A hands-on approach

LEARN Quantum Computing with Python and Q#

Sarah Kaiser
Chris Granade

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Learn Quantum Computing with Python and Q#
A Hands-on approach
Version 1

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

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: Measuring At Very Small Scales

PART 3: APPLIED QUANTUM COMPUTING

9 Computing Chemistry Problems With Quantum Computers

10 Searching Databases With Quantum Computers

11 Arithmetic With Quantum Computers

APPENDIXES

A Installing Required Software

1 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:

Textbooks and other resources for learning further

- *Quantum Computing: A Gentle Introduction* by 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.

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.

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?

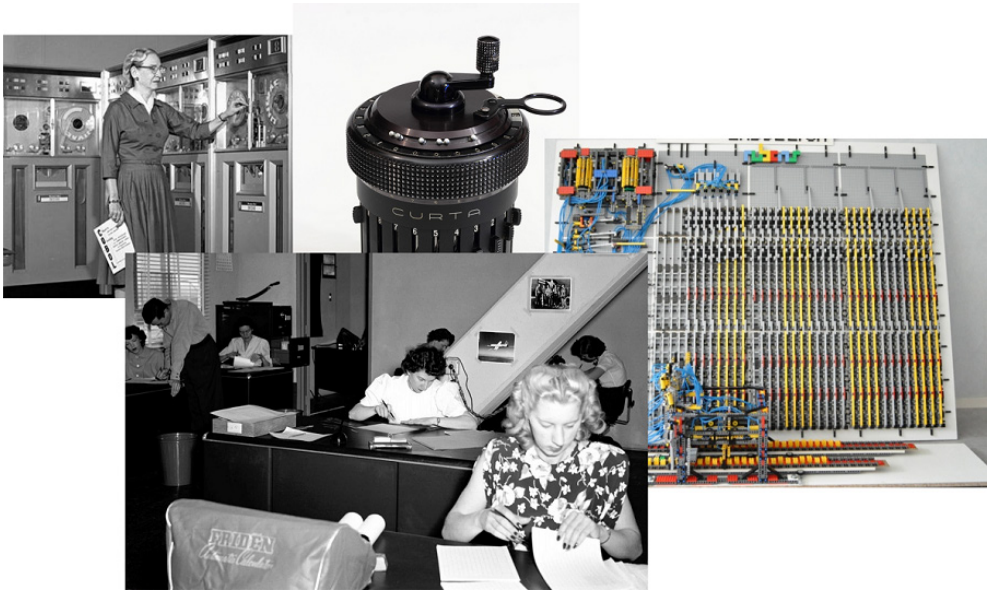
Let's talk a bit about *what* actually makes up a quantum computer. To facilitate this, let's briefly talk about what the term "computer" means. In a very broad sense, a computer is a device that takes data as input and does some sort of operations on that data.

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 Figure 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 Newtons'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 in to 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.

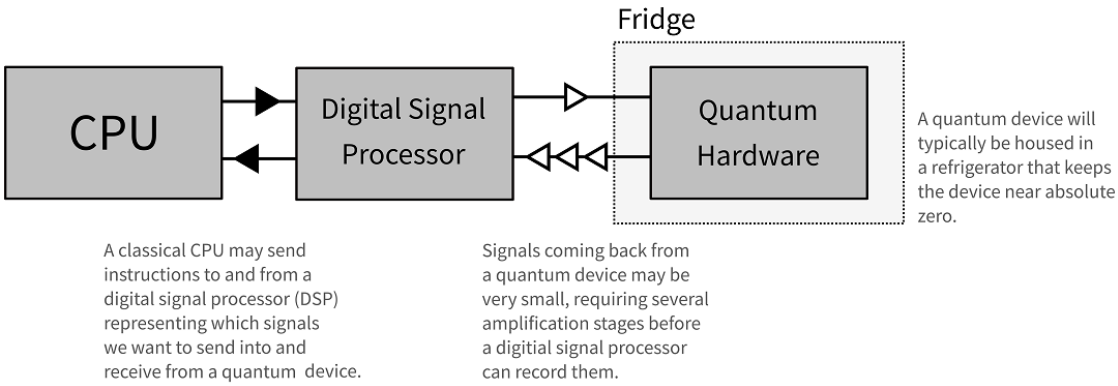
Other qubit devices may differ in the details of how they are controlled, but what remains consistent is that all quantum devices are controlled from and read out by classical computers and control electronics of some kind. After all, we are ultimately interested in classical data, and so there must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

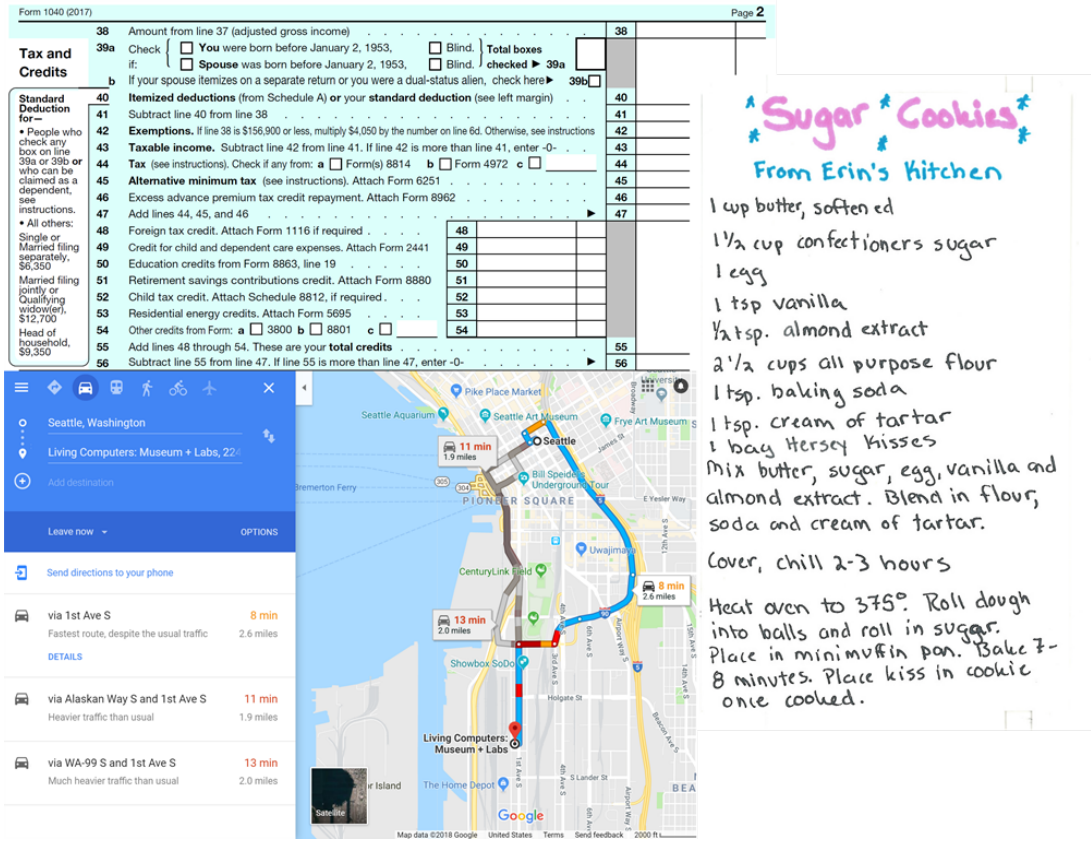
A program is a sequence of instructions that can be interpreted by a classical computer to perform a desired task.

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:

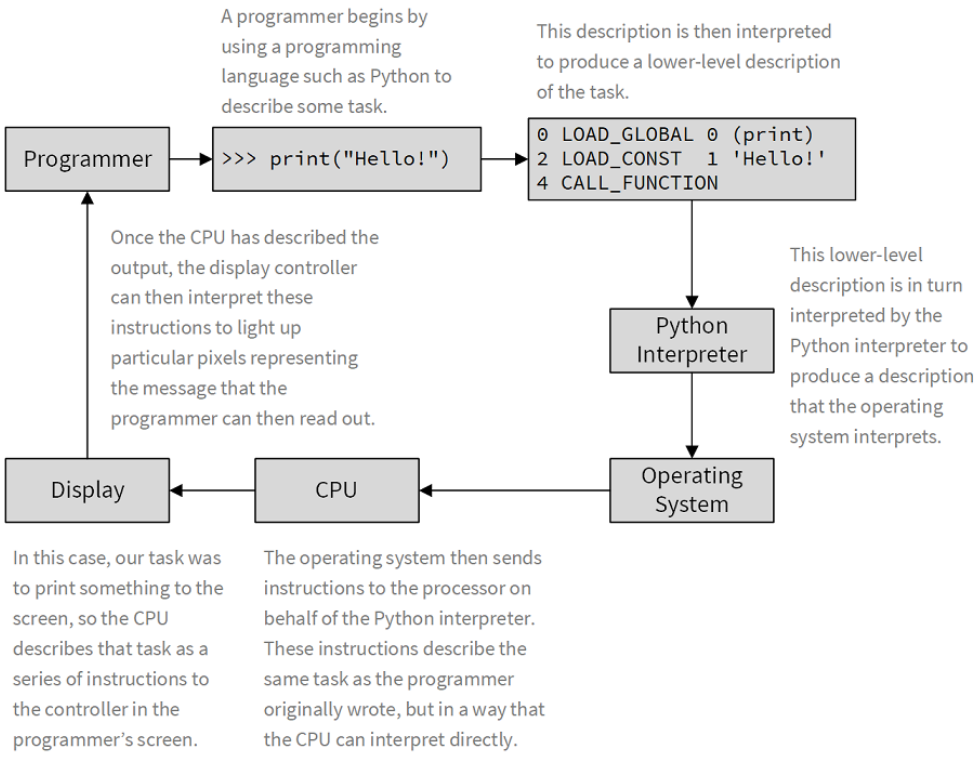
```
>>> import dis
>>> dis.dis(hello)
```

```
  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

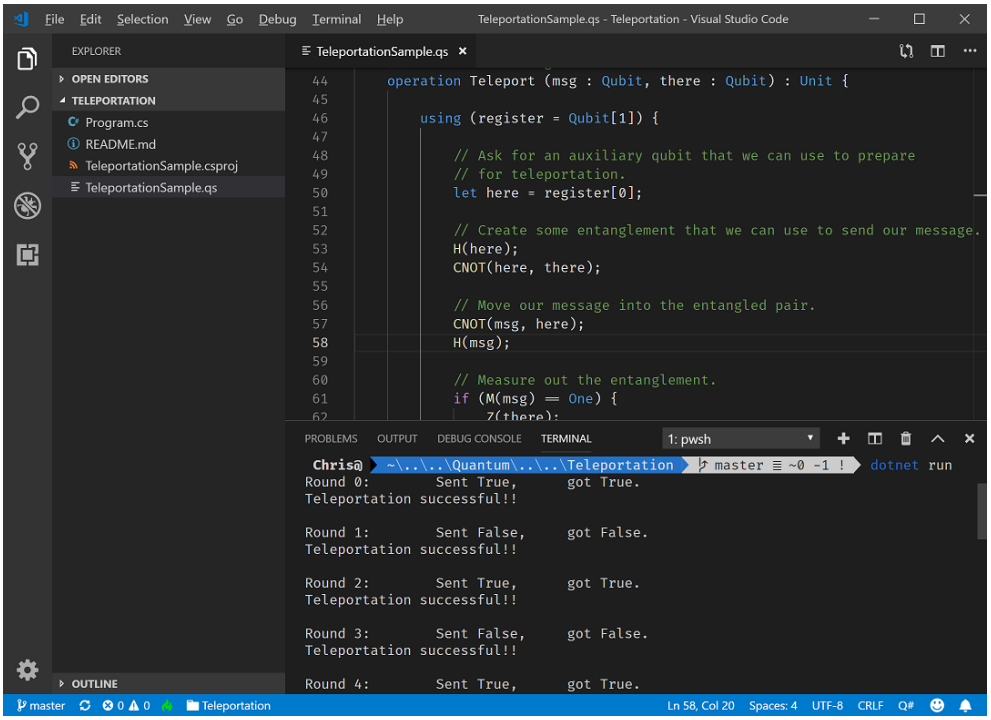
between description and interpretation motivates calling Python, C, and other such programming tools *languages*, emphasizing that programming is ultimately an act of communication.

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?

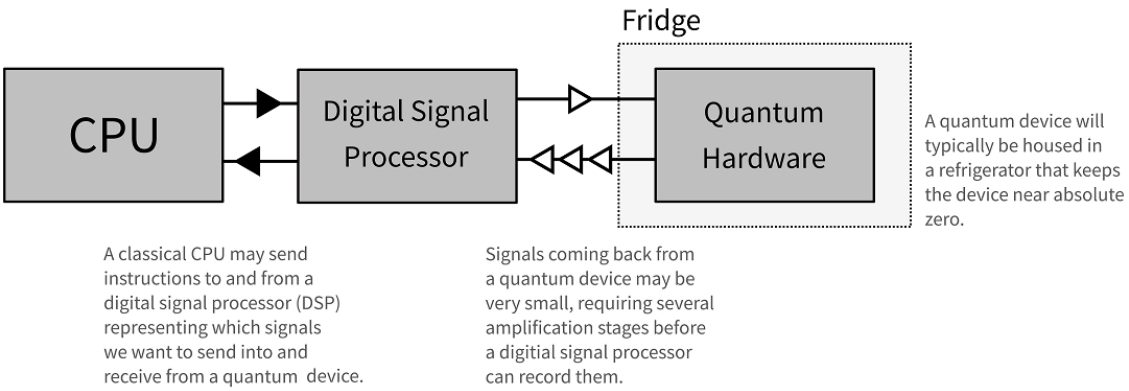
Like classical programs, quantum programs consist of sequences of instructions that are interpreted by classical computers to perform a particular task. The difference, however, is that in a quantum program, the task we wish to accomplish involves controlling a quantum system to perform a computation.

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



The 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 Figure 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.

2 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.

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)

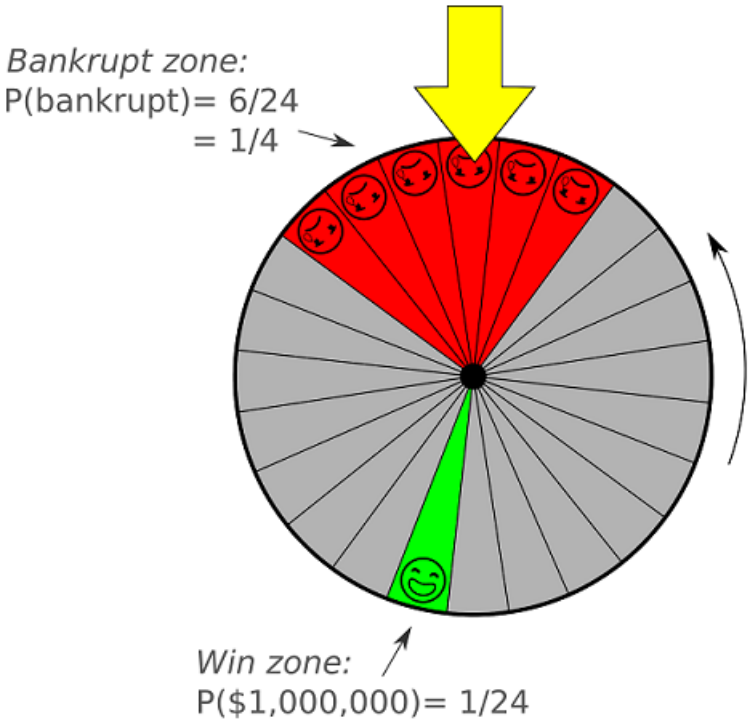
In all of these situations where we want randomness, we can describe the "chances" for each outcome. Being random events means that describing the chances is all we can say about the situation until the die is cast (or the coin is flipped or the password is re-used). When we describe the "chances" of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

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™ (Figure 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 may 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: priceonomics.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 :)

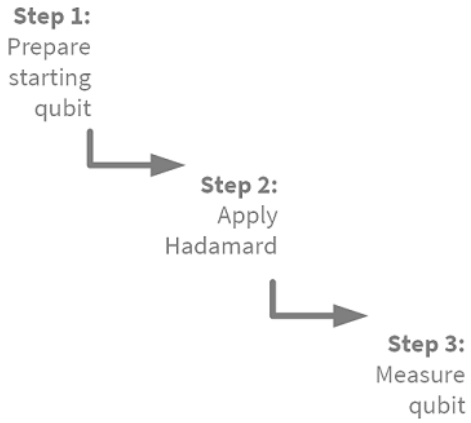
DEEP DIVE: Computational Security and Information Theoretic Security

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.

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

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

❶ 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.

❷ Quantum programs work by asking quantum computing hardware for *qubits*, quantum analogues of bits that we can use to perform computations.

3. 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.
4. 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 programming terms this means get a 1 or a 0. 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 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`, 𓍯 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 💮, 𓍯, 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.

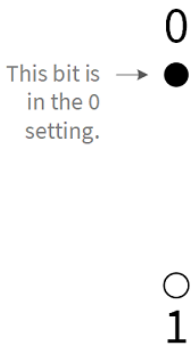
2.2.1 What Can We Do With Classical Bits?

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.

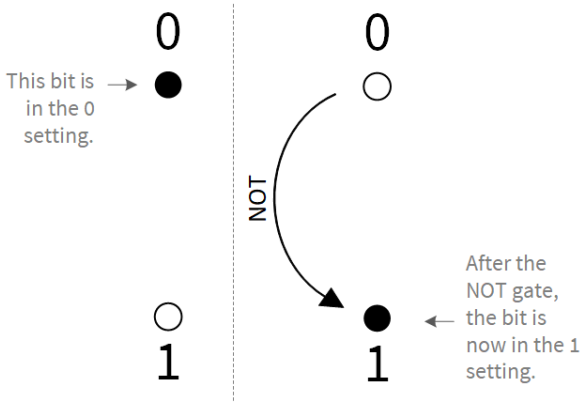
To visualize the NOT operation, let's imagine labeling two points as "0" and "1".

Figure 2.3. We depict a classical bit as a red 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.

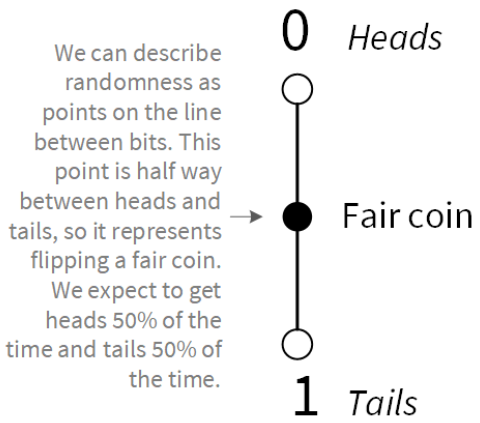
Figure 2.4. The classical NOT operation flips a 0 bit to a 1 bit and vice versa.



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 Figure 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*.

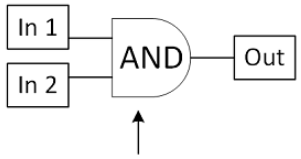
Definition 2.2: Truth table

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...

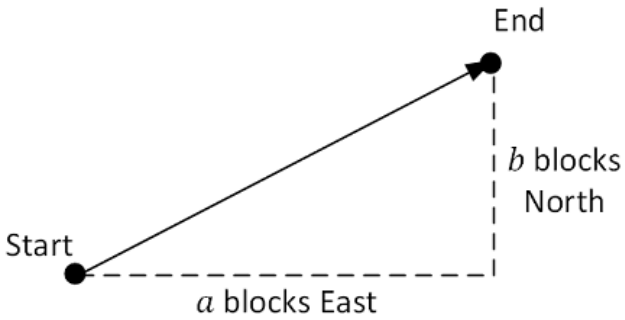
Vectors are a mathematical tool that can be used to represent a variety of different concepts — basically, anything that we can record by making an ordered list of numbers.

Example 2.1 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.

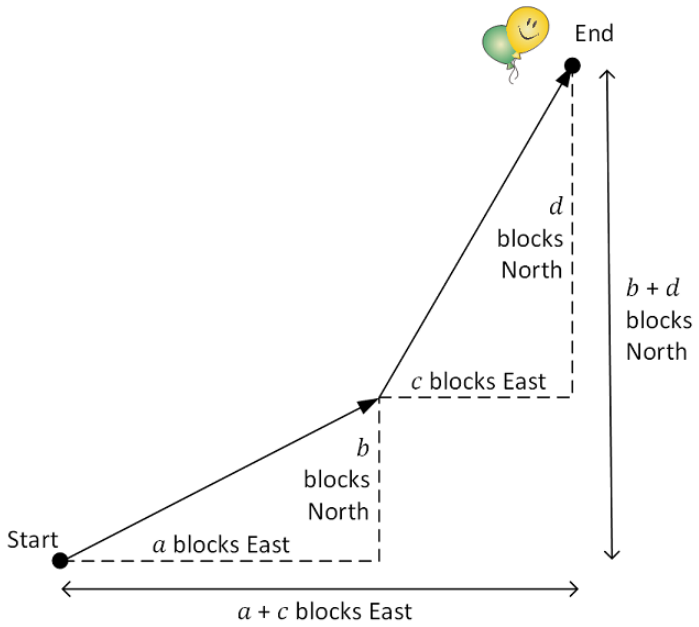


Like ordinary numbers, we can add different vectors together.

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]]`.

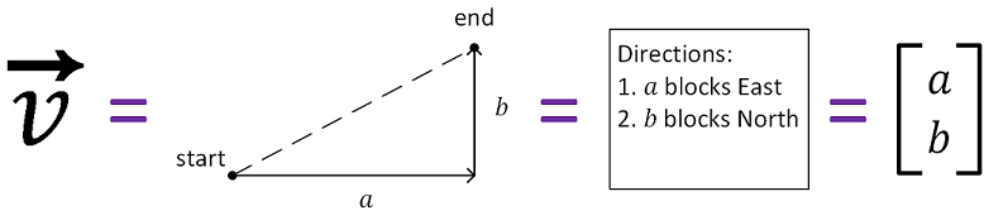
Figure 2.9. Adding vectors to find a party.



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(                    (1)
...     [[30], [50]])                                   (2)
>>> directions_in_feet = 3.28 * directions_in_meters    (3)
>>> directions_in_feet
array([[ 98.4],                                         (4)
       [164. ]])
```

(1) 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.

(2) Let's start with an example of going 30 meters East, then 50 meters North.

(3) NumPy represents multiplication between scalars and vectors by the Python multiplication operator `*`.

(4) 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 `swap(v + w)` is always the same as `swap(v) + 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: `swap(a * v) = a * 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.

Example 2.2 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 Figure 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 $\vec{b}_0$, $\vec{b}_1$, ..., $\vec{b}_{d-1}$, we say that $\vec{b}_0$, $\vec{b}_1$, ..., $\vec{b}_{d-1}$ 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)? We also know from looking parts of the map legend we see that the map warps the

horizontal direction to `np.array([[1], [1]])` and the vertical direction to `np.array([[1], [-1]])` _

Steps to compute `f(np.array([[2], [3]]))`

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

Listing 2.4. Using NumPy to help compute `f(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]])
```

(1) First, we define variables `horizontal` and `vertical` to represent the basis we will use to represent `[[2], [3]]`.

(2) We can write `[[2], [3]]` by adding multiples of `horizontal` and `vertical`.

(3) We next define how `f` acts on `horizontal` and `vertical` by introducing new variables `f_horizontal` and `f_vertical` to represent `f(horizontal)` and `f(vertical)`, respectively.

(4) 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

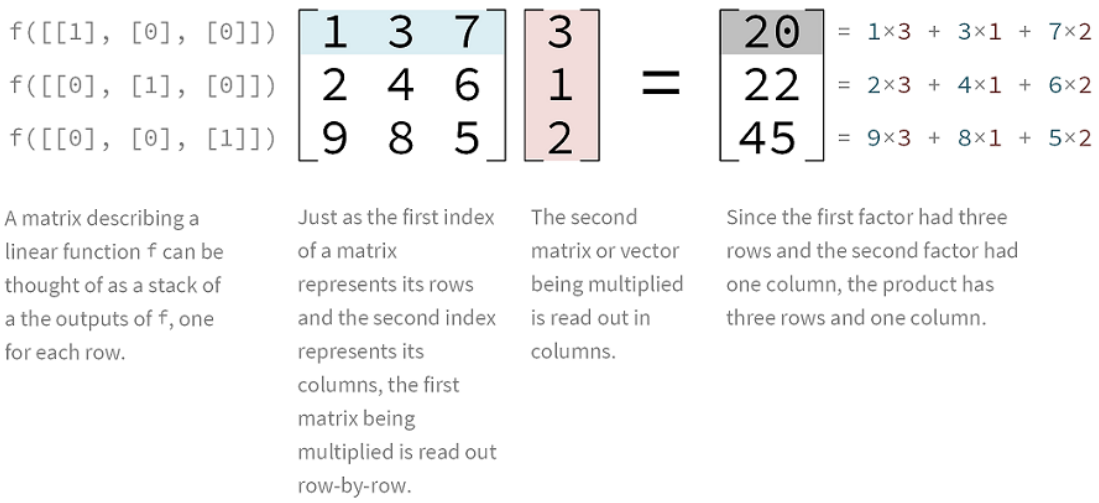
```
>>> swap_north_east = np.array([[0, 1], [1, 0]])
>>> swap_north_east
array([[0, 1],
       [1, 0]])
```

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 Figure 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], [3], [7]].



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]]` and `[[0], [1]]`, as illustrated in Figure 2.12 . 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 multplying even very small matrices by 10×. 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                                                    ❶
In [1]: def matmul(A, B):
   ...:     n_rows_A = len(A)                                ❷
   ...:     n_cols_A = len(A[0])
   ...:     n_rows_B = len(B)
   ...:     n_cols_B = len(B[0])
   ...:     assert n_cols_A == n_rows_B                      ❸
   ...:     return [                                         ❹
   ...:         [
   ...:             sum(                                     ❺
   ...:                 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                                   ❻
In [3]: X = np.array([[0+0j, 1+0j], [1+0j, 0+0j]])           ❼
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                                                ❽
Out[6]:
array([[ 0.+0.j, -1.+0.j],
       [ 1.+0.j,  0.+0.j]])
In [7]: %timeit matmul(X, Z)                                 ❾
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)
```

❶ 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.

❷ 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.

❸ 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.

❹ 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.

❺ 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 Figure 2.12 .

❻ For comparison, we can import NumPy, which provides us with a matrix multiplication implementation that uses modern processor instructions to accelerate the computation.

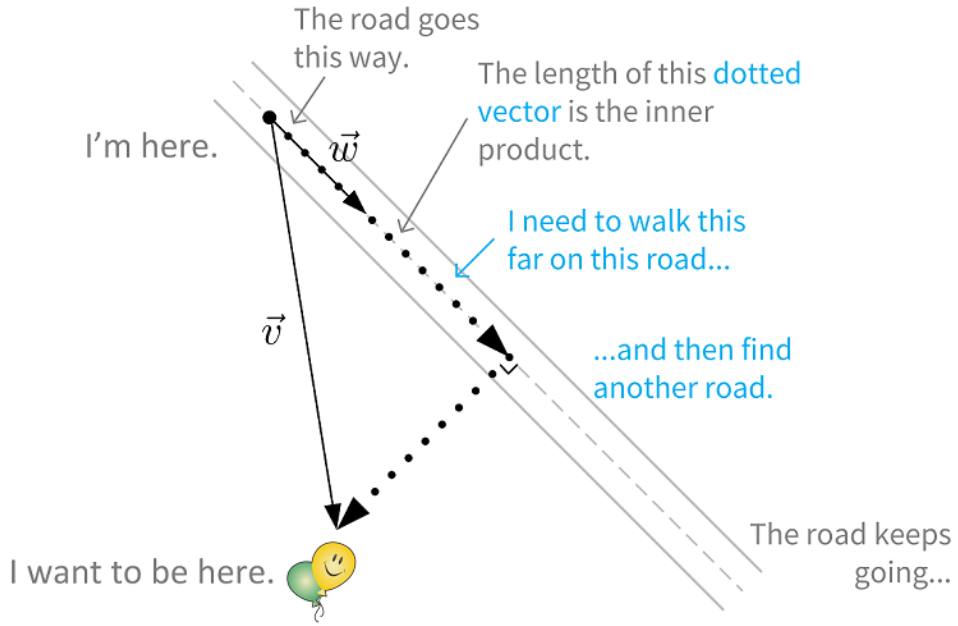
❼ We'll initialize two matrices as NumPy arrays as test cases. We'll see much more about these two particular matrices throughout the book.

8. Matrix multiplication in NumPy is represented by the `@` operator in Python 3.5 and later.
9. 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.

2.4.1 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 $\vec{v}^{\mathrm{T}}$..

Example 2.3

The transpose of $\vec{w} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$ is $\vec{w}^{\mathrm{T}} = \begin{pmatrix} 2 & 3 \end{pmatrix}$.

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 $\vec{v}^{\mathrm{T}}$ (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]])                    (1)
>>> south_east = np.array([[1], [-1]])            (2)
>>> np.linalg.norm(south_east)                    (3)
1.4142135623730951                                (4)
>>> w = np.array([[1], [-1]]) / np.sqrt(2)        (5)
>>> np.linalg.norm(w)                             (6)
0.9999999999999999
>>> v.transpose()                                 (7)
array([[2, 3]])
>>> v.transpose() @ w                             (8)
array([[ 0.70710678]])                            (9)
```

(1) In this case, $\vec{v}$ is the vector describing where I need to go, namely two blocks north and three blocks east.

(2) If the road available points southeast, then it goes one block south for every block east that it goes.

(3) We can find the length of a vector using Pythagoreas' 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 √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!

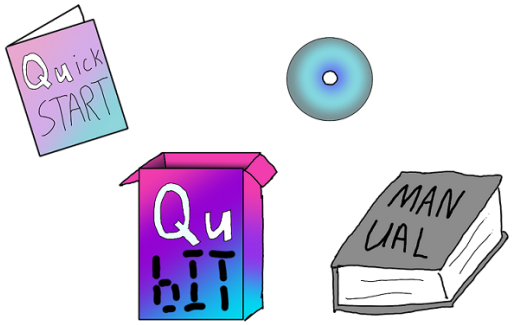
Qubits are the basic unit of information in a quantum computer. They can be physically implemented by systems that have 2 states like classical bits, but that behave according to the laws of quantum mechanics, which allows for some behaviors that classical bits are not capable of. Let's treat qubits 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 to 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 programing, 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.

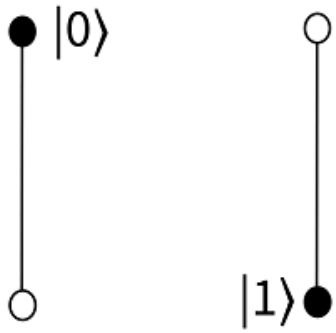
Similar to the 0 and 1 states of classical bits, we can write labels for quantum states. The qubit states that are most similar to the classical "0" and "1" are $|\boldsymbol{0}\rangle$ and $|\boldsymbol{1}\rangle$, as we draw in Figure 2.15 . These are referred to as "ket 0" and "ket 1" respectively. 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 is so difficult. 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. 😈

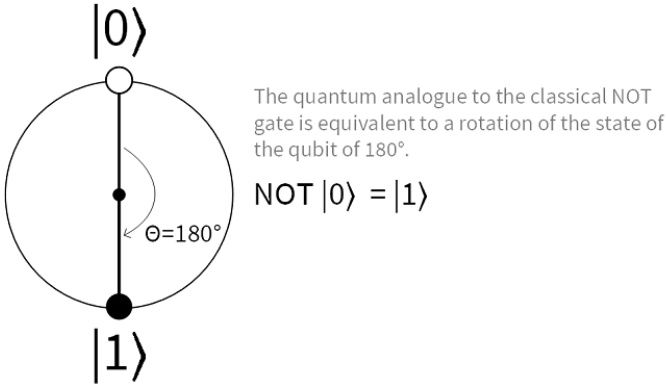
2.5.2 The game of Operations

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 is 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⟩ state, leaving the qubit in the |1⟩ state.



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

IMPORTANT **Reversibility**

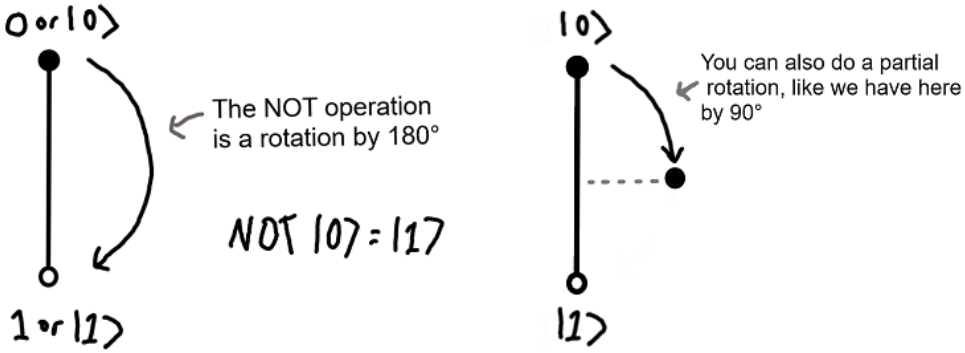
When we rotate a quantum state, we can always get back to the same state we started with by rotating backwards. This property, known as reversibility, turns out to be fundamental to quantum computing. With the exception of measurement, which we'll learn more about later in this chapter, all quantum operations 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 $|\mathbf{0}\rangle$ 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.

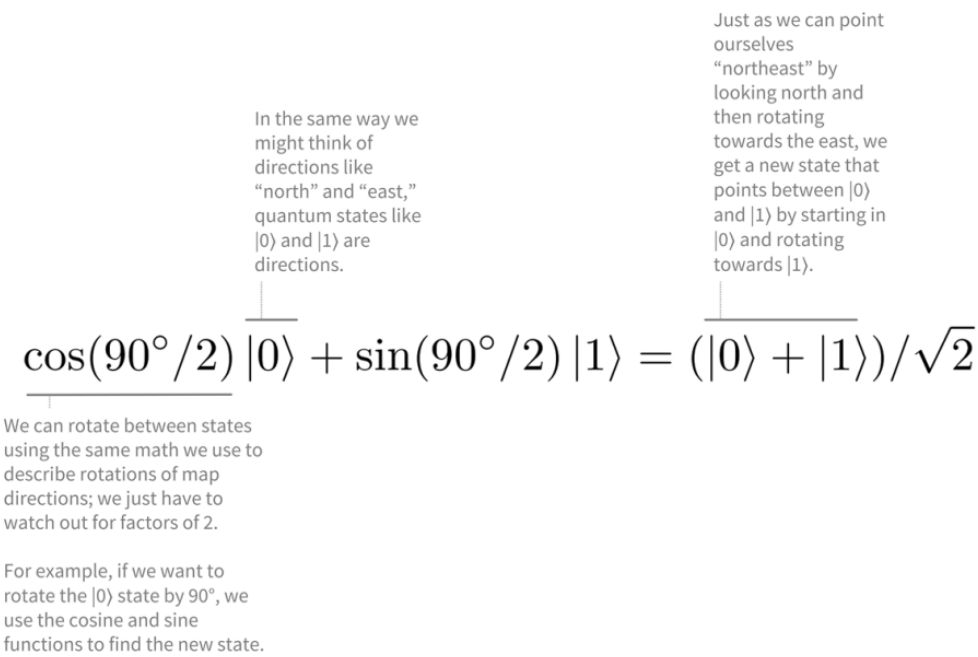
Halves and Halve-Nots

Every field has it 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 $|\mathbf{0}\rangle$ nor $|\mathbf{1}\rangle$, 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 Figure 2.18 .

Figure 2.18. Rotating a state by 90°.



Definition 2.6: |+⟩ and |−⟩

We call this state the |⟩ = (|0⟩ + |1⟩) / √2_* state (due to the sign between the terms). We say that the |⟩ state is a _superposition **of |0⟩ and |1⟩ . If the rotation was by a −90° (anti-clockwise), then we call the resulting state ***|-⟩ = (|0⟩ - |1⟩) / √2 instead. Try writing out the rotations above using −90° to see that you get |-⟩ !

A mouthful of math

At first glance, something like |+⟩ = (|0⟩ + |1⟩) / √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.

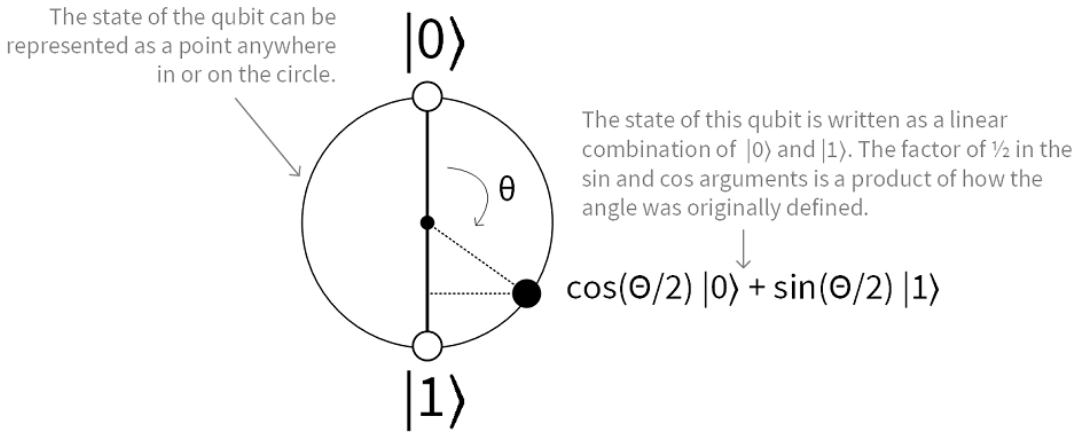
For instance, the "√2" part always has to be there, since vectors representing quantum states always have to be length one; that means we can sometimes be a little bit casual and write things like "|+⟩ = |0⟩ + |1⟩ ," relying on our audience to remember to divide by √2. If we're giving a talk, or discussing 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: $|\mathbf{0}\rangle$ and $|\mathbf{1}\rangle$

More generally, we will picture rotations by arbitrary angles θ between qubit states as follows in Figure 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 $\cos(\theta/2)\,|\mathbf{0}\rangle + \sin(\theta/2)\,|\mathbf{1}\rangle$, where $|\mathbf{0}\rangle$ and $|\mathbf{1}\rangle$ 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]] / √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 $|\mathbf{0}\rangle$ and $|\mathbf{1}\rangle$ with coefficients that describe the angle that $|\mathbf{0}\rangle$ 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 $|\mathbf{0}\rangle$ and $|\mathbf{1}\rangle$ 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.

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

$$= \begin{bmatrix}\cos(\theta/2)\\\sin(\theta/2)\end{bmatrix}$$

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 opeartions 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 / √2], [1 / √2]]
$\|-\rangle$	$(\|0\rangle - \|1\rangle) / \sqrt{2}$	[[1 / √2], [−1 / √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.

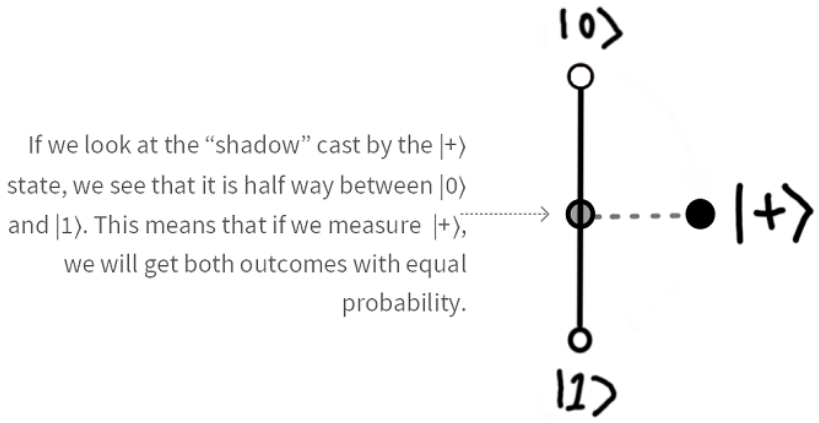
Definition 2.7: No-cloning Theorem

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

The No-cloning Theorem tells us something fundamental about how qubits are different than classical bits.

We'll see more about the No-cloning Theorem in Chapter 4, but the basic argument is that copying quantum information is not **linear**. 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.

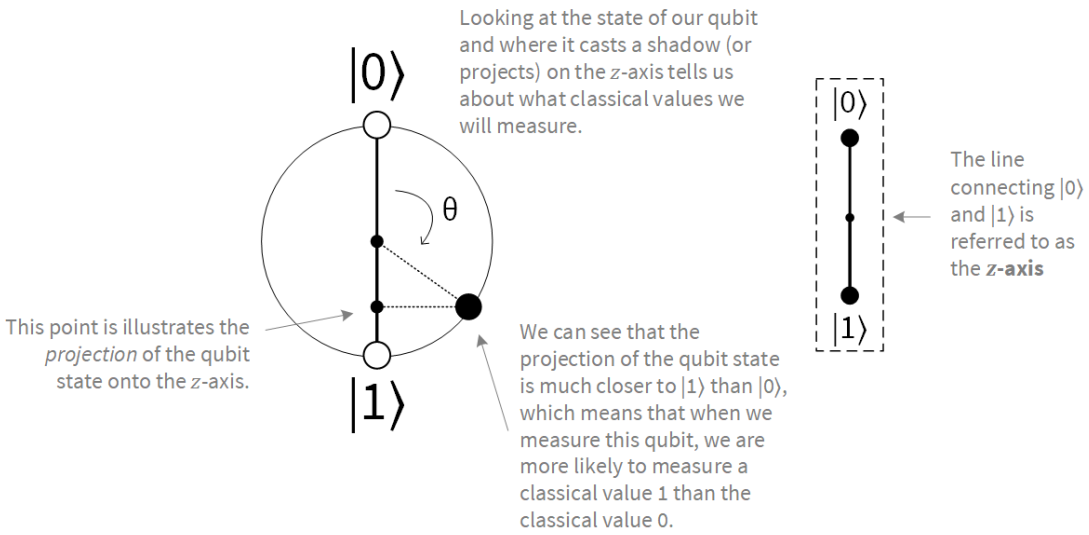
Figure 2.20. The $|+\rangle$ state overlaps equally with both $|0\rangle$ and $|1\rangle$, because the "shadow" it casts is exactly in the middle.



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 Figure 2.21), 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 intepretation*. 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 Figure 2.21, $|0\rangle$ 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 butting 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.

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

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.

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

Calculating each inner product makes things a lot simpler!

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

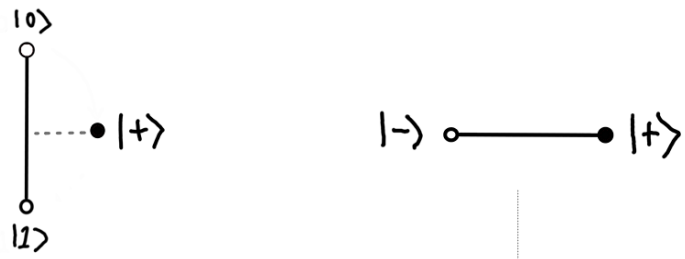
We now have the overlap between $|0\rangle$ and $|1\rangle$.

$= 1/\sqrt{2}.$

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 ½.

We will often project onto the $\boldsymbol{Z}$ axis because it is convenient in many real experiments, but we could have also measured along the $\boldsymbol{X}$-axis, to see if we have a $|+\rangle$ or a $|-\rangle$ state. Measuring along the $\boldsymbol{X}$ axis, we would have gotten $|+\rangle$ with certainty and would never have gotten $|-\rangle$.

Figure 2.22. Measuring $|+\rangle$ along the $\boldsymbol{X}$ 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 out come *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 us notation like " ⟨measurement | state⟩ " to represent measuring a qubit. The left component ⟨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 ⟨measurement | state⟩ , 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 Figure 2.21 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 $Pr(1 \mid +) = |\langle 1 \mid + \rangle|^2 = |\langle 1 \mid 0 \rangle + \langle 0 \mid 0 \rangle|^2 / 2 = |0 + 1|^2 / 2 = 1 / 2$. Thus, 50% of the time, we'll get a "1" outcome. The other 50% of the time, we'll get a "0" outcome.

Definition 2.8: Born's rule

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

$$\Pr(measurement \mid state) = |\langle measurement\} \mid state \rangle|^2.$$

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 Table 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.
$\|0\rangle$	$\langle 0\|$	$\|\langle 0 \mid 0\rangle\|^2 = 1$
$\|0\rangle$	$\langle 1\|$	$\|\langle 1 \mid 0\rangle\|^2 = 0$
$\|0\rangle$	$\langle +\|$	`\	⟨+ \	0⟩ \	² = \	(⟨0\	+ ⟨1\	) \	0⟩ / √2\	² = (1 / √2 + 0)² = 1 / 2`
$\|+\rangle$	$\langle +\|$	`\	⟨+ \	+⟩ \ |² = \ |(⟨0\ | + ⟨1\ |) (\ |0⟩ + \ |1⟩) / 2\ |² = \	⟨0 \	0⟩ + ⟨1 \	0⟩ + ⟨0 \	1⟩ + ⟨1 \	1⟩ \ |² / 4 = \	1 + 0 + 0 + 1 \ |² / 4 = 2² / 4 = 1`
$\|+\rangle$	$\langle -\|$	$\|\langle - \mid +\rangle\|^2 = 0$
$-\|0\rangle$	$\langle 0\|$	$\|-\langle 0 \mid 0\rangle\|^2 = \|-1\|^2 = 1^2$
$-\|+\rangle$	$\langle -\|$	`\ |- ⟨+ \	-⟩ \ |² = \ |(- ⟨0\	- ⟨1\ |) (\ |0⟩ - \ |1⟩) / 2\ |² = \	- ⟨0 \	0⟩ - ⟨1 \	0⟩ + ⟨0 \	1⟩ + ⟨1 \	1⟩ \ |² / 4 = \	-1 - 0 + 0 + 1 \ |² / 4 = 0² / 4 = 0`

TIP **In Table 2.3, we used that $\langle 0 \mid 0 \rangle = \langle 1 \mid 1 \rangle = 1$ and $\langle 0 \mid 1 \rangle = \langle 1 \mid 0 \rangle = 0$. (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 2.9: 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 Table 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

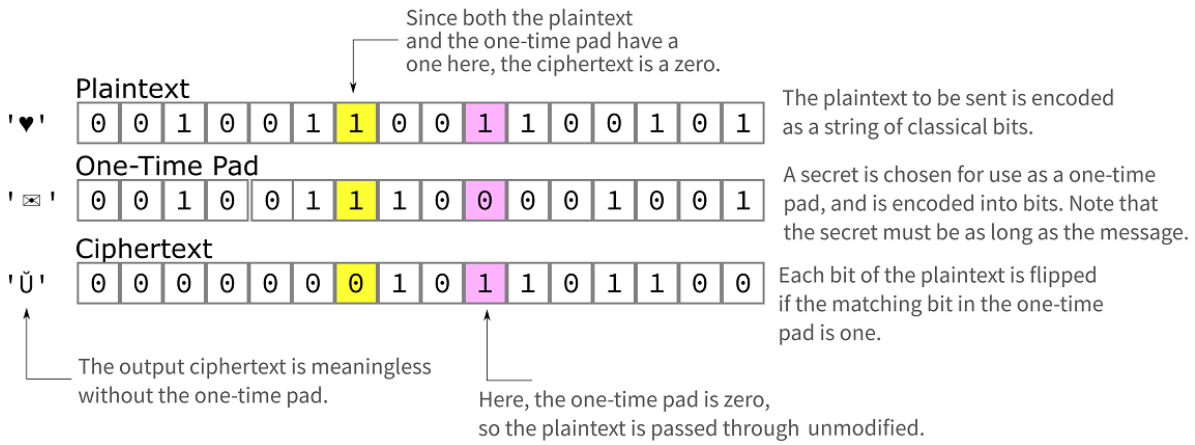
The quantum quick start guide lists cryptography as an initial application to check

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 Figure 2.23, 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 generator 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 2.10. Representing qubits in code with NumPy

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

(1) We use the NumPy library for Python to represent vectors, as NumPy is highly optimized and will make our lives much easier.

(2) We name our variable `ket0` after the notation $|0\rangle$, in which we label qubit states by the "ket" half of $\langle\rangle$ "brakets."

(3) 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 2.11. The vector representation of $|+\rangle$

```
>>> ket_plus = (ket0 + ket1) / np.sqrt(2)  (1)
>>> ket_plus
array([[0.70710678+0.j],                   (2)
       [0.70710678+0.j]])
```

(1) We can see NumPy using vectors to store the $|+\rangle$ state, which is a linear combination of $|0\rangle$ and $|1\rangle$.

(2) 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]]`.

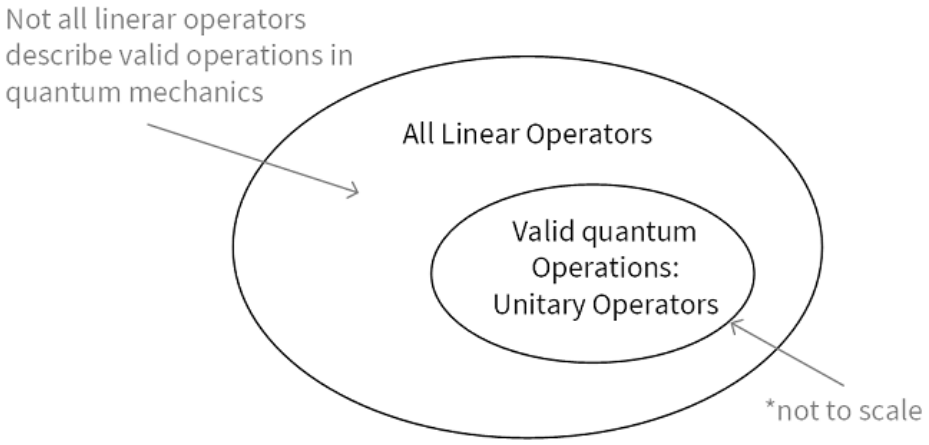
In classical logic, if we wanted to simulate how an operation would transform a list of bits, we could use a *truth table*. Similarly, since quantum operations other than measurement are always linear, to simulate how an operation transforms the state of a qubit, we can use a matrix that tells us how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



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 $\boldsymbol{Z}$-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 $|\boldsymbol{0}\rangle$ and $|\boldsymbol{1}\rangle$ 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 Table 2.4 as `H = np.array([[1, 1], [1, -1]]) / np.sqrt(2)`.

Listing 2.12. 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 Table 2.4 . We'll need `H` throughout the rest of this chapter, so it's helpful to define it here.

Definition 2.10: 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 $|\boldsymbol{0}\rangle$ to $|\boldsymbol{1}\rangle$ 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 2.13. 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                            (3)
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`.

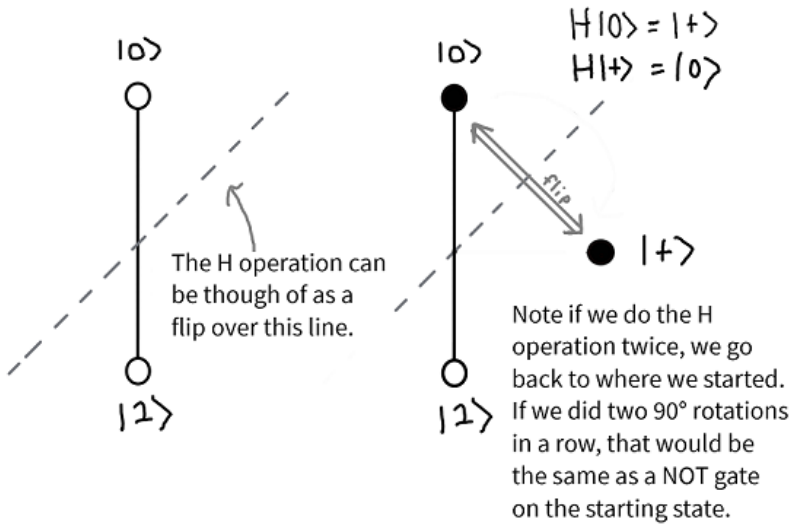
(2) We can confirm that `X` transforms $|0\rangle$ to $|1\rangle$. The NumPy method `all()` returns `True` if every element of `X @ ket0 == ket1` is `True`; that is, if every element of the array `X @ ket0` is equal to the corresponding element of `ket1`.

(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 @ ket0`.

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

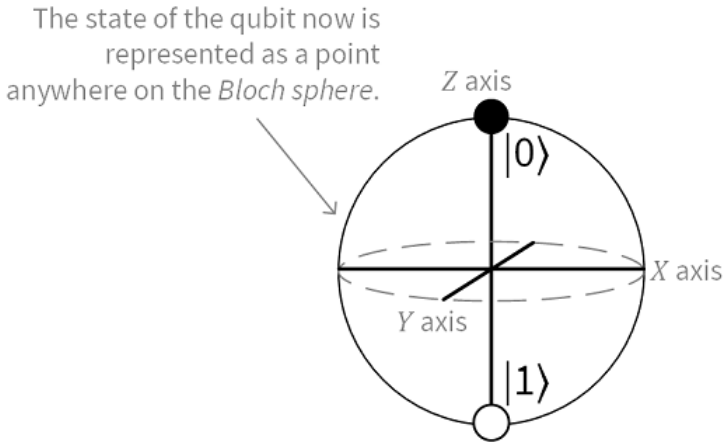
Figure 2.25. The `H` operation as a reflection or flip about ↗.



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 4.

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



DEEP DIVE: Infinitely many states?

It may seem from Figure 2.26 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 is 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 $4\,r_2 + 2\,r_1 + r_0$.

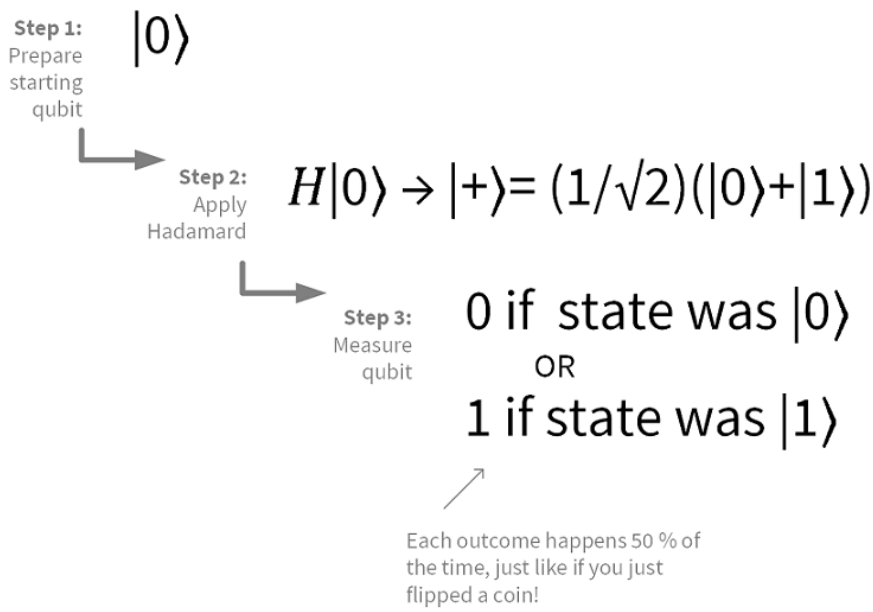
The case is slightly more tricky if N isn't given by a power of two, in that we have "left over" possibilities that we need to deal with. For instance, if we need to roll a six-sided die, but only have an eight-sided die on hand (maybe we played a Druid last time at RPG night), then we need to decide 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:

QRNG

1. Prepare a qubit in the state $|\mathbf{0}\rangle$.
2. Apply the Hadamard operation to our qubit, so that it is in the state $|+\rangle = \mathbf{H}|\mathbf{0}\rangle$.
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:

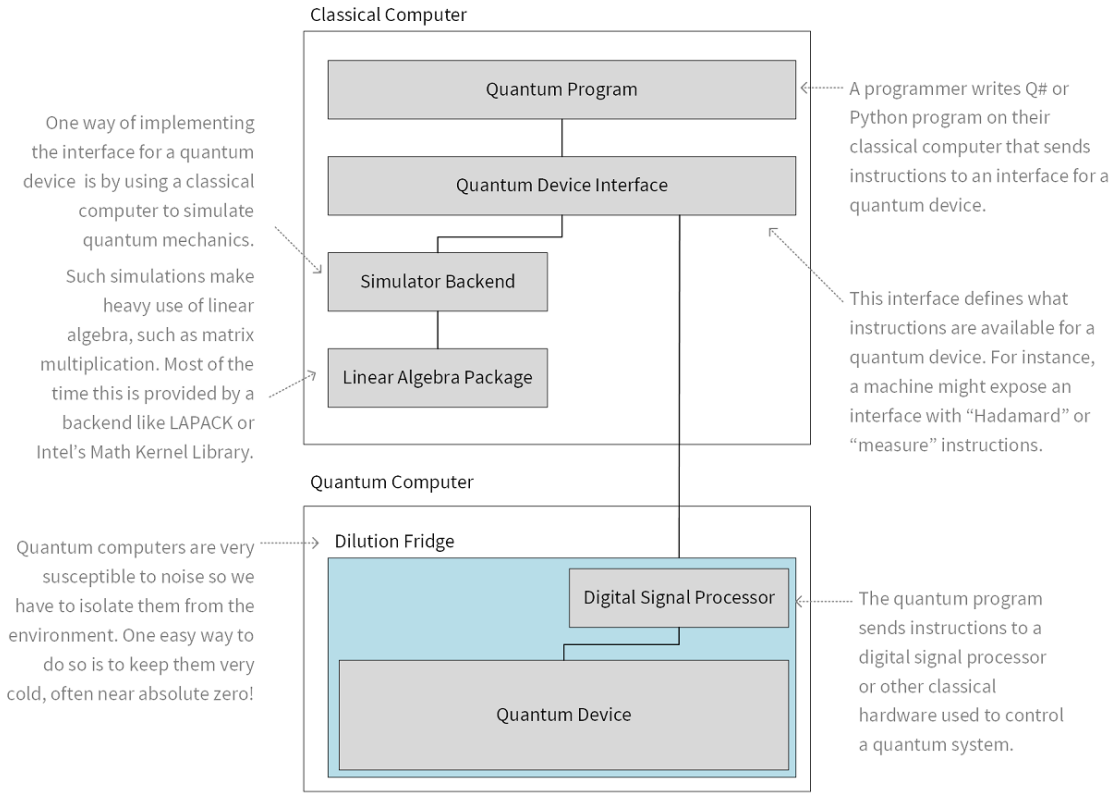
Listing 2.14. Example pseudocode for a QRNG program

```
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 Figure 2.28 , let's list out what we need our quantum device to be able to do:

Quantum device interface requirements.

- Users must be able to allocate and return qubits.

Listing 2.15. 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:

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 2.16. 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      ❶

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

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

❶ 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)`.
❷ The `measure` method can be implemented to allow users to measure qubits and extract classical data.
❸ 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 2.17. 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 2.18. 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 $|\boldsymbol{0}\rangle$ and the matrix representation of the Hadamard operation $\boldsymbol{H}$.

Listing 2.19. simulator.py

```
KET_0 = np.array([
    [1],
    [0]
], dtype=complex)              ❶
H = np.array([
    [1, 1],
```

```
    [1, -1]
], dtype=complex) / np.sqrt(2) ❷
```

❶ Since we'll be using |0⟩ a lot in our simulator it helps to define a constant for it.

❷ 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 2.20. simulator.py

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

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

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

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

❶ As a part of the `Qubit` interface, we ensure that the `reset` method prepares our qubit in the |0⟩ state. We can use that when we create the qubit to make sure that qubits always start in the correct state.

❷ 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.

❸ We stored the state of our qubit as a vector, so we know that the inner product with |0⟩ is simply the first element of that vector. For instance, if the state is `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 `np.abs(a) ** 2`. This gives us the probability that a measurement of our qubit returns 0.

❹ 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`.

❺ 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 2.21. 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.

6 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.

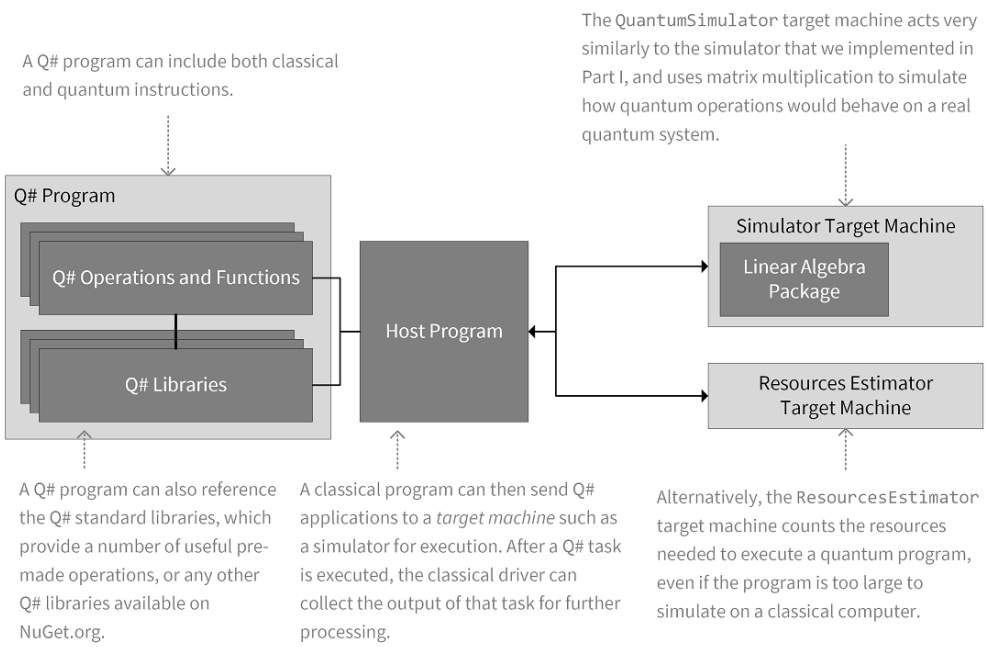
Up to this point, we've used Python to implement our own software stack to simulate quantum programs. Moving forward, though, we'll be writing more intricate quantum programs that will benefit from specialized language features that are hard to implement by embedding our software stack inside Python. 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.



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.

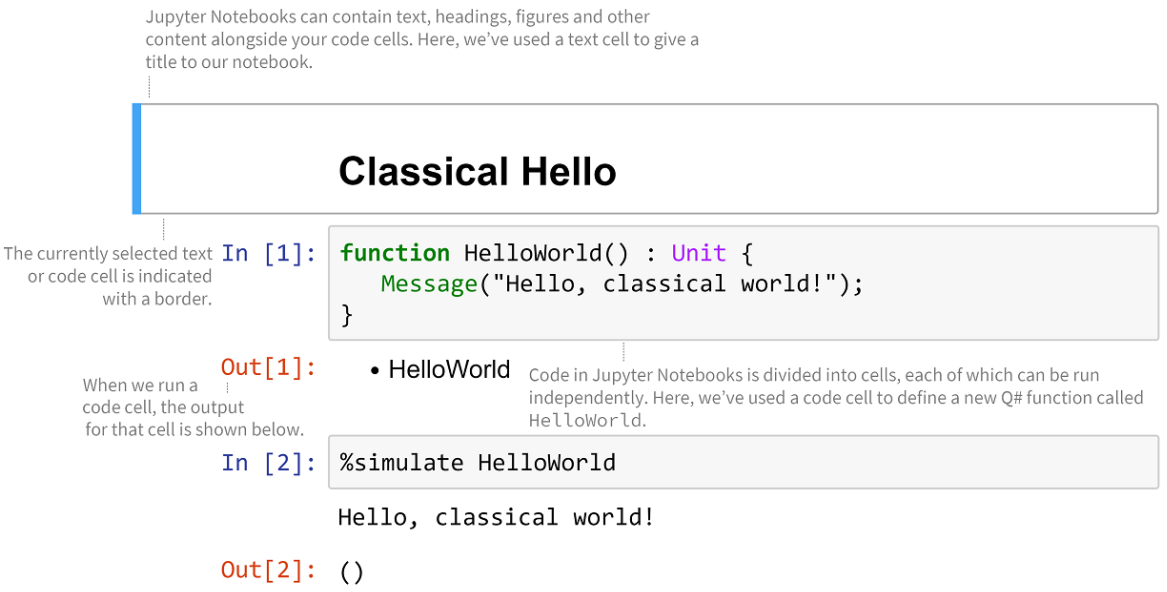
The Quantum Development Kit provides a plugin for Jupyter Notebook called IQ# that makes it easy to get started with Q# by providing host programs automatically for us. In Chapter 8, we'll see how to write host programs using Python and C#, but for now we'll focus on Q# itself. See Appendix B for instructions 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



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 {          (1)
    Message("Hello, classical world!");  (2)
}
```

(1) This line defines a new function which takes no arguments, and returns the empty tuple, whose type is written as `Unit`.

❷ 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 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 { ❶
    mutable result = Zero;           ❷
```

```
    using (qubit = Qubit()) {          ❸
        H(qubit);                      ❹
        set result = M(qubit);         ❺
        Reset(qubit);                  ❻
    }
    return result;                     ❼
}
```

❶ 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`.

❷ **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. We use the built-in value `Zero` to indicate that this variable holds measurement results, represented by the type `Result`. A value of type `Result` can either be `Zero` or `One`, corresponding to $\mathbb{Z}$-basis measurements of $\langle 0|$ and $\langle 1|$, respectively.

❸ 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`.

❹ Quantum operations such as the Hadamard operation can be found in the `Microsoft.Quantum.Primitive` namespace. For instance, we can call Hadamard using the `Microsoft.Quantum.Primitive.H` operation. After calling `H`, `qubit` is in the $\mathrm{H}|0\rangle = |+\rangle$ state.

❺ Next, we use the `M` operation to measure our qubit in the $\mathbb{Z}$ 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.

❻ Before returning our qubit to the target machine, we use the `Microsoft.Quantum.Primitive.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.

❼ We finish our operation by returning the measurement result back to 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;                                    ❶
    mutable done = false;
    repeat {                                                ❷
        set nRounds = nRounds + 1;
        set done = (NextRandomBit() == Zero);               ❸
    }
    until (done)                                            ❹
    fixup {}

    Message($"It took Lancelot {nRounds} turns to get home."); ❺
}
```

1. 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.
2. 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.
3. 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`.
4. If we got a `Zero`, then we can stop the loop.
5. 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)          ❶
) : Result {
    using (qubit = Qubit()) {
        statePreparation(qubit);                    ❷
        return result = MResetZ(qubit);             ❸
    }
}
```

❶ 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`.

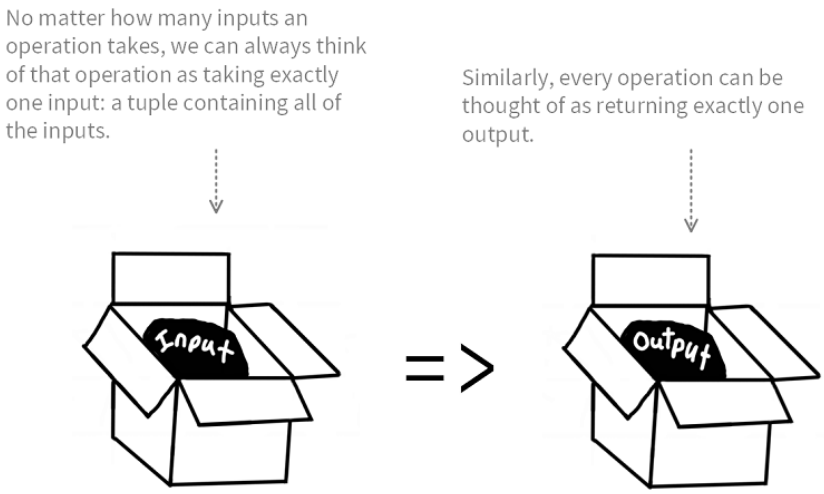
❷ Within `NextRandomBit`, the operation passed as `statePreparation` can be called in the same way as any other operation.

❸ 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

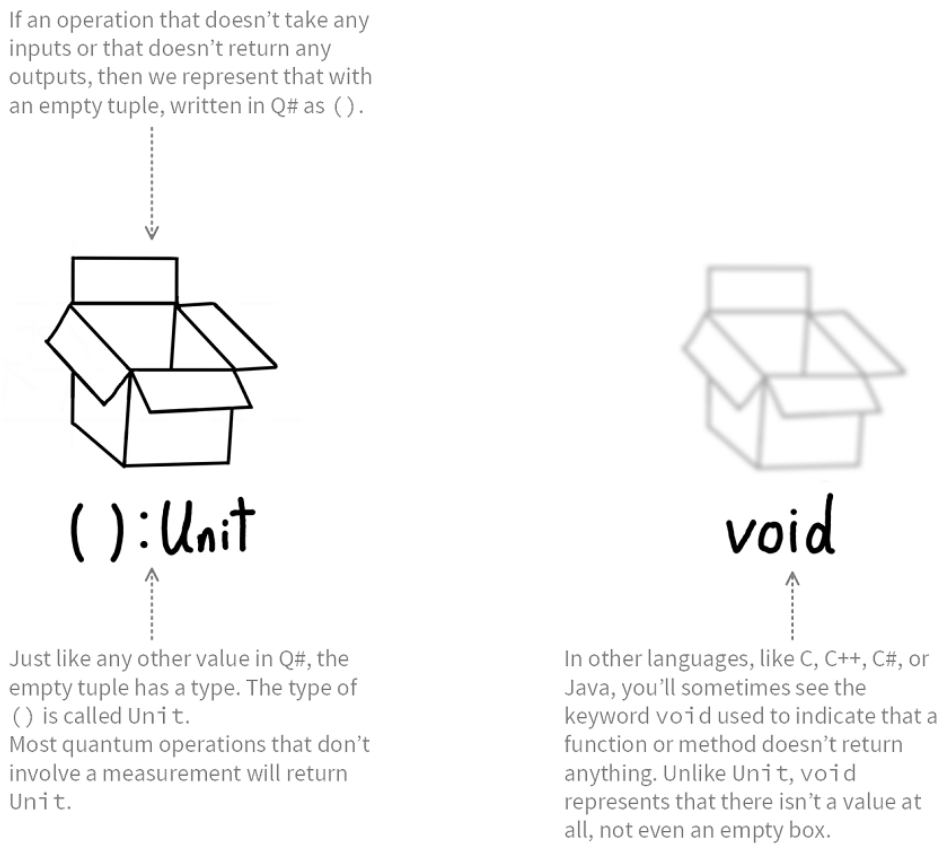


With this in mind, a function or operation that returns no outputs can be thought of as returning a tuple with no elements, `()`. The type of such tuples is called `Unit`, similar to other tuple-based languages such as F#. If we think of a tuple as a kind of box, then this is distinct from `void` as 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`



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 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` ¾ of the time and a `Zero` ¼ 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.Extensions.Math;           ❶

operation PrepareQuarterCoin(qubit : Qubit) : Unit {
    Ry(2.0 * PI() / 3.0, qubit);                  ❷
}
```

❶ Classical math functions such as `Sin`, `Cos`, `Sqrt`, and `ArcCos`, as well as constants like `PI()` are provided by the `Microsoft.Quantum.Extensions.Math` namespace, so we `open` it as well as the primitives.

❷ 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));    ❶
    Ry(rotationAngle, qubit);
}

operation PrepareMorganasCoin(qubit : Qubit) : Unit {                  ❷
    PrepareBiasedCoin(0.62, qubit);
}
```

❶ 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.

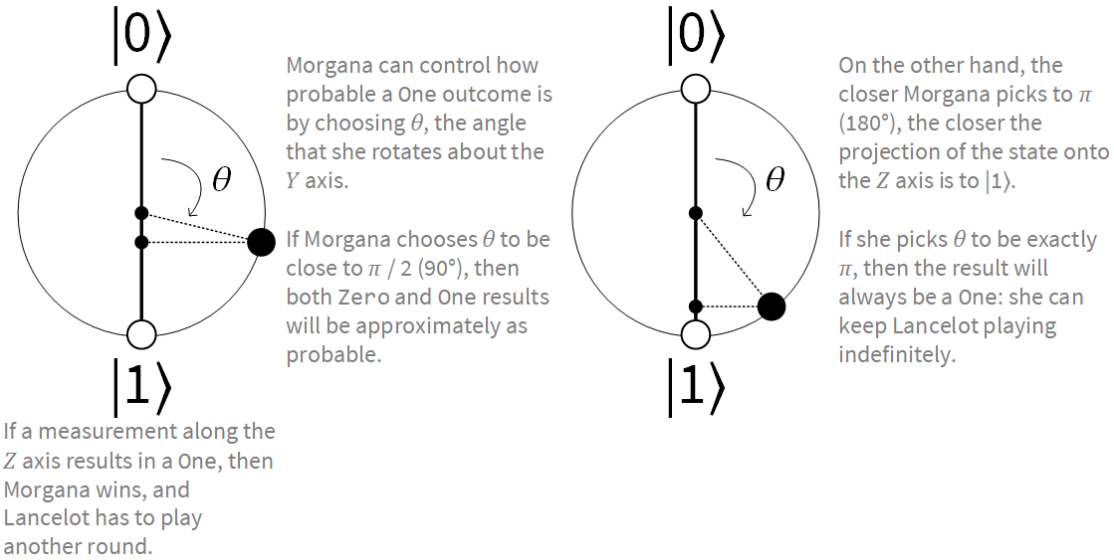
❷ This operation has the right type signature (`Qubit => Unit`) and we can see that the probability Morgana will win each round is 62%.

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 $\arccos(\cos(-\theta / 2)) = \arccos(\sqrt{62\%})$. We can cancel out the arccos and cos to find a rotation angle that gives us what we

need, -θ / 2 = arccos(√62%). Finally, we multiply both sides by -2 to get the equation we used in line ❶ of Listing 6.6. .

Figure 6.5. How Morgana can choose θ to control how her game plays out



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)
}
```

(1) 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!

(2) 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 Listing 6.9 or 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 *canon*, one of the main standard libraries that comes with the Quantum Development Kit. A full list of the operations and functions available in the canon can be found at docs.microsoft.com/qsharp/api/canon/microsoft.quantum.canon . For now, though, don't worry too much about it; we'll see more of the Q# standard libraries as we go.

By the way, the name "canon" builds on a kind of musical naming scheme that's common in functional languages. For instance, 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 whimsey you've come to expect by now.

Listing 6.9. Complete listing of Q# operations for the biased `PlayMorganasGame` example

```
open Microsoft.Quantum.Canon;                                          ❶

operation PrepareBiasedCoin(winProbability : Double, qubit : Qubit) : Unit {
    let rotationAngle = 2.0 * ArcCos(Sqrt(1.0 - winProbability));      ❷
    Ry(rotationAngle, qubit);
}

operation NextRandomBit(statePreparation : (Qubit => Unit)) : Result {
    mutable result = Zero;
    using (qubit = Qubit()) {
        statePreparation(qubit);                                       ❸
        set result = MResetZ(qubit);                                   ❹
    }
    return result;
}

operation PlayMorganasGame(winProbability : Double) : Unit {
    mutable nRounds = 0;
    mutable done = false;
    let prep = PrepareBiasedCoin(winProbability, _);                   ❺
    repeat {
        set nRounds = nRounds + 1;
        set done = (NextRandomBit(prep) == Zero);
    }
    until (done)
    fixup {}

    Message($"It took Lancelot {nRounds} turns to get home.");
}
```

❶ We also open the standard library for Q# this time, known as the canon.

❷ The rotation angle chooses the bias the coin has.

❸ Here we use the operation we passed in as `statePreparation` and apply it to the qubit.

❹ The `MResetZ` operation is defined in the canon 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.

❺ 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 Listing 6.9. "Complete listing of Q# operations for the biased PlayMorganasGameexample" 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 √p |0〉 + √(1 - p) |1〉 ,
/// 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 √p |0〉 + √(1 - p) |1〉 .
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 6.10. 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 6.11. 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.