basis(2, 1)
>>> register_state = qt.tensor(control_state, target_state)
>>> X = qt.sigmax()
>>> oracle = qt.tensor(I, I)
>>> oracle
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
Qobj data =
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]] ➊
>>> register_state = oracle * register_state
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket ➋
Qobj data =
[[ 0.5]
 [-0.5]
 [ 0.5]
 [-0.5]]
```

- ➊ Doing nothing on the control qubit and nothing on the target qubit can be simulated by doing nothing on the entire register; thus, we get out the two-qubit identity matrix $\mathbb{1} \otimes \mathbb{1}$ for the zero oracle.
- ➋ Here, we see our first difference from the previous listing. Before, we used the `id` oracle and got that the output state was $(|00\rangle - |01\rangle - |10\rangle + |11\rangle) / 2$. Using the zero oracle, we see that the output state is $(|00\rangle - |01\rangle + |10\rangle - |11\rangle) / 2$. These two states differ by more than a global phase, as there's no scalar that we can multiply by to turn the `id` output into the zero output.

Here now where the minus signs are on our state vector are different. There is no number we can multiply the entire vector by to change it into either $[[0.5], [-0.5], [-0.5], [0.5]]$ or $[[-0.5], [0.5], [0.5], [-0.5]]$. To see how this difference leads to us being able to tell for sure whether we had a balanced or constant oracle, let's continue to the next step of the Deutsch–Jozsa algorithm.

Exercise 7.2: Try out the one oracle

See if you can use the python tricks we have used above to work out how the state of the target and control qubits change when the one oracle is applied.

7.4.3 Steps 3 and 4. Undo the preparation on the target qubit and measure.

At this point, it is much easier to make sense of the output if we undo the steps we used to prepare $|+-\rangle$, so that everything is back in the computational basis ($|00\rangle \dots |11\rangle$). To review, 7.5 has the state vectors for all four oracles (three of which we worked out above).

Table 7.13. The vectors representing the register after applying the various oracles

Function Name	State of the register after applying the oracle
zero	$[[0.5], [-0.5], [0.5], [-0.5]]$
one	$[[-0.5], [0.5], [-0.5], [0.5]]$
id	$[[0.5], [-0.5], [-0.5], [0.5]]$
not	$[[-0.5], [0.5], [0.5], [-0.5]]$

Now we want to undo our preparation steps on the target qubit.

Why do we do "unprepare" the target qubit?

Recall from Chapter 6 that we need to reset our qubits to the $|0\rangle$ state before returning them to the target machine. At this point, our target qubit is always in the $|-\rangle$ state, **no matter which oracle we use**. This means that after applying the oracle, we know exactly how to put it back to $|0\rangle$. As we saw in Chapter 6, this helps us avoid an additional measurement, which can be expensive on some quantum devices.

Note that we can safely return the target qubit to $|-\rangle$ without affecting the results when we measure the control qubit, as the oracle call is the only two-qubit operation in the Deutsch–Jozsa algorithm. As we saw in Chapter 4, doing single-qubit operations on one qubit can't on their own affect results on another qubit; otherwise, we would be able to send information faster than light!

Let's try this out by undoing the preparation on the register from the oracle representing the id function.

Listing 7.15. Putting the oracle output back in the computational basis for the oracle representing id.

```
>>> I = qt.qeye(2)                                     1
>>> register_state_id = qt.cnot(2,0,1) *             2
... (qt.tensor(H * qt.basis(2, 0), H * (qt.sigmax() * qt.basis(2, 0))))
...
>>> register_state_id = qt.tensor(I, H) * register_state_id      3
>>> register_state_id
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.          ]
 [ 0.70710678]
 [ 0.          ]
 [-0.70710678]]
>>> register_state_id = qt.tensor(I, qt.sigmax()) * register_state_id 4
>>> register_state_id                                         5
```

```
>>> register_state_id
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket      6
Qobj data =
[[ 0.70710678]
 [ 0.
 [-0.70710678]
 [ 0.        ]]
>>> qt.tensor(H * qt.basis(2, 1), qt.basis(2, 0)) == register_state_id    7
True
>>> register_state_id = qt.tensor(H, I) * register_state_id                 8
>>> register_state_id
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket      9
Qobj data =
[[0.]
 [0.]
 [1.]
 [0.]]
```

- ➊ It's helpful to define a shorthand for the identity matrix `I`, which we use to represent what happens to a qubit when we don't apply any instruction to it. In QuTiP, we can get the identity matrix with the `qeye` function (the name stems from that the identity matrix is often written I , which is then pronounced as "eye").
- ➋ Here we reproduce the register for the oracle representing the `id` function, just after the oracle was applied.
- ➌ Since we're transforming a two-qubit state now, we need to say what happens to each qubit to get our matrix out. We can do this using the `tensor` function again.
- ➍ The output's now much easier to read: the register is in the state $(|01\rangle - |11\rangle) / \sqrt{2}$.
- ➎ In our Q# program, we used the `X` instruction to return the target qubit to $|0\rangle$ before releasing it. We can simulate that by using the `X` operator, implemented by the QuTiP function `sigmax()`.
- ➏ Since the `X` instruction flips its argument, applying the `X` matrix gives us the state $(|00\rangle - |10\rangle) / \sqrt{2}$.
- ➐ We can use QuTiP to confirm that another way of writing $(|00\rangle - |10\rangle) / \sqrt{2}$ is $(H|1\rangle) \otimes |0\rangle = |-0\rangle$.
- ➑ In our Q# program, we used the `MResetX` operation from the Q# standard libraries to measure in the `X` basis. An `X`-basis measurement returns `Zero` when its argument is in $|+\rangle$ and returns a `One` result when its argument is in $|-\rangle$. Thus, we can simulate the `MResetX` operation by applying `H` and then measuring in the `Z`-basis.
- ➒ Doing so, we read that the state of our register immediately before measuring has a `1` in row 2 (10 in binary), so we conclude that the state of our register is $|10\rangle$. Measuring the first qubit thus will give us a `One` with certainty.

Looking at the final vector in 7.4, we can see that that represents the state $|10\rangle$. If we were to measure the control qubit from that state, we would get the classical bit `One` 100% of the time. In `Algorithm.qs`, we returned to the user that an oracle is balanced if we measured `One` on the control qubit, so we correctly conclude that `id` is a balanced oracle! The fact that we will measure `One` on the control bit every time is really cool.

IMPORTANT**Quantum algorithms need not be random**

Though some quantum algorithms can be random, like the QRNG example from Chapter 2 or Morgana's and Lancelot's game from Chapter 6, they don't need to be. In fact, the Deutsch–Jozsa algorithm is *deterministic*: we get the same answer every time we run it.

Table 7.14. The vectors representing the state of the register after applying the various oracles.

Function Name	Register state just before measurement	Result of measuring control qubit along Z
zero	$[[1], [0], [0], [0]]$	Zero
one	$[[-1], [0], [0], [0]]$	Zero
id	$[[0], [0], [1], [0]]$	One
not	$[[0], [0], [-1], [0]]$	One

From these examples, we have made some important observations:

- Applying an oracle to a control and target qubit can affect the state of the control qubit.

7.5 Reflecting back

Phew, we've taken a couple pretty big steps here!

- We've used classical reversible functions to model Merlin's strategy in a way that we can write it down as a quantum oracle.
- We've used Q# and the Quantum Development Kit to implement the Deutsch–Jozsa algorithm and test that we can learn Merlin's strategy with a single oracle call, and
- We've used Python and QuTiP to model the Deutsch–Jozsa algorithm and see what happens to Nimue's qubits when she gives them to Merlin.

At this point, it's helpful to reflect back on what we learned from taking a splash in Nimue's quantum lake, as the techniques we used in this Chapter will be helpful throughout the rest of the book.

7.5.1 Shoes and socks: applying and undoing quantum operations

The first pattern that's helpful to reflect on is one that you might have noticed in `Algorithm.qs`. Let's take another look at the order in which operations were applied to the target qubit:

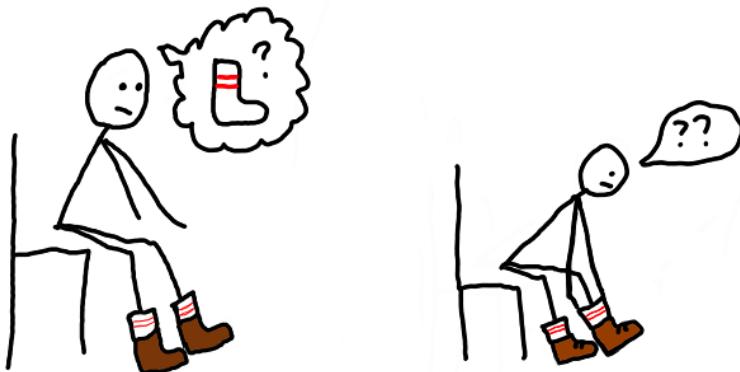
Listing 7.16. A sequence of instructions from the Deutsch–Jozsa algorithm for the target qubit.

```
// ...
```

```
X(target);
H(target);
oracle(control, target);
H(target);
X(target);
// ...
```

One way to think of this sequence is that the X(target); H(target); instructions prepare target in the $|-\rangle$ state, while the H(target); X(target); instructions "unprepare" $|-\rangle$, returning target back to the $|0\rangle$ state. We have to reverse the order due to what's often called the "shoes and socks" principle. If you want to put on shoes and socks, you'll have better results if you put on your socks first, but if you want to take them off, you need to take your shoes off first.

Figure 7.14. You can't take your socks off before your shoes.



The Q# language makes it easier to do shoes-and-socks kind of transformations of your code, using a feature called *functors*. Functors allow us to easily describe new variants of an operation that we have already defined. Let's jump right into an example and introduce a new operation, `PrepareTargetQubit`, that encapsulates the X(target); H(target); sequence:

Listing 7.17. Separating out the state preparation from 7.5.

```
operation PrepareTargetQubit(target : Qubit) : Unit is Adj { ①
    X(target);
    H(target);
}
```

- ① By writing `is Adj` as a part of the signature, we tell the Q# compiler to automatically compute the inverse operation — that is, the *adjoint* — of this operation.

We can then call the inverse operation generated by the compiler using `Adjoint`, one of the two functors provided by Q# (we'll see the other one in Chapter 8):

Listing 7.18. Using the Adjoint keyword to apply instructions following shoes-and-socks.

```
PrepareTargetQubit(target);
oracle(control, target);
Adjoint PrepareTargetQubit(target); ①
```

- ① Adjoint PrepareTargetQubit applies the Adjoint functor to PrepareTargetQubit, giving back an operation that "undoes" PrepareTargetQubit. Following shoes-and-socks sorts of thinking, this new operation works by first calling Adjoint H(target); and then Adjoint X(target);.

NOTE**Self-adjoint operations**

In this case, X and Adjoint X are the same operation, since flipping a bit and then flipping it again always gets you back to where you started. Put differently, X undoes X. Similarly, Adjoint H is the same as H, so the above snippet gives us the sequence H(target); X(target);. We say that the instructions X and H are *self-adjoint*.

Not all operations are their own adjoints, though! For instance, Adjoint Rz(theta, $_$) is the same operation as Rz(-theta, $_$).

In more practical terms, the Adjoint functor on the operation U is the same as the operation that reverses or undoes the effects of U . The name "adjoint" refers to that the conjugate transpose U^\dagger of a unitary matrix U is called the *adjoint* of U . The Adjoint keyword in Q# guarantees that if an operation U is described by the unitary U , then if Adjoint U exists, it is described by U^\dagger .

The pattern of performing an instruction is so commonly used that the Q# standard libraries provides the ApplyWith operation to express this pattern of doing and then undoing an operation.

NOTE**The Q# canon**

The ApplyWith operation is provided by the Microsoft.Quantum.Canon namespace in the Q# standard libraries. The name "canon" builds on a kind of musical naming scheme that's common in functional languages. For instance, intrinsic operations like M and X are defined in the *prelude*. Since the canon follows the prelude, and represents common patterns that play out over and over again throughout a program, the name fits the kind of whimsy you've come to expect by now.

Using ApplyWith and partial application, we can rewrite the using block of IsOracleBalanced in a compact way:

Listing 7.19. Using ApplyWith and partial application to help with shoes-and-socks ordering.

```
H(control);
```

```
ApplyWith(PrepareTargetQubit, oracle(control, _), target);
set result = MResetX(control);
```

The `ApplyWith` operation in the sample above will automatically apply the adjoint of `PrepareTargetQubit` after `oracle(control, _)` is done. Note that the `_` is used for partially applying the `oracle` to the `control` qubit.

Let's expand [7.6](#) step by step to see how it all works. The call to `ApplyWith` applies its first argument, then applies its second argument, then the adjoint of its first argument all to the qubit supplied in the last argument:

Listing 7.20. Expanding `ApplyWith` in [7.6](#).

```
H(control);
PrepareTargetQubit(target);
(oracle(control, _))(target);
Adjoint PrepareTargetQubit(target);
set result = MResetX(control);
```

The partial application on line 3 can then be replaced by substituting `target` in for the `_`:

Listing 7.21. Resolving partial application in [7.7](#).

```
H(control);
PrepareTargetQubit(target);
oracle(control, target);
Adjoint PrepareTargetQubit(target);
set result = MResetX(control);
```

Using operations like `ApplyWith` is helpful for reusing common patterns in quantum programming, and in particular for making sure we don't forget to take an `Adjoint` in a large quantum program!

7.5.2 Using Hadamard instructions to flip control and target

One way that we can use the "shoes-and-socks" kinds of thinking from the previous section is to change which qubits play the role of a control and a target in instructions like `CNOT`.

In understanding how this works, it's important to keep in mind with quantum computing is that quantum instructions transform the entire state of the registers that they act upon. In cases like the Deutsch–Jozsa algorithm, this means that the control qubit can be affected by applying gates to the control and target qubits together — not just the target qubit. This is an example of a more general pattern, that the control and target of a `CNOT` operation swap roles when we apply a `CNOT` instruction in the `X` basis instead of the `Z` (computational) basis.

To see this, let's look at the unitary operator (the quantum analogue to classical truth

tables) for what happens if we use H instructions to transform to the X basis, apply a CNOT instruction, and then use more H instructions to go back to the Z basis:

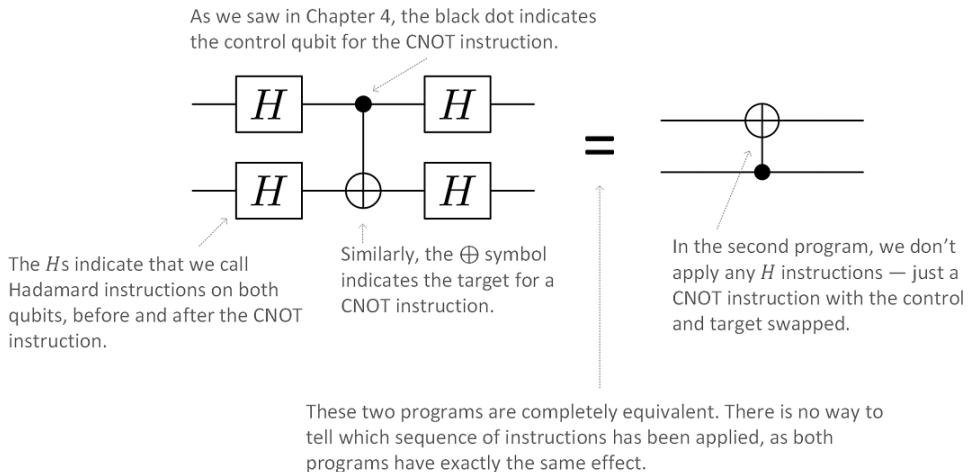
Listing 7.22. Using QuTiP to check that H flips between the control and target of a CNOT.

```
>>> import qutip as qt
>>> H = qt.hadamard_transform()
>>> HH = qt.tensor(H, H)
>>> HH
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
Qobj data =
[[ 0.5  0.5  0.5  0.5]
 [ 0.5 -0.5  0.5 -0.5]
 [ 0.5  0.5 -0.5 -0.5]
 [ 0.5 -0.5 -0.5  0.5]]
>>> HH * qt.cnot(2, 0, 1) * HH
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
Qobj data =
[[1.  0.  0.  0.]
 [0.  0.  0.  1.]
 [0.  0.  1.  0.]
 [0.  1.  0.  0.]]
>>> qt.cnot(2, 1, 0)
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
Qobj data =
[[1.  0.  0.  0.]
 [0.  0.  0.  1.]
 [0.  0.  1.  0.]
 [0.  1.  0.  0.]]
```

- ➊ It's helpful to define a shorthand for the unitary operator $H \otimes H$ that simulates the sequence of instructions $H(\text{control})$; $H(\text{target})$;
- ➋ Looking at the unitary operator $H \otimes H$, we see that $|00\rangle$ is transformed to $(|00\rangle + |01\rangle + |10\rangle + |11\rangle) / \sqrt{2}$, a uniform superposition over all four computational basis states.
- ➌ This line gives us the unitary operator representing $H(\text{control})$; $H(\text{target})$; $\text{CNOT}(\text{control}, \text{target})$; $H(\text{control})$; $H(\text{target})$. We can think of this sequence of instructions as applying a CNOT in the X basis instead of the Z basis.
- ➍ The unitary operator for this sequence looks a bit like CNOT, but with some of the rows flipped around... what happened?
- ➎ To try and figure out what applying H instructions to each qubit did to the CNOT instruction, let's look at the unitary operator for $\text{CNOT}(\text{target}, \text{control})$.
- ➏ Reversing the role of the control and target qubits in a call to the CNOT instruction gives us *exactly* the same unitary operator as using H instructions to apply a CNOT instruction in the X basis.

Figure 7.15. Changing which qubits are the "control" and "target" of a CNOT instruction using Hadamard instructions.

Let's look at two different programs (written as circuits) that each use a CNOT instruction.



From the above calculation, we can conclude that `CNOT(target, control)` does precisely the same thing as `H(control); H(target); CNOT(control, target); H(control); H(target);`. In the same way that the `H` flips the role of the `X` and `Z` bases, `H` instructions can flip between using a qubit as a control and as a target.

7.5.3 Phase Kickback

With these techniques in mind, we're now equipped to explore what makes the Deutsch–Jozsa algorithm do its thing, namely a quantum programming technique called *phase kickback*. This technique is what let us write the `IsOracleBalanced` operation so that it works for several different oracles, while revealing only the one bit we wanted to know (whether Merlin was acting as a good mentor or not).

To see how the the Deutsch–Jozsa algorithm uses phase kickback to work in *general*, let's go back to our three ways of thinking, and use math to predict what happens when we call any oracle. Recall that we defined the oracle U_f that we constructed from each classical function f such that, for all classical bits x and y , $U_f|xy\rangle = |x\rangle|f(x)\oplus y\rangle$.

TIP

Above, we used x and y to represent classical bits that label two-qubit states; this is another example of using the computational basis to reason about how quantum programs behave.

Let's start the same way as we did in our QuTiP programs above, by expanding the input state $|+-\rangle = |+\rangle \otimes |- \rangle$ in the computational basis. Starting by expanding the state

of the control qubit, we have that $|+-\rangle = |+\rangle \otimes |-\rangle = (|0\rangle + |1\rangle) / \sqrt{2} \otimes |-\rangle = (|0-\rangle + |1-\rangle) / \sqrt{2}$. As before, we can check our math using QuTiP (see 7.8).

Listing 7.23. Using QuTiP to check that $(|0-\rangle + |1-\rangle) / \sqrt{2} = |+-\rangle$.

```
>>> import qutip as qt
>>> from numpy import sqrt
>>> H = qt.hadamard_transform()
>>> ket_0 = qt.basis(2, 0)
>>> ket_1 = qt.basis(2, 1)
>>> ket_plus = H * ket_0
>>> ket_minus = H * ket_1
>>> qt.tensor(ket_plus, ket_minus)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.5]
 [-0.5]
 [ 0.5]
 [-0.5]]
>>> (
...     qt.tensor(ket_0, ket_minus) +
...     qt.tensor(ket_1, ket_minus)
... ) / sqrt(2)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.5]                                     ①
 [-0.5]
 [ 0.5]
 [-0.5]]
```

- ① We'll start by writing out some useful shorthand notation.
- ② Both vectors are the same, telling us that $(|0-\rangle + |1-\rangle) / \sqrt{2}$ is another way of writing out $|+-\rangle$.

Next, as we saw in Chapter 2, we can use linearity to predict how U_f transforms this input state.

NOTE

The matrix revisited

We've also implicitly used linearity earlier in this section as well, when we used matrices to model how the Deutsch–Jozsa algorithm works. As described in Chapter 2, matrices are one way of writing down linear functions.

Since U_f is a unitary matrix, we know that for *any* states $|\psi\rangle$ and $|\phi\rangle$, and for *any* numbers α and β , $U_f(\alpha |\psi\rangle + \beta |\phi\rangle) = \alpha U_f |\psi\rangle + \beta U_f |\phi\rangle$. Using this property with the computational basis, we have that in the same way that $|+-\rangle$ and $(|0-\rangle + |1-\rangle) / \sqrt{2}$ are the same state, $U_f |+-\rangle$ and $U_f (|0-\rangle + |1-\rangle) / \sqrt{2}$ are the also same state.

TIP

Using our shorthand for multi-qubit states, $|+-\rangle = |+\rangle \otimes |-\rangle$, $|0-\rangle = |0\rangle \otimes |-\rangle$, and $|1-\rangle = |1\rangle \otimes |-\rangle$.

Figure 7.16. Applying linearity to understand how our oracle transforms the input state.

$$\begin{aligned}
 U_f |+-\rangle &= U_f (\overline{|0-\rangle + |1-\rangle}) / \sqrt{2} \\
 &= (\overline{U_f |0-\rangle + U_f |1-\rangle}) / \sqrt{2}
 \end{aligned}$$

Since $|+-\rangle = (|0-\rangle + |1-\rangle) / \sqrt{2}$, we can substitute that in here.

We want to know what happens when we apply our oracle to the input state that we've prepared, so we start by writing that down.

Using linearity, we can apply U_f to each term above (that is, to $|0-\rangle$ and $|1-\rangle$).

Written like this, it's not immediately clear what advantage we've obtained by applying U_f to $|0-\rangle$ and $|1-\rangle$. Let's look at how the oracle operation applies to the first term by factoring out the control (first) qubit, so that we can consider the effect on the target qubit.

Doing so, we'll once again make use of linearity to understand how U_f works by passing one state at a time to our oracle. As we learned in Chapter 2, linearity is a very powerful tool that lets us break down even quite complicated quantum algorithms into pieces that we can understand and analyze more easily. In this case, we can understand how U_f acts on $|0-\rangle$ by using linearity (that is, by breaking $|0-\rangle$ down into a superposition between $|00\rangle$ and $|01\rangle$):

We start by using that $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$.

$$U_f |0-\rangle = U_f(|00\rangle - |01\rangle)/\sqrt{2}$$

We can apply U_f to each term by linearity.

$$= (U_f |00\rangle - U_f |01\rangle)/\sqrt{2}$$

The definition of an oracle tells us how U_f transforms states in the computational basis.

$$= (|0(0 \oplus f(0))\rangle - |0(1 \oplus f(0))\rangle)/\sqrt{2}$$

XORing something with 0 does nothing.

$$= (|0f(0)\rangle - |0(1 \oplus f(0))\rangle)/\sqrt{2}$$

XORing something with 1 is the same as NOT.

$$= (|0f(0)\rangle - |0(\neg f(0))\rangle)/\sqrt{2}$$

Since the control qubit is in $|0\rangle$ in both terms, we can factor it out.

$$= |0\rangle \otimes (|f(0)\rangle - |\neg f(0)\rangle)/\sqrt{2}$$

For instance, if we're considering the zero function, then $f(0) = 0$. Thus, $|f(0)\rangle = |0\rangle$ and $|\neg f(0)\rangle = |1\rangle$, so $U_f |0-\rangle = |0-\rangle$.

On the other hand, if $f(0) = 1$, then $U_f |0-\rangle = |0\rangle \otimes (|1\rangle - |0\rangle)/\sqrt{2} = -|0-\rangle$. That is, U_f flips the sign of $|0-\rangle$.

TIP

$(|1\rangle - |0\rangle)/\sqrt{2}$ can also be written as $-|-\rangle$, or as $X|-\rangle$.

We can note then that U_f either rotates the target qubit by X or not depending on the value of $f(0)$:

Equation 7.1

$$U_f |0-\rangle = (-1)^{f(0)} |0-\rangle / \sqrt{2}$$

We can use the exact same argument that we used above to understand what U_f does to $|0-\rangle$ to understand what U_f does in the case the the control qubit is in the $|1\rangle$ state instead. Doing so, we get a phase of $(-1)^{f(1)}$ instead of $(-1)^{f(0)}$, so that $U_f |1-\rangle = (-1)^{f(1)} |1-\rangle$.

Using linearity again to combine the terms for the two states of the control qubit, we now know how U_f transforms the state of both qubits when the control qubit is in $|+\rangle$:

Equation 7.2

$$U_f |+-\rangle = 1/\sqrt{2} ((-1)^{f(0)} |0-\rangle + (-1)^{f(1)} |1-\rangle)$$

The last step is to note that, as we saw in Chapter 4, we cannot observe *global* phases. Thus, we can factor out $(-1)^{f(0)}$ to express the output state in terms of $f(0) \oplus f(1)$, the question we were interested in to start, as shown in [7.17](#).

Figure 7.17. Working out the last couple steps to the Deutsch–Jozsa algorithm

By using linearity, and that $|+-\rangle = (|0-\rangle + |1-\rangle) / \sqrt{2}$, we can apply U_f to both of the cases we've already computed, $U_f|0-\rangle$ and $U_f|1-\rangle$.

In both cases, we get a phase that depends on the value of our irreversible classical function, evaluated at the label of the control qubit; e.g.: $U_f|0-\rangle = (-1)^{f(0)}|0-\rangle$.

$$U_f |+-\rangle = \frac{1}{\sqrt{2}} \overline{((-1)^{f(0)} |0-\rangle + (-1)^{f(0)} |1-\rangle)}$$

To finish our exploration of the Deutsch–Jozsa algorithm, we need to know what the oracle U_f does to $|+-\rangle$, the input state prepared by our H and X instructions.

Next, we can factor out $f(0)$, so that it's easier to see that it's a global phase.

$$= \frac{1}{\sqrt{2}} \overline{(-1)^{f(0)} (|0-\rangle + (-1)^{\overline{f(0) \oplus f(1)}} |1-\rangle)}$$

$$= \frac{1}{\sqrt{2}} (-1)^{f(0)} \left(|0\rangle + (-1)^{f(0) \oplus f(1)} |1\rangle \right) \otimes |-\rangle$$

Doing so leaves us with a phase on the $|1-\rangle$ part of the state that depends on $f(0) \oplus f(1)$, which is 0 for constant functions and is 1 for balanced functions.

Finally, we notice that the target qubit is in the same state *regardless of the state of the control qubit*. Thus, we can factor it out, and safely measure or reset the target qubit without affecting the control.

If $f(0) \oplus f(1) = 0$ (constant), then the output state is $|+-\rangle$, but if $f(0) \oplus f(1) = 1$ (balanced), then the output state is $|--\rangle$. With one call to U_f , we learn whether f was constant or balanced *even though we do not learn what $f(x)$ is for any particular input x* .

One way to think of what happened when we applied U_f with the input qubit in the $|+\rangle$ state is that the state of the input qubit represents the question we are asking about f . If we ask the question $|0\rangle$, we get the answer $f(0)$, while if we ask $|1\rangle$, we get the answer $f(1)$. The question $|+\rangle$ then tells us about $f(0) \oplus f(1)$ without telling us about either $f(0)$ or $f(1)$ alone.

When we ask questions in superposition like this, however, the role of "input" and "output" isn't as immediately clear as it is classically. In particular, the $|0\rangle$ and $|1\rangle$ inputs both cause the output qubit to flip, while the $|+\rangle$ input causes the *input* qubit to

flip, provided we start the output qubit in $|-\rangle$ state. In general, two-qubit operations like U_f transform the entire space of the qubits that it acts upon — our division into inputs and outputs is one of how we interpret the action of U_f .

IMPORTANT

Phase Kickback

The fact that the state of the input qubit changed based on transformations defined on the output qubit is an example of a quantum effect known as **phase kickback**. In the next two chapters, we'll leverage phase kickback in order to explore new algorithms, such as those used in quantum sensing and quantum chemistry simulations.

Deep Dive: Extending Deutsch–Jozsa

While we only considered functions with one bit inputs here, the Deutsch–Jozsa algorithm will only ever need one query for any sized input/output of our function!

To encode a two-qubit function $f(x_0, x_1)$, we can introduce a three-qubit oracle $U_f |x_0 x_1 y\rangle = |x_0 x_1\rangle \otimes |f(x_0, x_1) \oplus y\rangle$. For example, consider $f(x_0, x_1) = x_0 \oplus x_1$. This function is balanced since $f(0, 0) = f(1, 1) = 0$ but $f(0, 1) = f(1, 0) = 1$. When we apply U_f to the three-qubit state $|++-\rangle = (|00\rangle + |01\rangle + |10\rangle + |11\rangle) \otimes |-\rangle$, we get $(|00\rangle - |01\rangle - |10\rangle + |11\rangle) \otimes |-\rangle = |---\rangle$. Using an X -basis measurement, we can tell this apart from a constant function like $f(x_0, x_1) = 0$, which will give us an output of $|++-\rangle$.

As long as we're promised that f is either constant or balanced, the same pattern holds no matter how many bits f takes as input: we can learn a single bit of data about how f behaves with a single call to U_f . Talk about $O(1)$! If you are not familiar with Big O notation, see *Grokking Algorithms* by Aditya Bhargava.

7.6 Summary

In this chapter you learned:

- to recognize quantum algorithms,
- to design *oracles* to represent classical functions within quantum algorithms,
- to predict the output of quantum programs by using QuTiP and linearity, and
- to recognize quantum programming techniques like phase kickback.

In the next chapter, we'll build on these skills by looking at how the **phase estimation algorithm** enables spin-off technologies like quantum sensors.

Quantum sensing: It's not just a phase



This chapter covers:

- Predict how quantum operations can be used to learn useful information about unknown operations with the quantum algorithm for phase kickback.
- Create new types in Q#
- Run Q# code from a Python host program
- Recognize important properties and behaviors of eigenstates and phase
- Program controlled quantum operations in Q#

8.1 Phase estimation: leveraging useful properties of qubits for measurement

Throughout the book so far, you've seen that games can be a really helpful way to learn quantum computing concepts. In the previous Chapter, for instance, Lady Nimue's game with Merlin let you explore your first quantum algorithm, the Deutsch–Jozsa algorithm. In this Chapter you'll use another game to explore how to learn the phases of quantum states using *phase kickback*, the quantum development technique used by Deutsch–Jozsa and many other quantum algorithms.

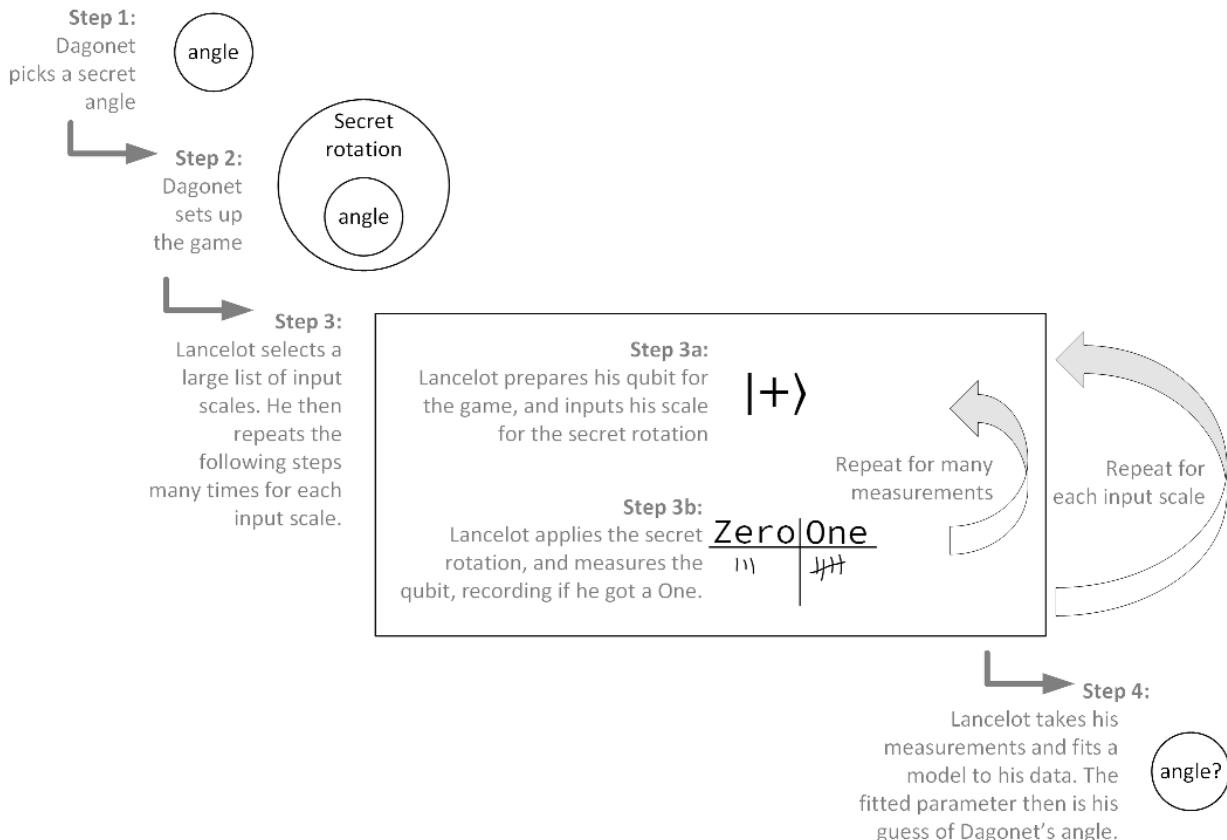
For this Chapter's game, let's go back and see what Lancelot has been up to. While Nimue and Merlin were deciding the fate of kings, you find Lancelot and the court jester Dagonet playing a guessing game where he has "borrowed" some of Nimue's quantum tools. Since they've had a while to play, however, Dagonet has gotten a bit bored and wants to start incorporating quantum computing to make their game a bit

more interesting.

8.1.1 Part and Partial Application

For Dagonet's new game, rather than having Lancelot guess a number, Dagonet has Lancelot guess what a quantum operation does to a single qubit by letting Lancelot call it with different inputs. Given all single qubit operations are rotations, this work really well for the game. Dagonet can pick a rotation angle about a particular axis, then Lancelot gets to input a number to Dagonet's operation which will change how to scale the rotation that Dagonet applies. What axis Dagonet picks doesn't really matter, as the game is to guess the rotation angle. For convenience here, Dagonet's rotations will always be around the Z axis. Finally, Lancelot can measure the qubit and use his measurement to try and guess what Dagonet's original rotation angle was. See [8.1](#) for a flowchart of these steps.

Figure 8.1. Playing Dagonet's guessing game



Steps for Dagonet and Lancelot's game

1. Dagonet picks a secret angle for a single qubit rotation operation
2. Dagonet prepares an operation for Lancelot to use that hides their secret angle, and allows Lancelot one additional input of a number (we'll call it a scale) that will get multiplied with the secret angle to give the total rotation angle of the operation.
3. Lancelot's best strategy for the game will be to select many scale values and estimate the probability of measuring One for each value. To do this, he will need to do steps a. and b. many times for each of the many scale values.
 - a. Prepare the $|+\rangle$ state and input the scale value in Dagonet's rotation. He uses the $|+\rangle$ state because he knows Dagonet is rotating around the Z axis, and for this state these rotations will result in a local phase change he can measure.
 - b. After preparing each $|+\rangle$ state, Lancelot can rotate it with the secret operation, measure the qubit, and record the measurement.
4. Lancelot will then have data relating his scale factor and the probability he measured a 1 for that scale factor. He can then fit this data in his head, and get Dagonet's angle from the fitted parameters (he *is* the greatest knight in the land). We can use Python to help us do the same!

Note that this *is* a game, as there is no way for Lancelot to directly measure this rotation with just a single measurement. If he could, it would violate the No-Cloning theorem, and he would transcend the laws of physics himself. As a Knight of the Round Table, Lancelot is bound not only by duty and honor, but also by the laws of physics, so he will have to play Dagonet's the game by the rules.

DEEP DIVE: Learning the axis with Hamiltonian learning

In this book, we've focused on the case where Dagonet's rotation axis is known but we need to learn his angle. This case corresponds to a common problem in physics, where one is tasked with learning the *Larmor precession* of a qubit in a magnetic field. Learning Larmor precessions isn't just useful in building qubits, but allows for detecting very small magnetic fields and for building very precise sensors.

More generally, though, you can learn much more than a single rotation angle. The case in which the axis is unknown as well is an example of a general kind of problem called *Hamiltonian learning*, a rich area of research in quantum computing. In Hamiltonian learning, one reconstructs a physical model for a qubit or register of qubits using a game very similar to the one you'll explore throughout the rest of this Chapter.

Let's jump into prototyping this game in Q#! It will be helpful to have access to different parts of the Q# standard libraries, so you can start by adding the open statements below to the top of your Q# file, `operations.qs`. We show the open statements used in this example in [8.1](#).

Listing 8.1. Opening Q# namespaces needed for Lancelot's and Dagonet's game.

```
namespace PhaseEstimation {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Convert as Convert;
    open Microsoft.Quantum.Measurement as Meas;
```

1
2
3
4

```
open Microsoft.Quantum.Arrays as Arrays;
// ...
```

5

- ➊ All open statements in a Q# file come right after your namespace declaration.
- ➋ As before, opening Microsoft.Quantum.Intrinsic gives you access to all the fundamental instructions (e.g.: R1, Rz, X, and so forth) that you can send to a quantum device.
- ➌ You can also give an alias to namespaces when you open them, similar to how you can alias Python packages and modules when you import them. Here, for instance, we've abbreviated Microsoft.Quantum.Convert so that we can later use type conversion functions in that namespace by prefixing them with Convert..
- ➍ Similarly, you can make the MResetZ operation that you saw in previous Chapters available as Meas.MResetZ to document where the operation came from.
- ➎ The last namespace you'll need to open in this Chapter is the Microsoft.Quantum.Arrays namespace, which provides a number of useful functions and operations for working with arrays.

In [8.2](#), you can see an example of the quantum operation Dagonet would need to implement the rotation he and Lancelot will play with.

Listing 8.2. A quantum operation Dagonet could use to setup his part of the game.

```
operation ApplyScaledRotation(angle : Double,
                             scale : Double,
                             target : Qubit)
: Unit is Adj + Ctl {
    R1(angle * scale, target);
}
```

1

2

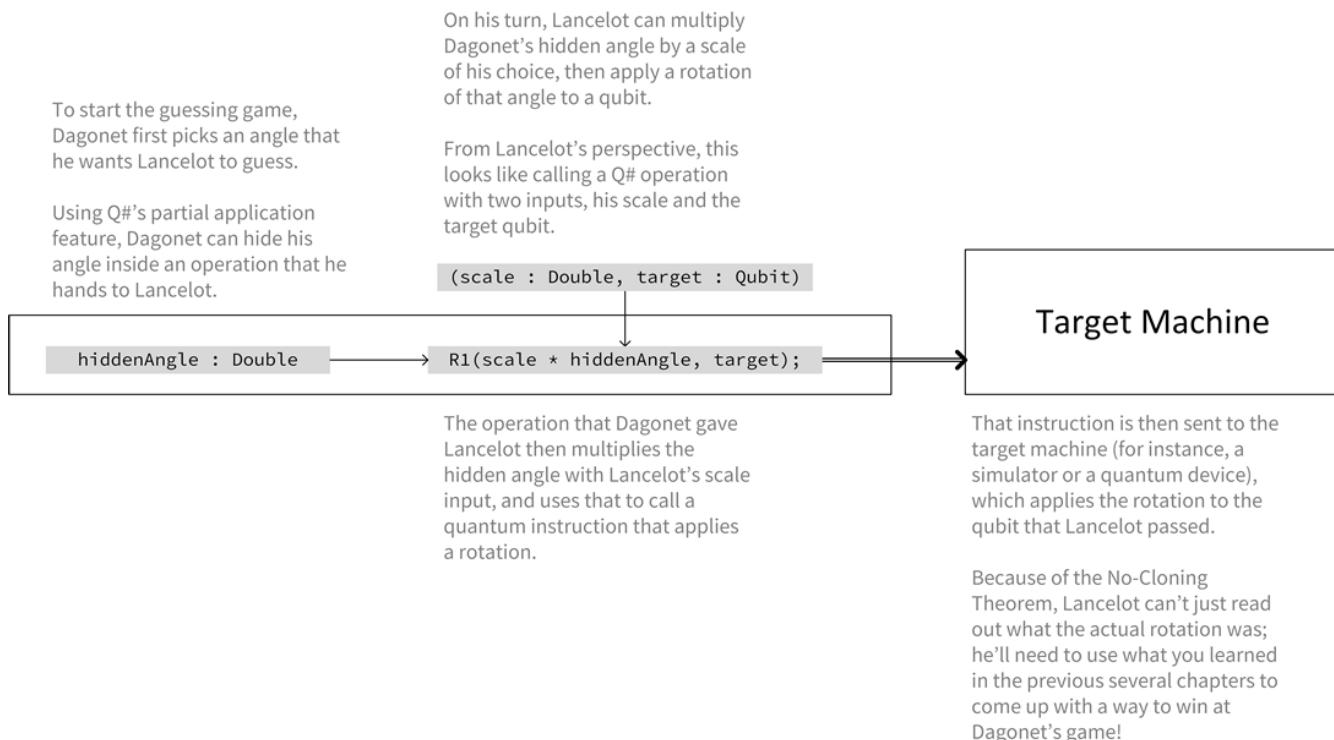
3

- ➊ To play the guessing game, we'll need a quantum operation that takes two classical arguments; one that Dagonet gets to pass, and one that Lancelot gets to pass.
- ➋ Like other rotations, our new rotation operation returns Unit (the type of the empty tuple ()), indicating that there's no meaningful output from the operation. The is Adj + Ctl part of the signature indicates that this operation supports the Adjoint functor that you first saw in Chapter 6, as well as the Controlled functor that you'll see later in this chapter. Don't worry too much about this part for now, though.
- ➌ For the actual body of the operation, you can find the angle to rotate by by multiplying Dagonet's hidden angle angle with Lancelot's scale factor scale. The rotation operation R1 here is almost identical to the Rz operation that you've seen a few times so far now. The difference between R1 and Rz will become important later, when you add the Controlled functor.

IMPORTANT

When writing Q# in its own file (that is, not from a Jupyter Notebook), all operations and functions must be defined inside of a namespace. This helps keep your code organized, and makes it harder to conflict with what code may be written in the various libraries that you use in your quantum application. For brevity, we often won't show the namespace declarations, but the full code can always be found in the samples repository for this book: github.com/crazy4pi314/learn-qc-with-python-and-qsharp

Figure 8.2. Playing Dagonet's guessing game with partial application



DEEP DIVE: Why not just measure the angle?

It may seem like Lancelot has to go through a lot of hoops to guess Dagonet's hidden angle. After all, what besides duty and honor is there stopping Lancelot from passing a scale of 1.0 and then just reading out the angle from the phase applied to his qubit? It turns out that the No-Cloning Theorem strikes again in this case, telling us that Lancelot can never learn a phase from a single measurement.

The easiest way to see this is to pretend for a moment that Lancelot could do that, then see what goes wrong. Suppose that Dagonet hides an angle of $\pi/2$, and that Lancelot prepares his qubit in the $|+\rangle = 1/\sqrt{2} (|0\rangle + |1\rangle)$ state. If Lancelot then passes 1.0 as his scale, his qubit ends up in the $1/\sqrt{2} (|0\rangle + i|1\rangle)$ state. If Lancelot could directly measure the phase $e^{i\pi/2} = +1$ in order to guess Dagonet's angle from a single measurement, he could use that to prepare another copy of $1/\sqrt{2} (|0\rangle + i|1\rangle)$ state, even though Lancelot didn't know the right basis to measure in. Indeed, Lancelot's magic

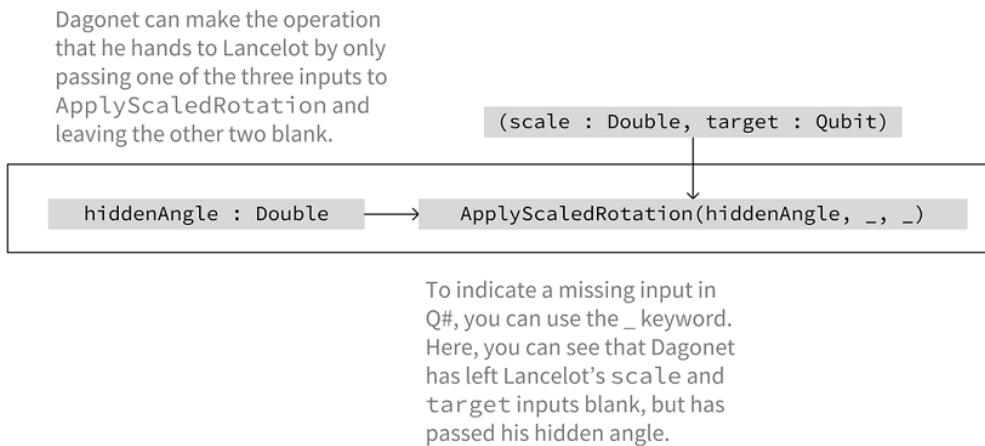
measurement device should also work if Dagonet hides the angle π , in which case Lancelot winds up with a qubit in the $1/\sqrt{2}(|0\rangle - |1\rangle)$ state.

Put differently, if Lancelot could figure out Dagonet's angle by measuring phases directly, he could make copies of arbitrary states of the form $1/\sqrt{2}(|0\rangle + e^{i\varphi}|1\rangle)$ for any angle φ , *without having to know φ ahead of time*. That rather badly violates the No-Cloning Theorem, so you can safely conclude that Lancelot will need to do a bit more work to win Dagonet's game.

Once you've defined an operation in this way, Dagonet can use the partial application feature of Q# that you first saw in the previous chapter to hide his input. Then Lancelot gets an operation that he can apply to his qubits, but not in a way he can directly see the angle he is trying to guess.

Using `ApplyScaledRotation`, Dagonet can easily make an operation for Lancelot to call. For instance, if Dagonet picks the angle 0.123 , he can 'hide' it by giving Lancelot the operation `ApplyScaledRotation(0.123, _, _)`. As with the examples of partial application that you've seen in Chapter 7, the `_` indicates a slot for future inputs.

Figure 8.3. Partially applying `ApplyScaledRotation` to make an operation for Lancelot.



As shown in [8.3](#), since `ApplyScaledRotation` has type `(Double, Double, Qubit) => Unit` is `Adj + Ctl`, providing only the first input results in an operation of type `(Double, Qubit) => Unit` is `Adj + Ctl`. This means that Lancelot can provide an input of type `Double`, a qubit he wants to apply his operation to, and can use the `Adjoint` and functor that you saw in Chapter 6.

Now, the fact that we can see the value of the angle in the syntax above does *not* mean Lancelot can. Indeed, the only things that Lancelot can do with a partially-applied operation or function are to call it, partially apply it further, or pass it to another function or operation. From Lancelot's perspective, `ApplyScaledRotation(0.123, _, _)` is a complete black box. Thanks to this partial application trick, he will just have an

operation that takes his scale value and can be used to rotate a qubit.

We can make our lives as Q# developers a bit easier, though, by giving a name to the type of Lancelot's operation that's a bit easier to read than (Double, Qubit) => Unit is Adj + Ctl. In the next section, you can see how Q# lets us annotate the type signatures that you use to play Dagonet and Lancelot's guessing game.

8.2 User-Defined Types

You have seen a bit already on how types play a role in Q#, particularly in the signatures for our functions and operations. You have also seen that both functions and operations are tuple-in, tuple-out. In this section you will see how you can build up your own types in Q# and why that might be handy.

In Q# (as well as many other languages) there are a number of types that are defined as a part of the language itself; types like Int, Qubit, and Result that we have seen already.

TIP

For a complete list of these basic types see the Q# language docs here: docs.microsoft.com/quantum/language/type-model#primitive-types

Building up from these basic types, you can make array types by adding a [] after the type. For example, in the game for this chapter you will likely need to input an array of doubles to represent Lancelot's multiple inputs to Dagonet's operation. You can use Double[] to indicate an array of Double values, see [8.3](#).

Listing 8.3. An example of defining an array type, here an array of `Double`'s of length 10.

```
let scales = new Double[10];
```

You can also define your own types in Q# with the newtype statement. This statement allows you to declare new *user-defined types* (often abbreviated "UDTs"). There are two main reasons why you may want to use UDTs.

Reasons you might define UDTs in Q#:

- convenience
- communicate intent

The first reason is a matter of *convenience*. Sometimes the type signature for your function or operation could get pretty long, so you can define your own type as a kind of shorthand. Another reason you may want to name your own type is so you can communicate *intent*. Say your operation takes a tuple of two Double values that represents a complex number. Defining a new type Complex could be useful to help remind you and your teammates what that tuple represents.

Listing 8.4. How complex numbers are defined in the Q# runtime.

```
namespace Microsoft.Quantum.Math {  
    newtype Complex = (Real: Double, Imag: Double);  
}
```

- ➊ The Quantum Development Kit provides a number of different functions, operations, and UDT to the various namespaces that make up the Q# libraries. For instance, complex numbers are implemented as a user-defined type in the Microsoft.Quantum.Math namespace; this lets you use this type by including the statement open Microsoft.Quantum.Math; in your quantum application.
- ➋ The Complex type is defined as a tuple of two Double values, where the first item is named Real and the second is named Imag.

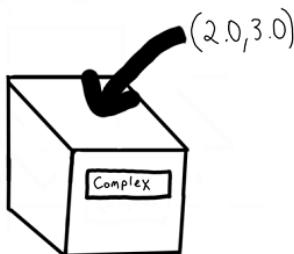
TIP**Back to the source**

The Quantum Development Kit is open source, so if you're curious you can always look up how various parts of the Q# language, runtime, compiler, and standard libraries all work. For example, the definition of the Complex user-defined type can be found in the src/Simulation/Intrinsic/Math/Types.qs file in Q# runtime repository at github.com/microsoft/qsharp-runtime/.

As shown in [8.4](#), there's two ways to get the different items back out of a user-defined type.

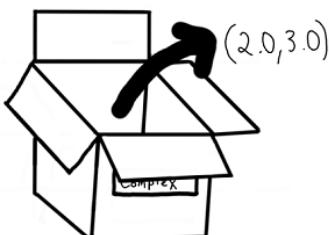
Figure 8.4. Using the :: and the ! operator with UDTs.

```
let z = Complex(  
    2.0, 3.0  
)
```



When you create a new UDT, you can think of it as a kind of label for a tuple of data. For example, the Complex UDT tells us that its values represent complex numbers.

`z!`



If you want to just get the data out of a UDT, you can use the ! operator to unwrap it.

`z::Real`



To get just a part of the UDT, you can use the :: operator with the name of the item you want.

You can either use named items together with the :: operator, or you can "unwrap" the user-defined type with the ! operator to get back to the original type wrapped by the UDT.

```
function TakesComplex(complex : Complex) : Unit {  
    let realFromNamedItem = complex::Real; ❶  
    let (real, imag) = complex!; ❷  
}
```

- ❶ Since the Complex UDT is defined with a named item called Real, you can access that item as ::Real to get back the real part of our input.
- ❷ Alternatively, since Complex was defined as wrapping a tuple of type (Double, Double), the unwrap operator ! takes you back to the real and imaginary parts of complex without the UDT wrapper.

TIP

Both ways of working with UDTs are useful in different cases, but most of the time we'll stick with using named items and the :: operator in this book.

Once a new UDT has been defined, it can also act as a way to instantiate a new

instance of that type. For example, the `Complex` type will act as a function that will create a new complex number with the input of a tuple of two `Double` values (see an example of this in [8.5](#)). This is similar to Python where types are also functions that create instances of that type.

Listing 8.5. Creating a complex number with the user defined type Complex.

```
let imaginaryUnit = Complex(0.0, 1.0);
```

1

- ➊ Defining a user-defined type with `newtype` also defines a new function with the same name as that type that returns values of your new user-defined type. For instance, we can call `Complex` as a function with two `Double` inputs representing the real and imaginary parts of the new `Complex` value. Here, we've defined a `Complex` value representing $0 + 1.0 i$ (`0+1j` in Python notation), also known as the imaginary unit.

Exercise 8.1

In Chapter 4, you used Python type annotations to represent the concept of a *strategy* in the CHSH game. User-defined types in Q# can be used in a similar fashion. Give it a go by defining a new UDT for CHSH strategies and then use your new UDT to wrap the constant strategy from Chapter 4.

HINT: Your and Eve's parts of the strategy can each be represented as operations that take a `Result` and output a `Result`. That is, as operations of type `Result` \Rightarrow `Result`.

For the game in this chapter, we have defined a new UDT both so that we can label how we intend to use it, and as a convenient shorthand for the operation type that Lancelot gets as his part of the guessing game. In [8.6](#), you can see the definition of this new type, called `ScalableOperation`, as a tuple with one named input called `Apply`.

Listing 8.6. Setting up the quantum guessing game

```
newtype ScalableOperation = (
    Apply: ((Double, Qubit) => Unit) is Adj + Ctl)
;

function HiddenRotation(hiddenAngle : Double) : ScalableOperation {
    return ScalableOperation(ApplyScaledRotation(hiddenAngle, _,_));
}
```

1

2

3

4

- ➊ You can declare a new user-defined type with the `newtype` statement by giving a name for your new type, and defining the underlying type that your new type is based on.
- ➋ You can give names to the various items in a user-defined type using the same syntax as to declare the signature for an operation or a function. Here, for example, your new UDT has a single item named `Apply` that allows for calling the operation wrapped by `ScalableOperation`.
- ➌ Once defined, you can use your new UDT just like any other type. Here, you've defined a function that outputs values of type `ScalableOperation`.

- ④ You can easily make output values by calling ScalableOperation with the operation to be wrapped in your new UDT. In this example, you can create new instances of ScalableOperation by using the same partial application of ApplyScaledRotation that you saw earlier in the Chapter.

TIP**What a capital idea!**

When you define inputs to functions and operations in Q#, those inputs have names that start with lowercase letters. In [8.6](#), however, the named item Apply in ScalableOperation starts with an initial upper-case letter. This is because the inputs to a function or operation only have meaning within that callable, while named items mean something more broadly; you can use the capitalization of inputs and named items to make it obvious where to look for the definitions of each.

The function HiddenRotation defined in [8.6](#) helps us in implementing Lancelot and Dagonet's game, by giving us a way for Dagonet to hide his angle. Calling HiddenRotation with Dagonet's angle will return a new ScalableOperation that Lancelot can then call to gather the data he needs to make a guess at the hidden angle.

Figure 8.5. Playing Dagonet's guessing game with partial application and user-defined types

Putting it all together, you now have an operation that Dagonet can call with his angle to hide that angle using partial application, and then wrap the result in a user-defined type to make the usage of that operation clear.

By unwrapping the ScalableOperation that Dagonet gives him with “`::Apply`”, Lancelot can now call the operation that Dagonet hid his angle in.

```
hiddenAngle : Double → function HiddenRotation(hiddenAngle : Double) : ScalableOperation {
    return ScalableOperation(ApplyScaledRotation(hiddenAngle, _,_));
}
```

```
(scale : Double, target : Qubit) → op::Apply(scale, target)
```

With some new types and a way for Dagonet to hide his angle, let's continue implementing the rest of the game! You have everything you need to go on to the next step: estimating the probability for each measurement that you make during Lancelot's and Dagonet's game. This is very similar to how you would estimate the probability of flipping a coin, see [8.6](#).

Figure 8.6. Lancelot's estimation is similar to estimating the outcome of a coin flip.

Suppose that someone hands you a coin, and you'd like to know the *bias* of that coin. That is, the probability that your shiny new coin lands heads.

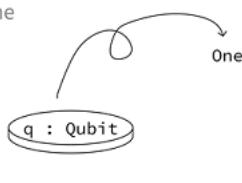


One way to do that is to flip your coin a bunch, and make a table of how many times it lands heads and how many times it lands tails.

H	T

If you get 3 heads and 5 tails, you could very reasonably estimate that the probability of getting heads is $3 / 8 = 37.5\%$. With more flips, you could get a more accurate estimate.

The same idea holds if you're trying to estimate the probability of getting a One from measuring a qubit.



Zero	One

For example, suppose you prepare a qubit in the state $|\psi\rangle$ and measure it, repeating the whole process eight times. If you get five One results, you would estimate that $|\langle 1 | \psi \rangle|^2 = 5 / 8 = 62.5\%$.

Listing 8.7. Estimating the probability of measuring $|1\rangle$

```
operation EstimateProbabilityAtScale(scale : Double,
                                      nMeasurements : Int,
                                      op : ScalableOperation)
: Double {
    mutable nOnes = 0;
    for (idx in 0..nMeasurements - 1) {
        using (target = Qubit()) {
            within {
                H(target);
            } apply {
                op::Apply(scale, target);
            }
            set nOnes += Meas.MResetZ(target) == One
        }
    }
}
```

1
2
3
4
5
6
7
8
9
10

```

        ? 1
        | 0;
    }
}
return Convert.IntAsDouble(nOnes) /
    Convert.IntAsDouble(nMeasurements);
}

```

11

- ➊ To play the game, Lancelot needs to estimate the probability of measuring a $|1\rangle$ at each given scale that he picks. Thus, his operation needs to take in a Double value representing what scale he picks to run the operation given to him by Dagonet.
- ➋ Next, Lancelot needs to pick a number of times to measure his qubit to get his estimate of the probability.
- ➌ The last input Lancelot needs to take is a value of the ScalableOperation user-defined type that you declared earlier in the chapter. This input represents the operation that Dagonet gives Lancelot.
- ➍ As output, Lancelot will want to return an estimated probability, so you can declare that as a Double output.
- ➎ To keep track of the number of $|1\rangle$ outcomes that have been observed so far, you can define a mutable variable. As before, initializing the variable with the Integer value 0 sets the type of the mutable variable as well as its initial value.
- ➏ For each measurement, you'll need to allocate a qubit that's the target for Dagonet's operation.
- ➐ Next, you can use the `within` and `apply` keywords to use the shoes-and-socks kind of pattern that you first learned about in Chapter 7.
- ➑ Since Lancelot and Dagonet agreed that the Z -axis should be the rotation axis for their game, Lancelot can prepare his target qubit in the $|+\rangle$ state so that Dagonet's rotation does something. Here, you can implement Lancelot's strategy by using the `H` operation to prepare a qubit in the $|+\rangle$ state.
- ➒ Once you have prepared the input to Dagonet's operation, you can call it by using `::Apply` to unwrap the ScalableOperation UDT.
- ➓ At this point, the `within/apply` block has made sure that Lancelot's qubit is back in the right axis. You can count how many times the final measurement returns a One result by adding either 1 or 0 to `nOnes`. Here, the `?|` ternary operator (much like the `if ... else` operator in Python or the `?:` operator in C, C++, and C#) provides a convenient way to increment `nOnes`.
- ➔ To get your final estimate as to the probability of measuring $|1\rangle$, you can take the fraction of how many times you got a one over how many times you measured. Here, `Convert.IntAsDouble` helps you to get the types right returning a floating-point number out.

TIP**An operation by any other name**

You may have noticed that operations tend to be named using verbs, while functions tend to be named as nouns. This helps you to remember the distinction that you saw in Chapter 6, namely that a function **Is** something while an operation **does** something. Being consistent with names can help you make sense of how a quantum program works, so Q# uses conventions like this throughout the language and libraries.

With this in place, you can now write out an operation that runs the whole game, and returns everything Lancelot needs to guess Dagonet's hidden angle.

Listing 8.8. Running the complete game.

```

operation RunGame(hiddenAngle : Double,
                  scales : Double[],
                  nMeasurementsPerScale : Int)
: Double[] {
    let hiddenRotation = HiddenRotation(hiddenAngle);
    return Arrays.ForEach(
        EstimateProbabilityAtScale(
            _,
            nMeasurementsPerScale,
            hiddenRotation
        ),
        scales
    );
}

```

1
2
3

4

- ➊ Dagonet can start off by making a new ScalableOperation value that hides his angle using the HiddenRotation function that you wrote earlier.
- ➋ The ForEach operation in Microsoft.Quantum.Arrays (which you gave the shorthand name Arrays above) takes an operation and applies it to every element of the scales array.
- ➌ To get the operation we pass to ForEach, we use partial application lock down how many measurements Lancelot does at each different scale, and what the hidden operation he was given by Dagonet. This leaves an operation that only takes the scale as an input.
- ➍ When you pass scales as the second input to ForEach, each element of scales will be substituted into the partial application slot _ above.

NOTE**Functions and operations redux**

It may seem funny that ForEach acts like map in Python and other languages, when Q# also has Microsoft.Quantum.Arrays.Mapped. The critical difference is that ForEach takes an operation, while Mapped takes a function.

For Lancelot to actually make sense of all the data he gets out of his Q# program, though, it might help to use some good old classical data science techniques. Since Python is great at that, running your new RunGame operation from a Python host program can be a great way for you to help Lancelot out.

8.3 Run, snake, run: Running Q# from Python

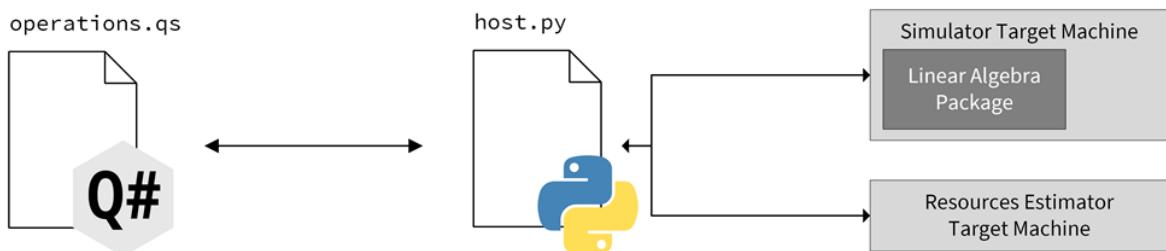
In previous Chapters, you ran your Q# code in a Jupyter Notebook with a Q# kernel. In this Chapter we want to look at a different way to run Q# code: from Python. Calling Q# from Python can be helpful in different scenarios, especially if you want to pre-process data before using it in Q#, or if you want to visualize your resulting output from your quantum program.

Let's start actual writing the files that will implement Dagonet and Lancelot's game. To try out the Q# and Python interop, we will use a Python host program to run the Q#

program. This means we will have two files for our game, our `operations.qs` file and a `host.py` file that we will directly use to run the game. Let's dive into the `host.py` file to see how we can interact with Q# from Python.

Figure 8.7. Using host programs written in Python

Instead of using Jupyter Notebook as a pre-made host program, you can also write your own host program in Python. As in previous chapters, the host program is responsible for sending your Q# program to a target machine.



All of the interoperable functionality we will need between Python and Q# is provided by the `qsharp` Python package.

TIP

Reminder that there are full installation instructions for the `qsharp` Python package in Appendix A.

Once you have the `qsharp` package, you can import it just like any other Python package. Let's look at a small sample Python file where we can see this in action.

Listing 8.9. An example of a Python file where we use Q# code directly in the Python file.

```

import qsharp                               1

prepare_qubit = qsharp.compile("""
    open Microsoft.Quantum.Diagnostics;      2
    3

    operation PrepareQubit(): Unit {          4
        using (qubit = Qubit()) {
            DumpMachine();
        }
    }
""")                                         5

if __name__ == "__main__":
    prepare_qubit.simulate()
  
```

- ➊ The qsharp Python package needs to be imported just like any other Python package.
- ➋ You can use the `qsharp.compile` Python function to take a string containing Q# code and compile it for use in the Python file, the callable result of the compilation here being called `prepare_qubit`.
- ➌ Just like a regular Q# file, you need to include open statements to use the different parts of the Q# standard library.
- ➍ The operation this string of Q# code describes simply prepares a qubit in the $|0\rangle$ state, and uses `DumpMachine` to show what the target machine knows about that qubit. Given you are using the simulator, `DumpMachine` shows the values the simulator is using to record the state of the qubit.
- ➎ When this Python script is run, we want to also use the callable defined as `prepare_qubit`. To do so, you can use the `simulate` method from the `qsharp` package which will run previously compiled Q# code snippets.

Let's try running the `qsharp-interop.py` script we just looked at in [8.9](#).

Listing 8.10. Running the previous example of a Python script containing Q# code.

```
$ python qsharp-interop.py
Preparing Q# environment...
# wave function for qubits with ids (least to most significant): 0
|0>:    1.000000 + 0.000000 i == ***** [ 1.000000 ] --- [
0.0000 rad ]
|1>:    0.000000 + 0.000000 i == [ 0.000000 ]
```

From the output in [8.10](#), you can see that it did indeed prepare a $|0\rangle$ state, as the only term in the output that has a coefficient of 1.0 is the $|0\rangle$ state.

The `qsharp` Python package will also look for Q# operations or functions defined in `*.qs` files in the same directory as your Python program. In your case, as you proceed through the rest of this chapter, you'll be adding things to a Q# file called `operations.qs`. This is a pretty convenient way for you to start your `host.py` file for the game.

Listing 8.11. The beginning of our `host.py` file for simulating the phase estimation game.

```
import qsharp
from PhaseEstimation import RunGame, RunGameUsingControlledRotations
from typing import Any
import scipy.optimize as optimization
import numpy as np

BIGGEST_ANGLE = 2 * np.pi
```

- ➊ First, you need to import the Q# Python package.
- ➋ The loaded `qsharp` package then allows you to import operations and functions from namespaces in Q# files in the same directory as `host.py`. Here, you import `RunGame` and `RunGameUsingControlledRotations` operations from `operations.qs`; this

automatically creates Python objects representing each Q# operation that you import. You saw RunGame above, and will see RunGameUsingControlledRotations shortly.

- ③ The rest of the imports will help us with type hinting in Python, visualizing the results of your Q# simulation, and fitting measurement data to get Lancelot's final guess.

Now that we have imported and setup our Python file, let's write run_game_at_scales: the function that will actually call the Q# operations.

Listing 8.12. The Python function that will call our Q# operations.

```
def run_game_at_scales(scales: np.ndarray,
                      n_measurements_per_scale: int = 100,
                      control: bool = False
                     ) -> Any:
    hidden_angle = np.random.random() * BIGGEST_ANGLE
    print(f"Pssst the hidden angle is {hidden_angle}, good luck!")
    return (
        RunGameUsingControlledRotations
        if control else RunGame
    ).simulate(
        hiddenAngle=hidden_angle,
        nMeasurementsPerScale=n_measurements_per_scale,
        scales=list(scales)
    )
```

1
2
3
4

- ➊ Here the return type hint is set to Any which tells Python to not worry about type checking the return value of this function.
- ➋ Here, Dagonet chooses the hidden angle that he wants Lancelot to guess.
- ➌ The return for run_game_at_scales is conditioned on control, which allows you to choose between two simulations you will develop for this game. Don't worry about the control = True case for now, you'll see more about that later in the Chapter.
- ➍ The Python objects created when qsharp importing these operations from Q# have a method called simulate, which takes the required arguments to the Q# operations and pass them along to the Q# target machine (here the simulator). When results are finished, they are returned by the Python function run_game_at_scales.

This Python file should be runnable as a script, so we also need to define `__main__` as well. This is where we can do what Lancelot does in his head by using our host program in Python to take the measurements and scales and fit them to a model for Dagonet's rotation. The best model for how the rotation angle changes the measurement results is given by where θ is Dagonet's hidden angle and where $scale$ is Dagonet's scale factor:

Equation 8.1

$$\Pr(1) = \sin(\theta * \text{scale})^2$$

Exercise 8.2

This model can be found if you use Born's rule! We have put the definition from Chapter 2 below, see if you can plot the resulting value as a function of Lancelot's scale using Python. Does your plot look like a trigonometric function?

Born's rule

$$\Pr(\text{measurement} | \text{state}) = |\langle \text{measurement} | \text{state} \rangle|^2$$

HINT: For Lancelot's measurements, the $\langle \text{measurement} |$ part of Born's rule is given by $\langle 1 |$. Immediately before measuring, his qubit will be in the state $HR_1(\theta * \text{scale})H|0\rangle$. You can simulate the R1 operation in QuTiP by using the matrix form in the Q# reference at docs.microsoft.com/qsharp/api/qsharp/microsoft.quantum.intrinsic.

Once you have that model and data, you can use the `scipy.optimize` function from the SciPy Python package to fit your data to the model. The value it finds for the θ parameter is Dagonet's hidden angle! Check out [8.13](#) for an example of how to pull this all together.

Listing 8.13. The code that will run when we run `host.py` as a script.

```

if __name__ == "__main__":
    import matplotlib.pyplot as plt
    scales = np.linspace(0, 2, 101)
    for control in (False, True):
        data = run_game_at_scales(scales, control=control)

        def rotation_model(scale, angle):
            return np.sin(angle * scale / 2) ** 2
        angle_guess, est_error = optimization.curve_fit(
            rotation_model, scales, data, BIGGEST_ANGLE / 2,
            bounds=[0, BIGGEST_ANGLE]
        )
        print(f"The hidden angle you think was {angle_guess}!")

        plt.figure()
        plt.plot(scales, data, 'o')
        plt.title("Probability of Lancelot measuring One at each scale")
        plt.xlabel("Lancelot's input scale value")
        plt.ylabel("Lancelot's probability of measuring a One")
        plt.plot(scales, rotation_model(scales, angle_guess))

    plt.show()

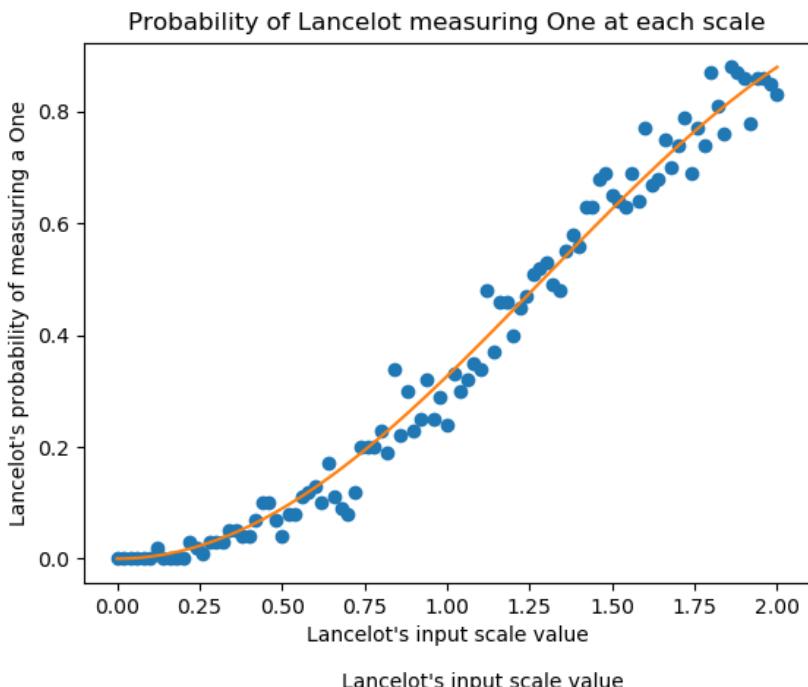
```

- ➊ This script will plot the data and fitted results so you will need to import the friendly `matplotlib`.
- ➋ Lancelot's list of inputs to the game (that is, his scales) are generated as just a regularly spaced, sequential list of numbers from `np.linspace`.

- ③ This script will run both versions of the game simulation so they can be compared; don't worry about the `control = True` case for now, we'll come back to that shortly.
- ④ `data` stores the result from the Q# simulation that runs from Python in `run_game_at_scales`.
- ⑤ This model represents the operation that is done on the qubit, i.e. its a rotation of an unknown angle where the angle can be multiplied by `scale` which is Lancelot's input to the game. Lancelot will take that data he gets back from his measurements and fit it to a model, with the goal of fitting the `angle` parameter from the data.
- ⑥ You can use a standard `scipy` function called `optimization.curve_fit` that takes a model of a function, inputs, measured data, and an initial guess to try and fit all the parameters of the model. This is what Lancelot needs to do to convert the data he collected into an actual angle to guess.
- ⑦ Validating the fit found by `optimization.curve_fit` is important so you can plot both the data and the fitted model to see if it looks right.
- ⑧ This displays the plots with the data and fit in new window.

Now that you have a host program you can use to run the whole game, you can see that Lancelot does a pretty reasonable job at figuring out what angle Dagonet hid in his Q# operation. By taking different measurements and using classical data science techniques, Lancelot can estimate the phase that Dagonet's operation applies to his qubits. Running `host.py` should generate 2 pop up windows that show plots of the measurement probabilities as a function of Lancelot's scale for two strategies he can use. The first is the approach we have already outlined above, the latter you will implement in the last section of the chapter. See [8.8](#) for an example for what you should see.

Figure 8.8. An example of one of the two plots that should pop up when you run `host.py`



TIP Since the SciPy fitting packages are not perfect, sometimes the fitted parameter it finds won't be right. Just try running it a few times and hopefully the fitting algorithm will do better the next time. If you have any questions about the plotting package `matplotlib` check out these other titles from Manning: *Data Science Bootcamp Chapter 2*, and *Data Science with Python and Dask Chapters 7 and 8*.

You can see from these plots, that you were able to fit Lancelot's data fairly well. That means that the fitted value that you find in `angle_guess` is a pretty good approximation to Dagonet's hidden angle!

There's still one more nagging problem with Lancelot's strategy, though: every time he performs a measurement, he needs to prepare the right input to pass to Dagonet's operation. In this particular game, that may not be much of a problem, but as you explore bigger applications of this game in the next chapter, it can be expensive to prepare the input register in the right state every time. Thankfully, you can use *controlled operations* to reuse the same inputs over and over again, as you'll see in the rest of the chapter.

You have seen examples of controlled operations already (like CNOT), but it turns out many other quantum operations can also be applied conditionally which can be very useful. Controlled operations along with the last new quantum computing concept you

will need (eigenstates) will help you to implement a technique you have already seen at the end of the last chapter, called phase kickback.

TIP

There will be a lot of discussion about local and global phases in next few sections. Recall that global phase is a complex coefficient that can be factored out of all of the terms of your state, and cannot be observed. If you need a refresher on phase, check out Chapters 4 and 5.

8.4 Eigenstates and Local Phases

By now, you've seen that the X quantum operation allows us to flip bits ($|0\rangle \leftrightarrow |1\rangle$) and that the Z operation allows us to flip phases ($|+\rangle \leftrightarrow |-\rangle$). Both of these operations, though, only apply global phases to some input states. As you saw in previous Chapters, we can't actually learn anything about global phases, so understanding what states each operation leaves alone is important to understanding what we can learn by applying that operation.

For example, let's revisit the Z operation. In [8.14](#), you can see what happens when we try to use Z not to flip a qubit between the $|+\rangle$ and $|-\rangle$ states, but on an input qubit in the $|0\rangle$ state.

Listing 8.14. Applying the Z instruction to a qubit in the $|0\rangle$ state.

```
using (qubit = Qubit()) {  
    Z(qubit);  
    DumpRegister("0.txt", [qubit]);  
    Reset(qubit);  
}
```

- ➊ As usual in Q#, we start by allocating a Qubit with a using block. This will provide us with a fresh qubit in the $|0\rangle$ state.
- ➋ We then apply a Z operation, such that the state of qubit is transformed to $Z|0\rangle = |0\rangle$.
- ➌ To confirm that the Z operation didn't do anything, we can use the DumpRegister function to instruct the simulator to print out all of its diagnostic information-- in this case, the full state vector. If we run this on a target machine other than a simulator, we won't get a state vector, but whatever other diagnostics that machine offers (e.g.: hardware IDs).
- ➍ Finally, we reset the qubit before releasing it. This isn't strictly needed, since we know beforehand that that qubit is still in the $|0\rangle$ state.

Listing 8.15. Output

```
# wave function for qubits with ids (least to most significant): 0  
|0>: 1.000000 + 0.000000 i == ***** [ 1.000000 ] --- [  
0.00000 rad ] ➊  
|1>: 0.000000 + 0.000000 i == [ 0.000000 ] ➋
```

- ➊ The first line of the output from DumpRegister shows the coefficient of the $|0\rangle$ state; here, DumpRegister shows you that the coefficient is simply 1 (written out as $1.000000 + 0.000000$

- i). The output also shows the magnitude of this coefficient as a bar plot filled in with asterisks (*) for convenience.
- ② The second line shows you that the coefficient of the $|1\rangle$ state is 0. Taken with what you learned from the first line, you have that the state being dumped is $|1\rangle|0\rangle + 0|1\rangle = |0\rangle$.

Note that in the listing above, Z didn't do anything to qubit since $Z|0\rangle = |0\rangle$. If we modify the above listing to prepare $|1\rangle$ instead by using X before Z, we'll see something very similar.

Listing 8.16. Applying the Z instruction to a qubit in the $|1\rangle$ state.

```
using (qubit = Qubit()) {
    X(qubit);
    Z(qubit);          1
    DumpRegister("1.txt", [qubit]); 2
    Reset(qubit);
}
```

- ① As before, if you want to prepare a $|1\rangle$ state, you can use that $|1\rangle = X|0\rangle$.
- ② We next repeat our experiment from above, but with a different input.
- ③ As before, we can write out the state of qubit to a text file, using that we're running on a simulator that keeps the state internally.

Listing 8.17. Output

```
# wave function for qubits with ids (least to most significant): 0
|0⟩: 0.000000 + 0.000000 i == [ 0.000000 ]
|1⟩: -1.000000 + 0.000000 i == ***** [ 1.000000 ] --- [ 3.14159 rad ] 1
```

- ① This file represents the vector $[|0\rangle, -|1\rangle]$, or $-|1\rangle$ in Dirac notation.

The effect of applying the Z operation to a $|1\rangle$ state was to flip the sign of the state of qubit. This is another example of a *global phase*, as you saw in Chapters 5 and 7.

Definition 8.1: Global phase

Whenever two states $|\psi\rangle$ and $|\varphi\rangle$ are only different by a complex number $e^{i\theta}$, $|\varphi\rangle = e^{i\theta}|\psi\rangle$, we say that $|\psi\rangle$ and $|\varphi\rangle$ vary by a *global phase*. For example, $|0\rangle$ and $-|0\rangle$ differ by a global phase of $-1 = e^{i\pi}$.

The global phase of a state doesn't affect any measurement probabilities, so we cannot ever detect whether we applied a Z operation when its input is in either the $|0\rangle$ or $|1\rangle$ state. We can confirm this by using the `AssertQubit` operation, which checks the probability of a particular measurement result.

```
using (qubit = Qubit()) {
    AssertQubit(Zero, qubit); 1
```

```

    Z(qubit);          2
    AssertQubit(Zero, qubit); 3

    Message("Passed test!"); 4

    Reset(qubit);
}

```

- ➊ This call to `AssertQubit` checks that measuring `qubit` will return the result `Zero`, and terminates the program if that's not the case.
- ➋ After this, `qubit` is in the $-|0\rangle$ state as opposed to the $|0\rangle$ state. That is, the `Z` operation applies a global phase to the state of `qubit`.
- ➌ Calling `AssertQubit` again, we can check that the probability of obtaining a `Zero` result is still 1.
- ➍ Printing a message lets us check that the quantum program proceeds past both assertions.

Running this snippet will simply print out `Passed test!`, since the calls to `AssertQubit` don't do anything in the case that the assertion succeeds. Using assertions like this lets us write unit tests that use simulators to confirm our understanding of how particular quantum programs will behave. On actual quantum hardware, since we can't actually do this kind of check due to the No-Cloning Theorem, assertions can be safely stripped out.

IMPORTANT

Don't depend on assertions!

Assertions can be really useful tools for writing unit tests and for checking the correctness of your quantum programs. That said, it's important to remember that they will be stripped out when running your program on actual quantum hardware, so don't use assertions to *make* your program run correctly.

Of course, this is also just good programming practice; assertions in classical languages like Python can often be disabled for performance reasons, such that you can't rely on assertions always being there for you.

Identifying which quantum states are assigned global phases by an operation U gives us a way of understanding the behavior of that quantum operation. We call such states *eigenstates* of the operation U . If two operations have the same eigenstates and apply the same global phases to each of those eigenstates, there is no way to tell those two operations apart. Just like if two classical functions have the same truth table, you can't tell which one is which, no matter what state your qubits are in when you apply each operation. This means that we can understand operations not only by a matrix representation of them, but also by understanding what their eigenstates are and what global phase the operation applies to each. As you have seen, we cannot directly learn about the global phase of a qubit, so in the next section you will learn how you can use controlled versions of an operation to turn that global phase into a local one you can actually measure. For now though, let's summarize this with a more formal definition of what an eigenstate is.

Definition 8.2:

If after applying an operation U , the state of a register of qubits qs is only modified by a global phase, then we say that the state of that register is an *eigenstate* of U . For example, $|0\rangle$ and $|1\rangle$ are both eigenstates of the Z operation. Similarly, $|+\rangle$ and $|-\rangle$ are both eigenstates of X .

Try out an exercise to practice working with eigenstates.

Exercise 8.3

Try writing Q# programs that use `AssertQubit` and `DumpMachine` to verify that:

- $|+\rangle$ and $|-\rangle$ are both eigenstates of the X operation.
- $|0\rangle$ and $|1\rangle$ are both eigenstates of the Rz operation, regardless of what angle you choose to rotate by.

For even more practice, try figuring out what the eigenstates of the Y and $CNOT$ operations and writing a Q# program to verify your guesses!

HINT: You can find the vector form of the eigenstates of a unitary operation using QuTiP. For instance, the eigenstates of the Y operation are given by `qt.sigmay().eigenstates()`. From there, you can use what you learned about rotations in Chapters 4 and 5 to figure out which Q# operations prepare those states.

Don't forget you can always test if a particular state is an eigenstate of an operation by just writing a quick test in Q#!

Eigenstates are a very useful concept and can be leveraged in a variety of quantum computing algorithms. You will use them in the next section along with controlled operations to implement a quantum development technique called *phase kickback* which you saw a bit of at the end of Chapter 7.

DEEP DIVE: It's only proper

Eigenstates get their name from a concept used throughout linear algebra, known as *eigenvectors*. Just as an eigenstate is a state that is left alone by a quantum operation, an eigenvector is a vector that is preserved up to a scaling factor when multiplied by a matrix. That is, if for a matrix A , $A\vec{x} = \lambda\vec{x}$ for some number λ , \vec{x} is an eigenvector of A . We say that λ is the corresponding *eigenvalue*.

The prefix "eigen-", German for "proper" or "characteristic," points to that eigenvectors and eigenvalues help us to understand the properties or characteristics of matrices. In particular, if a matrix A commutes with its conjugate transpose (that is, if $AA^\dagger = A^\dagger A$), then it can be *decomposed* into projectors onto eigenvectors, each scaled by their eigenvalues,

Equation 8.3

$$A = \sum_i \lambda_i \vec{x}_i \vec{x}_i^\dagger.$$

Since this condition always holds for unitary matrices, this means that we can always understand quantum operations by decomposing them into their eigenstates and the phases applied to each eigenstate. For example, $Z|0\rangle\langle 0| - |1\rangle\langle 1|$ and $X|+\rangle\langle +| - |-\rangle\langle -|$.

This way of thinking about states and operations can often help us make sense of different quantum computing concepts. In fact, another way of thinking about the phase estimation game you're working on playing in this chapter is as an algorithm for learning the phases associated with each eigenstate! You'll see in the next chapter that this connects especially well to some applications, such as to learning properties of chemical systems.

8.5 Controlled application: Turning global phases into local phases

From what you have seen and can test, *global* phases of states are unobservable, while *local* phases of states can be measured. For example, consider the state $1/\sqrt{2}(-|0\rangle - i|1\rangle) = -i/\sqrt{2}(|0\rangle + |1\rangle)$. There is no measurement you could do to differentiate that state from $(|0\rangle + |1\rangle)/\sqrt{2}$. However, you could distinguish either of those two states from $(|0\rangle - |1\rangle)/\sqrt{2}$ as it differs by a *local* phase; that is, one of the states has a $+$ in front of the $|1\rangle$ and the other has a $-$.

TIP

If you want a refresher on phases and how to think of them as rotations, see Chapters 4 and 5. When you are using the simulator as your target machine, the output of `DumpMachine` and `DumpRegister` can also be used to help learn about the phases of states.

From the last section, you played around with eigenstates and saw the global phases of eigenstates can carry information about an operation, let's call it U . If you want to learn this global phase information about the eigenstates, then it seems like you're stuck. If Lancelot were to only prepare eigenstates of Dagonet's operation, he'd never be able to learn what angle Dagonet had hidden.

Quantum algorithms to the rescue! There's a very useful trick that you can apply to turn global phases applied by an operation U into *local* phases applied by a closely related operation. To see how this works, let's return to the `CNOT` operation. Recall from Chapter 5 that you can simulate `CNOT` using a unitary matrix,

Equation 8.4: Simulating the CNOT operation with a unitary matrix.

$$U_{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

When you first encountered this matrix in Chapter 5, you used the analogy between unitary matrices and classical truth tables to work out that the CNOT operation swaps the $|10\rangle$ and $|11\rangle$ states, but leaves qubits in the $|00\rangle$ and $|01\rangle$ states alone. That is, CNOT flips the state of its second qubit, *controlled* on the state of the first qubit. As shown in [8.9](#), you can read the unitary matrix for the CNOT operation as describing a kind of "quantum if" statement: "**if** the control qubit is in the $|1\rangle$ state, **then** apply the X operation to the target qubit."

To use the CNOT operation in Q# you can try out the following snippet:

```
using (control = Qubit(), target = Qubit()) {
    H(control);
    X(target);                                ①

    CNOT(control, target);                     ②
    DumpMachine();

    Reset(control);
    Reset(target);
}
```

- ① Prepare the control qubit in $|+\rangle$.
- ② Prepare the target qubit in $|1\rangle$.
- ③ Apply CNOT and print out the state of the simulator.

By thinking about CNOT as a quantum analogue to a conditional statement, you can write out its unitary matrix a bit more directly. In particular, you can look at the unitary matrix for the CNOT operation as a kind of "block matrix" that you can build up using the tensor product that you saw in Chapter 4.

Equation 8.5

$$U_{CNOT} = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & X \end{pmatrix} = |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes X.$$

Exercise 8.4

Verify that $|0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes X$ is the same as the equation above.

HINT: You can verify this by hand, by using NumPy's `np.kron` function, or QuTiP's `qt.tensor` function. If you need a refresher, check out how you simulated teleportation in Chapter 5, or check out the derivation of the Deutsch–Jozsa algorithm in Chapter 7.

You can construct other operations following this pattern, such as the CZ (controlled-z) operation:

Equation 8.6: Simulating the CNOT operation with a unitary matrix.

$$U_{\text{CZ}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} = |00\rangle\langle 00| + |01\rangle\langle 10| + |10\rangle\langle 10| - |11\rangle\langle 11|$$

Much like the CNOT operation did the same thing as an X operation (namely, applied a bit flip), but controlled on the state of another qubit, when its control qubit is in the $|1\rangle$ state, the CZ operation flips a phase in the same way as the Z operation.

Figure 8.9. Writing out unitary matrices for controlled operations

Start with the CNOT operation that you saw in Chapter 5. You can represent it by a unitary matrix. Since CNOT acts on two qubits (the control and the target qubit), its unitary matrix is a 4×4 matrix:

The upper-left part of this matrix tells you what the CNOT operation does **when the control qubit is in the $|0\rangle$ state**.

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

The lower-right part tells you what happens when the **control qubit is in the $|1\rangle$ state**. Note that this part of the matrix for CNOT is the same as the matrix for the X operation.

$$U_{\text{CZ}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

You can write down unitary matrices for other controlled operations in the same way. For instance, the Controlled Z operation (CZ for short) is represented by a matrix with the **I** and **Z** matrices on its diagonal.

You can see an example of how this works in [8.9](#), let's see how controlling Z works in practice by writing out some Q# to give CZ a try:

Listing 8.18. Snippet showing how to test out the Q# operation CZ.

```
using ((control, target) = (Qubit(), Qubit())) {
```

```

H(control);
X(target);1  
2

CZ(control, target);
DumpRegister("cz-output.txt", [control, target]);3

Reset(control);
Reset(target);
}

```

- ➊ Prepare the control qubit in $|+\rangle$.
- ➋ Prepare the target qubit in $|1\rangle$.
- ➌ Apply CZ and save out the resulting state.

Listing 8.19. Output

```

# wave function for qubits with ids (least to most significant): 0;1
|0>: 0.000000 + 0.000000 i == [ 0.000000 ]
|1>: 0.000000 + 0.000000 i == [ 0.000000 ]
|2>: 0.707107 + 0.000000 i == **** [ 0.500000 ] --- [
0.0000 rad ]
|3>: -0.707107 + 0.000000 i == ***** [ 0.500000 ] --- [
3.14159 rad ]

```

If you run this, the contents of cz-output.txt will show you that the final state of the [control, target] register is $U_{CZ}|+1\rangle = |-1\rangle$.

Exercise 8.5

Either by hand or using QuTiP, verify that the above output is the same as $|-1\rangle = |-\rangle \otimes |1\rangle$.

NOTE: If you seem to get the right answer other than that the order of the qubits are swapped, note that DumpMachine uses a *little-endian* representation to order states. In little-endian, $|2\rangle$ is short-hand for $|01\rangle$, not $|10\rangle$. If this seems confusing, blame the x86 processor architecture...

That is, based on the state of the **target**, the state of the **control** changed as a result, just as you saw in Chapter 6 with the Deutsch–Jozsa algorithm! This is because the phase applied by Z in the case where control was in the $|0\rangle$ state is not the same as when control was in the $|1\rangle$ state, an effect known as **phase kickback**. In Chapter 6, you used phase kickback with a pair of qubits in the $|+-\rangle$ state to tell if the CNOT operation had been applied or not. Here, you've seen that you can use the CZ operation (also known as the Controlled Z operation) to learn about the global phase applied by the Z operation.

IMPORTANT

Even though $|1\rangle$ is an eigenstate of the Z operation, $|+1\rangle$ is **not** an eigenstate of the CZ operation. This means that calling CZ on a register in the $|+1\rangle$ state has an observable effect!

Phase kickback is a common quantum programming technique, as it allows us to turn what would otherwise be global phases into a phase between the $|0\rangle$ and $|1\rangle$ branches of the control qubit. In the CZ example above, both the input state $|+\rangle|1\rangle$ and the output state $|-\rangle|1\rangle$ are product states, allowing us to measure the control qubit without affecting the target qubit.

IMPORTANT

Think globally, learn phases locally.

Note that a global phase difference between $|1\rangle$ and $Z|1\rangle = -|1\rangle$ became a *local* phase difference between $|+1\rangle$ and $U_{CZ}|+1\rangle = |-1\rangle$. That is, by controlling the Z instruction on a qubit in the $|+\rangle$ state, we were able to learn what would have been a global phase without control.

```
using ((control, target) = (Qubit(), Qubit())) {
    H(control);                                ①
    X(target);

    CZ(control, target);
    if (M(control) == One) { X(control); }      ②
                                                ③

    // Now let's dump only the state of the target.
    DumpRegister("cz-target-only.txt", [target]);

    Reset(target);                            ④
}
```

- ① Prepare the control qubit in $|+\rangle$, and the target qubit in $|1\rangle$.
- ② Apply CZ and save out the resulting state. Before you dump the state of the target qubit, though, let's measure and reset the control qubit.
- ③ Fun fact: this is actually how the Reset operation is implemented in Q#.
- ④ You already reset control above, so we don't need to reset it again here.

Listing 8.20. Output

```
# wave function for qubits with ids (least to most significant): 1
|0⟩: 0.000000 + 0.000000 i == [ 0.000000 ]
|1⟩: -1.000000 + 0.000000 i == **** [ 1.000000 ] --- [ 3.14159 rad ] ①
```

- ① As expected, we get that the target qubit stays in the $|1\rangle$ state, ready to feed into another CZ operation.

8.5.1 Controlling any operation

Thinking back to Lancelot's and Dagonet's game, it'd be really helpful if you could help Lancelot reuse the qubit that he passes into Dagonet's operation, so that he

doesn't have to re-prepare it every time. Thankfully, using controlled operations to implement phase kickback gives a hint as to how you could do it. In particular, when you used phase kickback in Chapters 6 to 7 to implement the Deutsch–Jozsa algorithm, the target qubit was in the $|-\rangle$ state both at the start and end of the algorithm. That means that Lancelot could re-use the same qubit for each round of his game, and not need to re-prepare each time. That didn't matter for Deutsch–Jozsa, since you only ran one round of Nimue's and Merlin's game, but it's exactly the right trick for Lancelot to win his game with Dagonet, so let's look at how you can help him use phase kickback.

The trouble is that, while phase kickback is a very helpful tool to have in your toolbox as a quantum developer, so far you have only seen how to use it with the X and Z operations. We know for our game, Dagonet told Lancelot that he will use the R1 operation; is there a way you can use phase kickback to help here? The pattern you used to implement phase kickback in the previous section only required you to be able to control an operation, so what you will need is a way to control the op::Apply operation that Dagonet gives Lancelot. In Q#, this is as simple as writing Controlled op::Apply instead of op::Apply, thanks to the Controlled functor. Much like the Adjoint functor that you saw in Chapter 6, Controlled is a Q# keyword that modifies how an operation behaves, in this case to turn it into its controlled version.

TIP

Just like is Adj indicates that an operation can be used with Adjoint, is Ctl in the type of an operation indicates that it can be used with the Controlled functor. If you want to denote that an operation supports both, you can write is Adj + Ctl. For example, the type of the X operation is (Qubit \Rightarrow Unit) is Adj + Ctl, letting you know that X is both adjointable and controllable.

Thus, to help Lancelot out, you can take the op::Apply(scale, target) line and make it Controlled op::Apply([control], (scale, target)) and you have the controlled version of R1.

While that does solve Lancelot's problem, it can be helpful to unpack what's happening under the hood just a little bit more. Any unitary operation (that is, a quantum operation that doesn't allocate, deallocate, or measure qubits) can be controlled, the same way you controlled the Z operation to get CZ, and as you controlled X to get CNOT. For instance, we can define a controlled-controlled-NOT (CCNOT, also known as Toffoli) operation as an operation that takes two control qubits and flips its target if *both* controls are in the $|1\rangle$ state. Mathematically, we write that the CCNOT operation transforms an input state $|x\rangle|y\rangle|z\rangle$ to the output $|x\rangle|y\rangle|z\rangle \text{XOR}(y \text{ AND } z)$. We can also write down a matrix that lets us simulate the CCNOT operation:

Equation 8.7: Simulating the CNOT operation with a unitary matrix.

$$U_{\text{CCNOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Similarly, the controlled-SWAP operation (also known as the Fredkin operation) transforms its input states from $|1\rangle|z\rangle|y\rangle$, and leaves its input the same when the first qubit is in the state $|0\rangle$.

TIP We can make a controlled-SWAP out of three CCNOT operations: $\text{CCNOT}(a, b, c)$; $\text{CCNOT}(a, c, b)$; $\text{CCNOT}(a, b, c)$; is equivalent to $\text{Controlled SWAP}([a], [b], [c])$. To see this, note that we can also make the uncontrolled SWAP operation from three CNOT operations for the same reason that we can swap two classical registers in-place using a sequence of three classical XOR operations.

We can generalize this pattern for any unitary operation U (that is, any operation which neither allocates, deallocates, nor measures its qubits). In Q#, the transformation performed by using the Controlled functor adds a new input to an operation representing which qubits should be used as controls.

TIP This is where the fact that Q# is a tuple-in tuple-out language comes in very handy. Since every operation takes exactly one input, for any operation U , $\text{Controlled } U$ takes the original input to U as its second input.

In fact, the CNOT and CZ operations are simply shorthand for appropriate calls to `Controlled`. In [8.1](#), you can see more examples of this pattern.

TIP Just like `Adjoint` works on any operation that has `is Adj` in its type (as you saw in Chapter 6), the `Controlled` functor works on any operation that has `is Ctl` in its type.

Table 8.1. Some examples of controlled operations in Q#

Description	Shorthand	Definition
Controlled-NOT	<code>CNOT(control, target)</code>	<code>Controlled X([control], target)</code>
Controlled-Controlled-NOT (Toffoli)	<code>CCNOT(control0, control1, target)</code>	<code>Controlled X([control0, control1], target)</code>

Controlled-SWAP (Fredkin)	n/a	Controlled SWAP([control], (target1, target2))
Controlled Y	CY(control, target)	Controlled Y([control], target)
Controlled-PHASE	CZ(control, target)	Controlled Z([control], target)

As you saw with the CZ example, controlling operations in this way lets you turn global phases such as those applied to eigenstates into relative phases that we can learn through measurements.

More than that, by using controlled rotations to kickback phase onto the control register, you can also reuse the same target qubit over and over. When you applied CZ to a target register that was in an eigenstate of Z, that target register stayed in the same state, even though the control register changed. In the rest of this Chapter, you'll see how you can use that to finish Lancelot's strategy for his little game with Dagonet.

8.6 *Implementing Lancelot's best strategy for the phase estimation game*

You now have everything you need to write out a slightly different strategy for Lancelot that will allow him to use controlled operations to re-use the same qubits. As noted before, this may not make a huge impact for Dagonet's game, but does for other applications of quantum computing.

For example, in the next Chapter, you'll see how problems in quantum chemistry can be solved using a very similar game to the one that Dagonet and Lancelot are playing. There, however, preparing the right input state can require calling a lot of different quantum operations, such that if you can preserve the target qubit for later use, you can gain quite a lot of performance.

Let's review briefly what Lancelot's previous steps were before.

Steps for Dagonet and Lancelot's game

1. Dagonet picks a secret angle for a single qubit rotation operation
2. Dagonet prepares an operation for Lancelot to use that hides their secret angle, and allows Lancelot one additional input of a number (we'll call it a scale) that will get multiplied with the secret angle to give the total rotation angle of the operation.
3. Lancelot's best strategy for the game will be to select many scale values and estimate the probability of measuring One for each value. To do this, he will need to do steps a. and b. many times for each of the many scale values.
 - a. Prepare the $|+\rangle$ state and input the scale value in Dagonet's rotation. He uses the $|+\rangle$ state because he knows Dagonet is rotating around the Z axis, and for this state these rotations will result in a local phase change he can measure.
 - b. After preparing each $|+\rangle$ state, Lancelot can rotate it with the secret operation, measure the qubit, and record the measurement.
4. Lancelot will then have data relating his scale factor and the probability he

measured a 1 for that scale factor. He can then fit this data in his head, and get Dagonet's angle from the fitted parameters (he *is* the greatest knight in the land). We can use Python to help us do the same!

5. The step that will need to change to leverage your newfound skills with *controlled* rotations is step 3. For step 3a, what will change is the *allocation* of the qubits. Rather than allocating, preparing and measuring one qubit per measurement, he can allocate one **target** qubit to rotate with Dagonet's black box, and instead allocate and measure **control** qubits. He still can repeat the measurements, but won't have to measure or re-prepare the **target** each time.
6. You could summarize these changes by re-writing step 3 like this:
7. Lancelot's best strategy for the game will be to select many scale values and estimate the probability of measuring One for each value. To do this, he will need to do steps a. and b. many times for each of the many scale values. He prepares one qubit the $|1\rangle$ state to use as the **target** for all of his measurements as it's an eigenstate of the hidden rotation.
 - a. Prepare a second **control** qubit in the $|+\rangle$ state,
 - b. Apply the new controlled version of the secret rotation with Lancelot's scale value, unprepare the **control** qubit and measure it, then record the measurement.

In your code, these changes can be accomplished by modifying the previous `EstimateProbabilityAtScale` operation. Since the rotation axis could be anything Dagonet chooses (here it's just `Zaxis` for convenience), Lancelot needs to know how to control an arbitrary rotation. You can do this with the `Controlled` functor before calling the `ScalableOperation` that is passed from Dagonet. The `Controlled` functor is very similar to the `Adjoint` functor, in that it takes an operation and returns a new operation. `Controlled U(control, target)` is an example of the syntax that would allow you to apply `U` to your **target** qubit, controlled on one or more **control** qubits. Take a look at [8.16](#) for how you can modify `EstimateProbabilityAtScale` to use the `Controlled` functor.

Listing 8.21. Operation EstimateProbabilityAtScaleUsingControlledRotations

```
operation EstimateProbabilityAtScaleUsingControlledRotations(
    target : Qubit,
    scale : Double,
    nMeasurements : Int,
    op : ScalableOperation)
: Double {
    mutable nOnes = 0;
    for (idx in 0..nMeasurements - 1) {
        using (control = Qubit()) {
            within {
                H(control);
            } apply {
                Controlled op::Apply(
                    [control],
                    (scale, target)
1
2

```

```

        );
    }
    set nOnes += Meas.MResetZ(control) == One
        ? 1
        | 0;
}
return Convert.IntAsDouble(nOnes) /
    Convert.IntAsDouble(nMeasurements);
}

```

- ➊ Now instead of allocating and preparing the target register, our guessing operation takes the target register as an input and reuses it. Thus, we only need to allocate and prepare the control register; we can do so with an H operation, regardless of what rotation Dagonet is hiding.
- ➋ The only other change we need to make is to call Controlled op::Apply instead of op::Apply, passing our new control qubit along with the original inputs.

The other modification you will have to make (step five) is the operation that actually runs the game. Since using the controlled operation actually allows Lancelot to reuse the target qubit, it only needs to be allocated once at the beginning of the game. See [8.17](#) for how you can implement this.

Listing 8.22. Implementing the operation RunGameUsingControlledRotations.

```

operation RunGameUsingControlledRotations(hiddenAngle : Double,
                                         scales : Double[],
                                         nMeasurementsPerScale : Int)
: Double[] {
    let hiddenRotation = HiddenRotation(hiddenAngle);
    using (target = Qubit()) {
        X(target);
        let measurements = Arrays.ForEach(
            EstimateProbabilityAtScaleUsingControlledRotations(
                target,
                ,
                nMeasurementsPerScale,
                hiddenRotation
            ),
            scales
        );
        X(target);
        return measurements;
    }
}

```

➊
➋

- ➊ Using EstimateProbabilityAtScaleUsingControlledRotations, you can now allocate the target qubit once, since you'll be using it over and over again through each guess.
- ➋ Using the X operation, you can prepare the target in the $|1\rangle$ state, an eigenstate of the (uncontrolled) R1 operation that Dagonet hid his angle in. Since each measurement uses phase kickback to affect only the control register, this preparation can be done once before playing the game.

8.7 Summary

In this Part of the book, you've had a lot of fun using Q# and quantum computing to help the various denizens of Camelot. Using a quantum random number generator written in Q#, you were able to help Morgana pull one over on poor Lancelot. At the same time, you were able to help Merlin and Nimue each play their respective roles in deciding the fate of kings, learning about the Deutsch–Jozsa algorithm and phase kickback all the while. With the land at peace and the fires in Castle Camelot burning down for the night, you saw how to use everything you learned to help Lancelot play another game, winning this time by guessing a quantum operation hidden by Dagonet.

Throughout your Camelot escapades, you picked up quite a few new tricks to help you on your way as a quantum developer:

You have learned as a quantum developer:

- What is a quantum algorithm, and how to implement it with the Quantum Development Kit and Q#,
- How to use Q# from Python and Jupyter Notebook,
- How to design *oracles* to represent classical functions in quantum programs,
- User defined types,
- Controlled operations, and
- Phase kickback.

Going forward, it's time to bring what you've learned from Camelot back home, and apply these new techniques to something a bit more practical. In the next Chapter, you'll see how quantum computing can help in understanding chemistry problems. Don't worry if you don't remember the periodic table, you'll will be working with some colleagues who know the chemistry side of things, and are looking for your help in using everything that you learned in this Part to upgrade their workflow with quantum technology.

In this chapter you learned:

- Predict how quantum operations can be used to learn useful information about unknown operations with the quantum algorithm for phase kickback.
- Create new types in Q#
- Run Q# code from a Python host program
- Recognize important properties and behaviors of eigenstates and phase
- Program controlled quantum operations in Q#

Installing Required Software



In this appendix, you will learn:

- How to install a Python environment that you can use to design your own quantum simulation stack, and
- How to install the Microsoft Quantum Development Kit, including a compiler for the Q# language.

A.1 *Installing a Python Environment*

In the first several chapters of the book, we make heavy use of Python as a tool to explore quantum programming. In doing so, we rely on several Python libraries that make it easier to write scientific programs. As a result, it's often easier to start from a Python *distribution* that packages Python along with other libraries and package management tools. We'll go through getting up and running with the Anaconda distribution, but you should be able to follow a similar procedure if you would prefer to use a distribution such as Enthought Python, or Python(x, y).

A.1.1 *Installing Anaconda*

To install Anaconda, follow the instructions at docs.anaconda.com/anaconda/install/.

WARNING

At the time of this writing, the Anaconda distribution is provided with either Python 2.7 or 3.7. Python 2.7 has officially reached end-of-life, and is provided for compatibility reasons only. We will assume Python 3.6 or later in this book, so make sure that you install a version of Anaconda that provides Python 3.

NOTE

The Anaconda installer will offer to also install Visual Studio Code, an open source and cross platform IDE from Microsoft. While this is not required to get your Python environment up and running, we will later make use of Visual Studio Code together with the Quantum Development Kit, so we suggest allowing Anaconda to install it at this step.

A.1.2 *Installing Python packages with Anaconda: QuTiP*

Packages are a great way to collaborate and save time when you are trying to learn or develop new code. They are a way of collecting related code and wrapping it up in such a way it is easy to share to other machines. Packages can be installed on your machine with what is sensibly called a package manager, of which there are a few common options for Python. The reasons you might choose one manager over the other is each one has its own listing of packages and the package you want to install may only be known by a particular manager (depending on how the author deployed it). Let's start by looking the package managers we have already installed as a part of Anaconda.

As a default, Anaconda comes with two package managers pip and conda. Given we have installed Anaconda, the conda package manager has some additional features to pip that make it a good default choice for package management. conda has support for installing dependencies automatically when you install a package, it has the concept of environments (See sidebar) that are really helpful for creating dedicated Python sandboxes? for each project you are working on. A good general strategy is thus to install packages from conda when they are available, and to install packages from pip otherwise.

NOTE

The conda package manager can be used with most common command line environments, but if you would like to use conda with PowerShell, you will need version 4.6.0 or later. To check your version, run `conda --version`. If you need to update, run `conda update conda`.

Conda Environments

So far, we've simply run commands like `pip install` and `conda install` to install new packages for our entire Anaconda installation. We may run into the situation, however, that the packages we need for two different projects contradict each other. To help isolate projects from each other, the Anaconda distribution provides `conda env` as a tool to help manage multiple *environments*. Each environment is a completely independent copy of Python with only the packages needed for a particular project or application. Environments may even use different versions of Python from each other, with one environment using 2.7 and another using 3.7. Environments are also great for collaborating with others, as you can send your teammates a single small text file, `environment.yml`, to tell their `conda env` how to create an environment that's identical to yours.

For more information:

docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html

Following that strategy, let's start by making a new environment with the packages we need. Run the following at your favorite command line:

Listing A.1. Creating a new conda environment

```
conda create -n=quantum -c conda-forge python>=3.6 ipython numpy matplotlib
notebook qutip
```

This will create a new environment called `quantum` using packages from the `conda-forge` channel, and will install Python (version 3.6 or later), the IPython interpreter, NumPy, the matplotlib plotting engine, Jupyter Notebook, and QuTiP into the new environment. The `conda` package manager will prompt you to confirm the list of packages that will be installed, press "y" and then "Enter," and grab a cup of coffee.

Once `conda` has finished creating your new environment, let's give it a go. To test your new environment, first activate it:

Listing A.2. Activating the new conda environment

```
conda activate quantum
```

Once `quantum` has been activated, the `python` command should invoke the version installed into that environment. To check this, you can print the path to the `python` command for your environment. Run `python`, and then run the following at the Python prompt:

Listing A.3. Testing the new conda environment

```
>>> import sys; print(sys.executable)
C:\Users\Chris\Anaconda3\envs\quantum\python.exe ①
```

① Note that you might see a different path depending on your system.

If your environment was created successfully, you can use it either at the command line with IPython, or in your browser with Jupyter Notebook. To get started with IPython, run `ipython` from your command line: (Make sure that you've activated `quantum` first!)

Listing A.4. Using IPython from the quantum environment

```
$ ipython
In [1]: import qutip as qt
```

```
In [2]: qt.basis(2, 0)
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[1.]
 [0.]]
```

If you would like to learn more about using NumPy, continue reading in Chapter 2. If you would like to learn more about using QuTiP, continue reading in Chapter 4. If you would like to learn about the Quantum Development Kit, continue reading here.

For more information

- Anaconda documentation: docs.anaconda.com/anaconda/
- NumPy documentation: numpy.scipy.org/
- Jupyter Notebook documentation: jupyter-notebook.readthedocs.io/en/stable/notebook.html

A.2 *Installing the Quantum Development Kit*

TIP

The latest version of the installation instructions for the Quantum Development Kit can be found at docs.microsoft.com/quantum/install-guide/.

The Quantum Development Kit from Microsoft is a set of tools for working with and programming in Q#, a new language for quantum programming. The Quantum Development Kit can be used with either Visual Studio 2017 on Windows 10 or later, Visual Studio 2019 on Windows 10 or later, or with Visual Studio Code on Windows 10 or later, macOS, or on Linux.

NOTE

In this book we will focus on using Visual Studio Code, but the Quantum Development Kit can be used with any other text editor as well by following the command line instructions in the main text.

The Quantum Development Kit can also be used with Visual Studio 2017 or later by using the extension at marketplace.visualstudio.com/items?itemName=quantum.DevKit.

Using the installation of Visual Studio Code from setting up your Python environment, we need to do a few things to use the Quantum Development Kit with C#, Python, and Jupyter Notebook.

- Install the .NET Core SDK,
- Install the project templates for Q#,
- Install the Quantum Development Kit extension for Visual Studio Code,
- Install Q# support for Jupyter Notebook, and
- Install the qsharp package for Python.

Meet the .NETs

At this point, answering the question of what .NET is has become a bit more complicated than it used to be. Historically, ".NET" was a reasonable shorthand for the .NET Framework, a virtual machine and compiler infrastructure for any of the .NET languages (most notably, C#, F#, and Visual Basic .NET). The .NET Framework is Windows-only, but third-party reimplementations such as Mono exist for other platforms including macOS and Linux.

A couple years ago, though, Microsoft and the .NET Foundation open-sourced a new flavor of .NET called .NET Core. Unlike .NET Framework, .NET Core is cross-platform out of the box. \.NET Core is also much smaller, with much of the functionality separated out into optional packages. This makes it much easier to have multiple versions of .NET Core on the same machine, and for new .NET Core features to be introduced without compatibility issues.

This bifurcation of .NET into Framework and Core comes with its own foibles, however. To make .NET Core work better as a cross-platform programming environment, a few things in the .NET standard libraries were changed in ways that aren't entirely compatible with .NET Framework. To resolve this, the .NET Foundation introduced the concept of .NET Standard, a set of APIs offered by both .NET Framework and .NET Core. The .NET Core SDK can then be used to make libraries for either .NET Core or .NET Standard, and can build applications for .NET Core. Much of the libraries provided Quantum Development Kit target .NET Standard, so that Q# programs can be used from traditional .NET Framework applications or from new applications built using the .NET Core SDK.

Once you've done this, you have everything you need to write and run quantum programs written in Q#.

A.2.1 Installing the .NET Core SDK

To install the .NET Core SDK, go to dotnet.microsoft.com/download and select your operating system from the selections near the top of the page.

A.2.2 Installing the Project Templates

One thing that might be different than what you're used to is that both .NET Framework and .NET Core development center around the idea of a *project* that specifies how a compiler is invoked to make a new binary. For instance, a C# project (*.csproj) file tells the C# compiler what source files should get built, what libraries are needed, what warnings are turned on and off, etc. In this way, project files work similarly to makefiles or other build management systems. The big difference is in how project files on .NET Core reference libraries.

A project file can specify one or more references to *packages* on NuGet.org, a package repository for .NET Framework and .NET Core software. Each package can then provide a number of different libraries. When a project that depends on a NuGet package is built, the .NET Core SDK will automatically download the right package, and then will use the libraries in that package to build the project.

From the perspective of quantum programming, this allows for the Quantum Development Kit to be distributed as a small number of NuGet packages that can be

installed not on a machine, but into each project. This makes it easy to use different versions of the Quantum Development Kit on different projects, or to include only the parts of the Quantum Development Kit that you need for a particular project. To help get started with a reasonable set of NuGet packages, the Quantum Development Kit is provided with templates for creating new projects that reference everything you need.

To install the project templates, run the following command at your favorite terminal:

Listing A.5. Installing the project templates for the Quantum Development Kit

```
dotnet new -i "Microsoft.Quantum.ProjectTemplates"
```

Once the project templates are installed, you can use them by running dotnet new again:

Listing A.6 Creating a new project with the Quantum Development Kit project templates

```
dotnet new console -lang Q# -o ProjectName ①
```

- ① Make sure to replace ProjectName by the name of the project you would like to create.

A.2.3 *Installing the Visual Studio Code extension*

NOTE

This section assumes that you installed Visual Studio Code with your Anaconda installation above. If you do not have Visual Studio Code, you can install it manually from code.visualstudio.com/.

To install the extension for Visual Studio Code, open a new Visual Studio Code window and press **Ctrl+Shift+X** (Windows and Linux) or **⌘+Shift+X** to bring up the extensions sidebar. From the search bar, type in "Microsoft Quantum Development Kit," and press the "Install" button. Once Visual Studio Code has installed the extension, the "Install" button will change to a "Reload" button, which will close Visual Studio Code and reopen your window with the Quantum Development Kit extension installed.

Alternatively, press **Ctrl+P** or **⌘+P** to bring up the Go To palette. In the palette, type `ext install quantum.quantum-devkit-vscode` and press Enter.

In either case, once the extension has been installed, to use it, open a folder (**Ctrl+Shift+O** or **⌘+Shift+O**) containing the Q# project you'd like to work on. At this point, you should have everything you need to get up and programming with the Quantum Development Kit!

For more information

- Quantum Development Kit documentation: docs.microsoft.com/quantum
- Using the `dotnet` command: docs.microsoft.com/en-us/dotnet/core/tools/dotnet
- Getting started with Visual Studio Code: code.visualstudio.com/docs/introvideos/basics

A.2.4 Installing IQ# for Jupyter Notebook

Run the following from your favorite command line:

```
dotnet tool install -g Microsoft.Quantum.IQSharp
dotnet iqsharp install
```

TIP

On some Linux installations, you may need to run `dotnet iqsharp install --user` instead.

A.2.5 Installing the qsharp Python package

Run the following from your favorite command line:

```
conda activate quantum
pip install qsharp
```

To test, make sure that you can import `qsharp` and print out the versions of each component.

NOTE

You may get slightly different version numbers here; that's OK!

```
>>> import qsharp
>>> qsharp.component_versions()
{'iqsharp': LooseVersion ('0.5.1903.2902'), 'Jupyter Core': LooseVersion
('1.1.12077.0'), 'qsharp': LooseVersion ('0.5.1903.2902')}
```