COMP 314 Class 3: Turing machines: the simplest computers

Computing is normally done by writing certain symbols on paper.

— Alan Turing, *On computable numbers . . .* (1936)

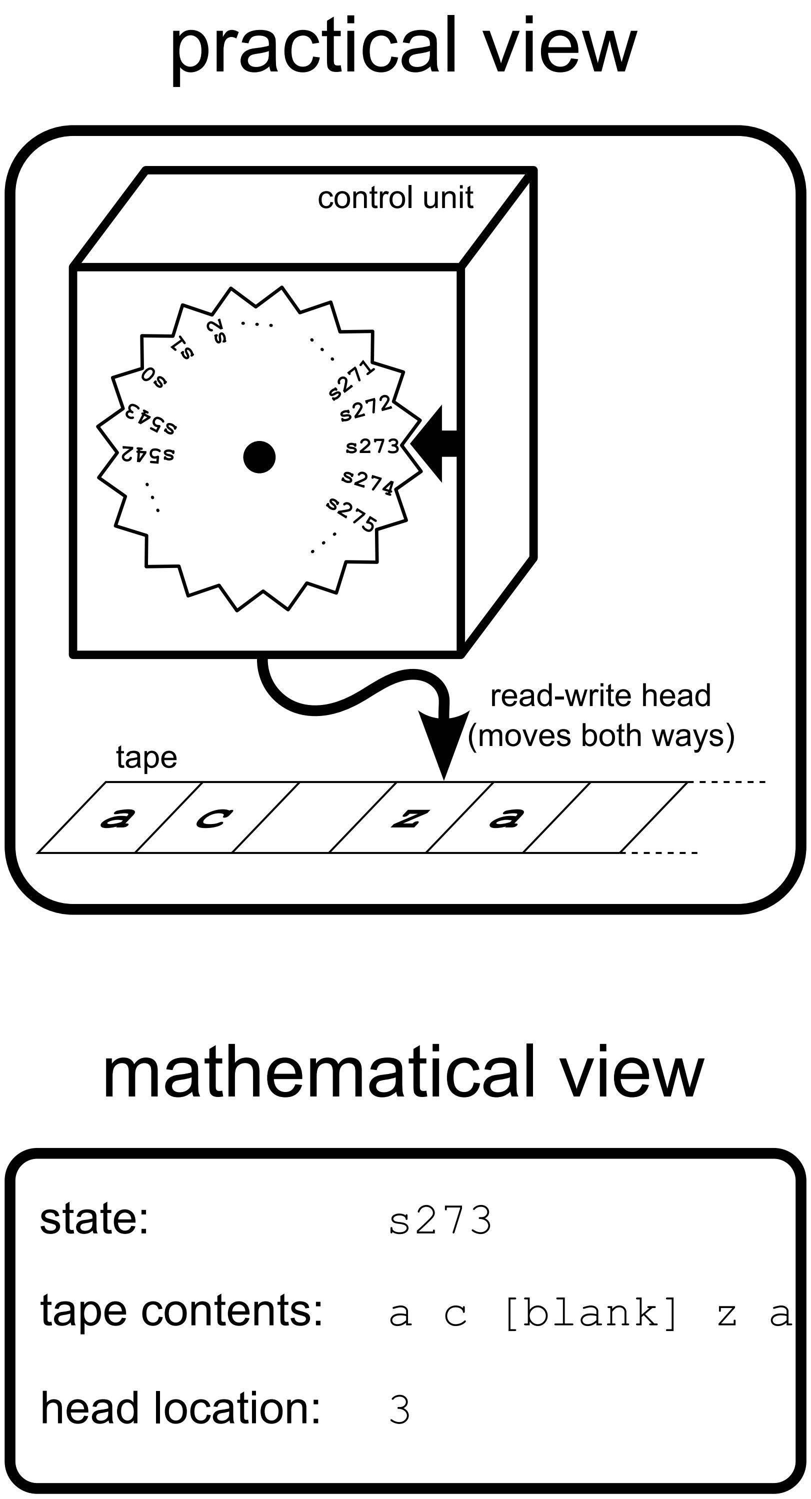
Python programs are a little problematic for studying the theory of computer science. It's hard to be absolutely certain about the meaning of a Python program. Even a very simple program, such as "`print 'yes'`", could do something unexpected. Perhaps, when invoked with certain commandline arguments, this program triggers a bug in the Python interpreter which causes a crash. This is extremely unlikely, but we can't completely rule it out. Because of this (and for some other reasons too), we need to study a much more simple, abstract and fundamental model of computation.

Our fundamental model of computation is called the *Turing machine*. Before trying to understand the definition of a Turing machine, take a look at Figure 1. Although Turing machines are completely abstract, it's helpful for us humans to maintain two distinct views of them: (i) a practical view, as a machine that could be constructed from physical materials; and (ii) a mathematical view, as a collection of mathematical objects with no physical counterparts. Figure 1 shows these two views. In the practical view, we see the three main physical pieces of the Turing machine: the control unit, the read-write head, and the tape. The tape contains cells, and each cell can store a single symbol (or be blank). The read-write head can move up and down the tape, informing the control unit of what symbols it sees and (optionally) erasing the existing symbols and writing new ones. The control unit issues instructions to the read-write head, and switches between the various states labeled on the dial at the front of the control unit. Note that this dial is for display only: we can't reach in and turn this dial ourselves—it is completely under the control of the control unit.

The mathematical view of this Turing machine is shown in the bottom panel of Figure 1: the machine is in state `s273`, the content of the tape is "`a c [blank] z a`" followed by infinitely many blanks, and the location of the head is cell 3 (assuming we index the tape cells starting with 0).

Now we're ready for a more formal, mathematical definition a Turing machine. A Turing machine is composed of five separate mathematical objects: two sets and three functions. The two sets in the definition of a Turing machine are:

- **Alphabet.** A finite set of *symbols*, which must include a special symbol called the *blank symbol*. In this book, we will almost always use the set of ASCII characters as the alphabet, with one particular character designated as the blank symbol. It just so happens that the ASCII value 7 doesn't represent a printable character (it is used to represent a beep from the computer's speaker), so let's designate this as the blank symbol. To write the blank symbol as part of a string, we will use "$\square$ ". So the content of the tape in Figure 1 is "`ac`$\square$ `za`"



state: s273

tape contents: a c [blank] z a

head location: 3

Figure 1: Two views of a Turing machine

- **State-set.** A finite set of *states*, which must include a special state called the *start state* and another special state called the *accepting state*. In this book, the states will be denoted `s0`, `s1`, `s2`, ..., and the start state will always be `s0`. The accepting state will always be one of the numbered states (e.g. `s47`), but we will mostly use the notation `sAccept` as a synonym for the accepting state.

The three functions in the definition of a Turing machine each take two inputs: the current state s and the current symbol x. Their definitions are:

- **State function** $\mathit{NewState}(s,x)$**:** output is the new state s' which the Turing machine will transition into.

- **Symbol function** $\mathit{NewSymbol}(s,x)$**:** output is the new symbol x' which the head writes in the current tape cell. (Of course, it's possible to have $x' = x$ in which case the symbol on the tape remains unchanged.)

- **Direction function** $\mathit{Direction}(s,x)$**:** output d' is either Left, Right or Stay, depending on whether the head should move left, right or stay where it is. (If the head is already at the left end of the tape, and it is commanded to move left, then it stays where it is instead.)

Sometimes, it's more convenient to think of these three functions as a single function that returns a 3-tuple (s', x', d'). This combined function is called the *transition function* of the Turing machine. Mostly, however, we will think of these functions as three separate entities and refer to them as transition function*s* instead. Note that the transition functions don't have to be defined for all possible inputs s and x. (Technically speaking, they are *partial* functions.)

You have, no doubt, already guessed how a Turing machine works. It begins a computation in its start state, with some finite sequence of symbols—the *input*—already written on the tape, and with the read-write head at position zero. The machine then applies the transition functions over and over again (typically writing some new symbols on the tape and moving the head around) until it gets into a situation where the transition functions are not defined. At this point, the machine halts, and the sequence of symbols left on the tape is defined to be the *output* of the computation.

As our first example, let's define a Turing machine called `LastTtoA` that works on genetic strings. A *genetic string* is just an ASCII string that contains only the characters `a`, `c`, `g`, and `t`. As you probably know, these letters represent the possible bases in a string of DNA. We are using genetic strings here to emphasize that, even though Turing machines are completely abstract mathematical concepts, they can be used to work with real-world data.

Anyway, let's get back to our first example of a Turing machine, `LastTtoA`. The machine will output an exact copy of the input, except that the last `t` of the input will be mutated to a `c`. (For simplicity, we will assume that the input is guaranteed to contain at least one

state s	symbol x	$NewState(s,x)$	$NewSymbol(s,x)$	$Direction(s,x)$
s0	c	s0	c	Right
	g	s0	g	Right
	a	s0	a	Right
	t	s0	t	Right
	blank	s1	blank	Left
s1	c	s1	c	Left
	g	s1	g	Left
	a	s1	a	Left
	t	s2	a	Stay
	blank	s1	blank	Left

Figure 2: Transition functions for a Turing machine that changes the last `t` of a genetic string to an `a`.

`t`.) The alphabet for this Turing machine will be the set of ASCII characters (including a blank as described above). It turns out we need three states for this machine, so the state-set is {`s0`, `s1`, `s2`}, with `s0` being the start state and `s2` being the accepting state.

The transition functions for the `LastTtoA` machine are defined in Figure 2. It should be immediately obvious that tabulating the transition functions, as in this Figure, is a terrible way to explain a Turing machine to a human. It's almost impossible to gain intuition about how the machine works by looking at a table of this kind.

Instead, Turing machines are generally described using a *state diagram* like the one shown in Figure 3. The states of the machine are represented by circles (except for the start state which has a triangle next to it, and the accepting state which is an double circle). Arrows between the states indicate transitions, and the labels on these arrows give details about the transition. Specifically, these labels consist of the current symbol followed by a colon, then the new symbol to be written (if any) followed by a comma, then the direction to move (using the obvious abbreviations `L`, `R`, and `S`). For example, the label `c:R` means "if the symbol currently under the head is a `c`, follow this arrow and move the head right." Another example from Figure 3 is `t:a,S`, which means "if the symbol currently under the head is a `t`, replace it with an `a`, follow this arrow, and leave the head where it is."

We're ready to do a computation on this Turing machine now. Suppose the input is `c t c g t a`. Then the initial configuration of the machine is `s0:`$\boxed{\texttt{c}}$ `t g a t a`. Here you can see we have adopted an obvious notation for the machine configuration: the name of the current state, followed by a colon, followed by the tape contents, with the current symbol in a box. Using this notation, the entire computation would be:

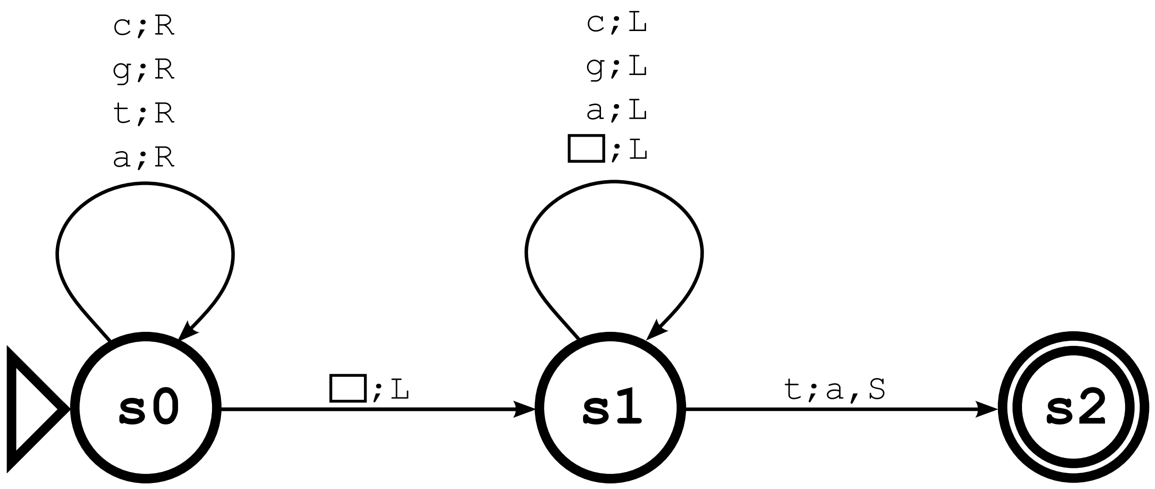


Figure 3: State diagram for the `LastTtoA` Turing machine.

s0: $\boxed{\texttt{c}}$ t c g t a

s0: c $\boxed{\texttt{t}}$ c g t a

s0: c t $\boxed{\texttt{c}}$ g t a

s0: c t c $\boxed{\texttt{g}}$ t a

s0: c t c g $\boxed{\texttt{t}}$ a

s0: c t c g t $\boxed{\texttt{a}}$

s0: c t c g t a $\boxed{\square}$

s1: c t c g t $\boxed{\texttt{a}}$

s1: c t c g $\boxed{\texttt{t}}$ a

s2: c t c g $\boxed{\texttt{a}}$ a

Note that we generally don't bother showing any blanks after the last non-blank symbol on the tape. But when the head goes into that region, as in the seventh step of the computation above, we can explicitly show the blanks up to and including the head location.

As you can see, this `LastTtoA` Turing machine has done what it promised: the output consists of the input with its last `t` converted to an `a`.

0.1 Abbreviated notation for Turing machine diagrams

We can make our Turing machine diagrams simpler by using abbreviated notation. When there are several possible read characters that produce the same action, these can all be listed separated by commas. To specify an action that applies to all read characters other than one particular one, use an exclamation point. Examples of both of these abbreviations

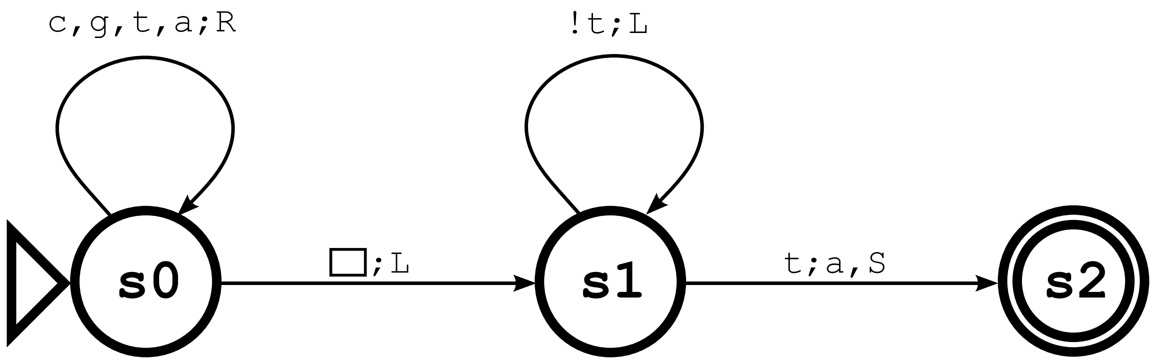


Figure 4: State diagram for the `LastTtoA` Turing machine, using abbreviated notation. This diagram is equivalent to the one in Figure 3.

are shown in Figure 4. This figure represents exactly the same information as Figure 3, assuming the input string contains only symbols drawn from the set $\{\texttt{c}, \texttt{g}, \texttt{a}, \texttt{t}, \square\}$.

We'll using one additional abbreviation: the symbol "∼" will represent the character currently under the read head. This makes it easy, for example, to specify a transition that moves right regardless of the current symbol: this would be achieved with the label "`∼;R`".

1 Creating your own Turing machines

One good way to experiment with your own Turing machines is using some software called JFLAP, developed by researchers at Duke University. You can download the software from `jflap.org`. The software is mostly self-explanatory, but here are a few points that will make it easier to use:

- JFLAP Turing machines have a *two-way infinite tape*—that is, the machine starts with a tape that extends infinitely in both directions. In contrast, our definition of a Turing machine uses a *one-way* infinite tape, which has a starting point at cell 0, and extends infinitely in only one direction. We will soon discover that the two definitions are equivalent (in terms of what the Turing machine can compute), but the distinction between one-way and two-way tapes does affect some of the details of how to program the Turing machine.
- In JFLAP, you must specify the value of the symbol to be written at each step, even if you want to leave the symbol unchanged. However, you can use the special character "∼" to achieve this. So instead of "`a,b;R`" you can use "`a,b;∼,R`".
- JFLAP uses the terminology *final state*, instead of *accepting state*. And in JFLAP, machines can have multiple final states.

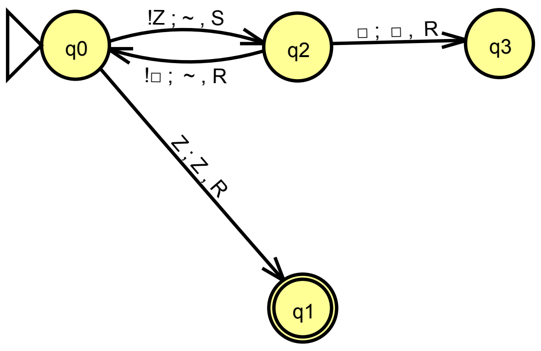


Figure 5: State diagram for the `containsZ` Turing machine.

Useful exercise: use JFLAP to implement a Turing machine that changes every "`c`" to a "`g`", and every "`g`" to a "`c`". Test your machine on multiple inputs.

2 Turing machines as decision programs

Clearly, we could regard the output of a Turing machine as its decision. So, if the machine halted with tape contents exactly equal to "`yes`", we would regard this as a Yes decision (and any other output would correspond to a No decision). However, computer scientists usually take a different approach to this definition. Instead of looking at the output of the machine, we look at which state it halted in. If it halted in the accepting state, the decision is Yes (and we also say that the Turing machine *accepts* the input); if it halted in any other state, the decision is No (and we also say that the Turing machine *rejects* the input).

For the remainder of the book, decisions by Turing machines will be defined this way (i.e. according to whether they halted in the accepting state). But decisions by Python programs will be defined, as before, according to whether or not they output "`yes`". Figure 5 gives an example of a Turing machine regarded as a decision program. This is the `containsZ` Turing machine, which accepts any input containing a "`Z`".

3 Fancier Turing machines

As we will soon see, the standard Turing machine is in fact capable of performing any computation that can be done by a modern computer. But it turns out to be very useful to also consider fancier versions of the Turing machine. As our first example of this, let's consider a Turing machine that has two tapes. From now on, the original, standard Turing machine defined above will be called the *vanilla* Turing machine, to distinguish it from its fancier variants.

3.1 Two-tape Turing machines

It's obvious that a two-tape machine is easier to program than a vanilla machine. Input will still arrive on one of the two tapes, but we can use the other tape for storing intermediate results during a calculation, and that will avoid having to shuffle the head up and down the tape so much. Note that the two tapes have to move in sync, so at any particular instant, the read-write head has access only to the same cell on both tapes. For example, suppose the read-write head is positioned at cell number 5. Then the machine can read the symbol at cell 5 on tape A, and write a symbol at cell 5 on tape B, without having to move the tape. But to write a symbol at cell 7 on tape B, both tapes would need to move up to cell 7 first. When we want to emphasize that the positions of the tapes are fixed, relative to each other, we will call this kind of multi-tape machine a *multiple-fixed-tape* machine.

So, is a two-tape machine actually *more powerful* than a vanilla machine? This depends on what we mean by "more powerful". If we are concerned with how many steps it takes to complete a computation, then it can be shown a two-tape machine is more efficient than a single-tape one. But if we are concerned with whether or not a computation can be performed at all, it turns out that the two models are equivalent. Specifically, is there any computation that can be done on a two-tape machine, but not on a vanilla machine? The answer is no: any two-tape machine can be converted into a one-tape machine that produces exactly the same output on a given input (but it may take many more steps to do so). The following claim gives a (mostly) formal proof of this fact.

Claim: Given a two-tape Turing machine T, there exists a vanilla Turing machine V that computes the same function.

Proof of the claim: Recall that our Turing machines usually use the set of ASCII characters as their alphabet—an alphabet of 256 possible symbols, such as `a`, `b`, `X`, `Y`, and so on. However, in defining a Turing machine, we are free to use any alphabet that we want. The machine V will instead use *pairs* of ASCII characters as its alphabet. This gives us a huge set of $256 \times 256 = 65,536$ possible symbols, including `aa`, `ab`, `bX`, `Xa`, `XY`, and `YY`. The new machine V will have the same states as T. The main idea is that V will perform the same operations as T, but whenever T operates on the first tape, V will operate on the first symbol in the current pair. And when T operates on the second tape, V operates on the second symbol in the current pair. (A completely formal proof would give a careful specification of V's transition functions, but these details are omitted. Hopefully, the description of V's operation is already convincing enough.) ■

3.2 Multi-tape Turing machines

A very similar proof shows that a vanilla Turing machine can simulate a k-fixed-tape machine, for any positive integer k. For example, to simulate a 5-tape machine, we use an alphabet whose symbols consist of groups of five ASCII characters (like "`xy2yz`"). So, in terms of computability, a vanilla machine is just as powerful as any multiple-fixed-tape machine.

3.3 Multiple independent tapes

The requirement to keep the tapes in sync means that programming a multi-tape machine is still rather laborious. It would be much easier if we could allow the tapes to move independently (or, equivalently, allow the machine to have multiple independent read-write heads). Once again, it turns out that this new feature improves efficiency and ease of programming, but adds nothing in terms of computability: a vanilla machine can still compute anything that a multiple-independent-tape machine can compute. To prove this, we don't need to go all the way back to the vanilla machine. We already know that vanilla machines are equivalent to multiple-fixed-tape machines—so all we need to do now is show that a multiple-fixed-tape machine can simulate a multiple-independent-tape machine.

Claim: Let I be a multiple-independent-tape Turing machine. Then there exists a multiple-fixed-tape Turing machine F that computes the same function as I.

Proof of the claim: Initially, assume I has only two tapes, A and B (the generalization to more tapes will be obvious). Machine F will have four tapes: A_1, A_2, B_1, B_2. Tapes A_1 and B_1 in F perform exactly the same roles as tapes A and B in I. Tapes A_2 and B_2 are used to keep track of where the head of I is on each of its tapes. Specifically, A_2 and B_2 are always completely blank except for a single "`x`" symbol in the cell corresponding to I's head position. For example, suppose that at some point in I's computation, the read-write head is positioned at cell 9 for tape A and cell 23 for tape B. Then at the corresponding point in F's simulation of this computation, tape A_2 is blank except for an "`x`" at cell 9, and tape B_2 is blank except for an "`x`" at cell 23. Of course, the *real* head on machine F has to find the relevant "`x`" before every single simulated read or write, which could be time-consuming. In the worst case, it involves moving the head all the way to the left end of the tape, then scanning right until the "`x`" is found. Nevertheless, this can certainly be done, so the claim is proved. ■

Now that we know all these models have equivalent computational power, the distinction between fixed and independent tapes becomes mostly irrelevant. But for concreteness, let's agree that from now on the term *multi-tape* refers to multiple independent tapes.

3.4 Two-way infinite tapes

Since the JFLAP software uses Turing machines with two-way infinite tapes, it's worth briefly mentioning how these can be simulated using our own one-way variant. This relies on a simple trick: you use a 2-tape machine, where tape A stores the content of the left-hand half of the two-way infinite tape, and tape B stores the content of the right-hand half. Completing a formal proof that this simulation works correctly requires several details to be fleshed out, but it's not hard to do.

3.5 From vanilla Turing machine to Python program

To summarize what we know so far: vanilla Turing machines can simulate multiple-fixed-tape machines, which can simulate multiple-independent-tape machines. These two simulation proofs are just the first baby steps in a chain of proofs that can be used to show how a vanilla Turing machine can simulate any real computer or programming language. To examine the entire chain in detail would take us too long (and, quite frankly, be too tedious). But it's extremely important to be aware of the basic outline of this chain of simulations which takes us from vanilla Turing machines to the Python programs we know and love. Here is a brief sketch of how you could do it:

- **Random-access Turing machine.** A *random-access Turing machine* has, in addition to the usual tape(s) and head, a register that stores an arbitrary integer. The register can be used as an address to read or write symbols in a given cell on the tape. For example, if the register currently contains the value 588, the machine can read or write tape cell 588 in a single operation, regardless of the current position of the head. The machine also has special operations for transferring integer values between the register and the tape(s). We can simulate a random-access Turing machine with a standard multi-tape machine by using a separate tape for the register. Each random access is simulated by running a small sub-program that counts the relevant number of cells from the left end of the tape.

- **Scute: simple computer with RAM, registers, and a basic instruction set.** From the random-access Turing machine, we can leap straight to a simulation of a simple idealized computer. Let's call our simple computer a *Scute*—the name comes from selecting some of the letters from the phrase "*s*imple *c*omp*ute*r". A scute has several general-purpose registers that can store a single ASCII character, and a special address register that can store an arbitrary integer. (Yes, the address register has an infinite amount of storage available, since the integer it stores could be arbitrarily long. Remember, this is an idealized computer.) Scutes have an infinite amount of RAM, numbered sequentially, with each cell capable of storing a single ASCII character. A given scute has a fixed program (*not* stored in the RAM—it's best to think of the program being in a ROM that is executed at boot time). The program is written in a simple assembly language called *scutel*. ("Scutel" is a contraction of "scute language", and is pronounced *scoot-ill*). Scutel programs are written with a basic instruction set containing familiar instructions such as LOAD, STORE, ADD, SUBTRACT, MULTIPLY, DIVIDE, AND, OR, NOT, BRANCH, BRANCH_IF_ZERO, BRANCH_IF_NEGATIVE. The precise definitions of these instructions are unimportant. But in case you're curious, the model I have in mind is that: LOAD and STORE transfer a single ASCII character between RAM and one of the general-purpose registers (with the RAM address specified by the address register); the arithmetic operations like ADD operate on signed 8-bit values (i.e. ASCII values regarded as signed integers) stored in the general-purpose registers; logical operations like AND are bitwise; the conditional branch operations are based on the value stored in a specified register.

It's relatively easy to sketch the simulation of a scute S by a random-access Turing machine R. R's single register mimics S's address register. S's general-purpose registers are each simulated by a separate tape in R (this is unnecessarily wasteful, but that's irrelevant). S's RAM is simulated by another of R's tapes. The ROM-like scutel program is built into R's transition function. R has a state for each line in the scutel program, and various other states that help with implementing individual instructions. It's not hard to check each scutel instruction in turn, verifying that each can be translated into a short sequence of transitions in R's ccontrol unit.

- **A real, modern computer.** The CPUs of modern computers often have dozens of registers and a vast array of instructions, but it's obvious that these features can be simulated by a scute. In fact, undergraduate computer organization and architecture classes are sometimes taught with this underlying theme: by starting with a few simple components (like AND and OR gates), we can build successively more complex components until we have a complete modern computer and operating system. My favorite demonstration of this is the book *From NAND to Tetris*, by Noam Nisan and Shimon Schocken. Needless to say, we won't be deviating into a computer architecture class here. We have too much theoretical computer science to study instead.

- **Python programs.** Once we can simulate a modern computer, we can simulate any software too, and this includes Python programs. One way to see this is to appeal to the well-known equivalence between hardware and software. But an alternative is to imagine configuring your computer so that it automatically runs some particular Python program on startup. Once it is set up in this way, your computer can be regarded as a single, fixed piece of computer hardware that simulates the Python program. In particular, this means we can use Turing machines to simulate the Python programs we have been using as the model of computation in earlier chapters of the book. This point is probably obvious already, but I would like to make it absolutely clear

So, although he have omitted all the formal details, the above chain of reasoning shows that a vanilla Turing machine can simulate any Python program. Hence, any computation that can be performed by a Python program, can also be performed by a Turing machine.

3.6 Going back the other way: simulating a Turing machine with Python

In this book, Python programs are the main model of computation. But most theoretical computer science is done in terms of Turing machines. Therefore, it's important for us to realize that Python programs and Turing machines are precisely equivalent, in terms of what computations they are capable of performing. In the previous section, we saw that Turing machines can simulate Python programs. But what about the other direction? Can Python simulate a Turing machine? If so, this would complete the proof that the two models are equivalent.

Fortunately, it's an easy exercise to implement a Turing machine in Python. We would need to agree on a notation for Turing machines, but once that is done, it's trivial to read in the description of a Turing machine and its input, and then simulate the execution of the machine. I'll leave it to you to try this on your own if you're interested.

There is one wrinkle in the simulation. Turing machines have an infinite tape, whereas real computers have a finite amount of memory. So it's tempting to think that we cannot, in fact, simulate every Turing machine using Python. This turns out to be false, and a simple example will demonstrate why. Consider a Turing machine that prints the infinite sequence "`1↵2↵3...`" on its tape. Clearly, we can simulate this with the Python program "`x=1↵while True:  print x; x=x+1`".

We are taking advantage of one of the nice features of Python here: Python uses unbounded integers by default, so unlike an `int` in Java or C++, the value of `x` will never overflow. But on the real computer, the length of `x` would eventually exceed the size of the computer's memory. So we need another caveat: Python programs can simulate Turing machines, provided we are willing to add sufficient memory to our computer. In principle, we can add as much memory as we need (perhaps by using some kind of virtual memory backed by cloud storage, for example). Alternatively, we can just imagine a Python program running on an idealized computer that does have infinite memory. All the results in this book will be valid for this kind of idealized computer.