Chapter 6

Cellular Automata

A **cellular automaton** is a model of a world with very simple physics. "Cellular" means that the space is divided into discrete chunks, called cells. An "automaton" is a machine that performs computations—it could be a real machine, but more often the "machine" is a mathematical abstraction or a computer simulation.

Automata are governed by rules that determine how the system evolves in time. Time is divided into discrete steps, and the rules specify how to compute the state of the world during the next time step based on the current state.

As a trivial example, consider a cellular automaton (CA) with a single cell. The state of the cell is an integer represented with the variable x_i, where the subscript i indicates that x_i is the state of the system during time step i. As an initial condition, $x_0 = 0$.

Now all we need is a rule. Arbitrarily, I'll pick $x_i = x_{i-1} + 1$, which says that after each time step, the state of the CA gets incremented by 1. So far, we have a simple CA that performs a simple calculation: it counts.

But this CA is atypical; normally the number of possible states is finite. To bring it into line, I'll choose the smallest interesting number of states, 2, and another simple rule, $x_i = (x_{i-1} + 1)\%2$, where % is the remainder (or modulus) operator.

This CA performs a simple calculation: it blinks. That is, the state of the cell switches between 0 and 1 after every time step.

Most CAs are **deterministic**, which means that rules do not have any random elements; given the same initial state, they always produce the same result. There are also nondeterministic CAs, but I will not address them here.

6.1 Stephen Wolfram

The CA in the previous section was 0-dimensional and it wasn't very interesting. But 1-dimensional CAs turn out to be surprisingly interesting.

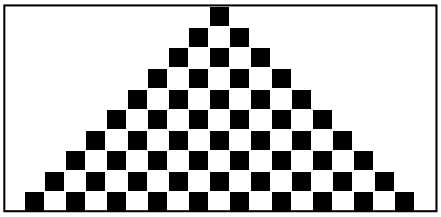

Figure 6.1: Rule 50 after 10 time steps.

In the early 1980s Stephen Wolfram published a series of papers presenting a systematic study of 1-dimensional CAs. He identified four general categories of behavior, each more interesting than the last.

To say that a CA has dimensions is to say that the cells are arranged in a contiguous space so that some of them are considered "neighbors." In one dimension, there are three natural configurations:

Finite sequence: A finite number of cells arranged in a row. All cells except the first and last have two neighbors.

Ring: A finite number of cells arranged in a ring. All cells have two neighbors.

Infinite sequence: An infinite number of cells arranged in a row.

The rules that determine how the system evolves in time are based on the notion of a "neighborhood," which is the set of cells that determines the next state of a given cell. Wolfram's experiments use a 3-cell neighborhood: the cell itself and its left and right neighbors.

In these experiments, the cells have two states, denoted 0 and 1, so the rules can be summarized by a table that maps from the state of the neighborhood (a tuple of 3 states) to the next state for the center cell. The following table shows an example:

prev	111	110	101	100	011	010	001	000
next	0	0	1	1	0	0	1	0

The row first shows the eight states a neighborhood can be in. The second row shows the state of the center cell during the next time step. As a concise encoding of this table, Wolfram suggested reading the bottom row as a binary number. Because 00110010 in binary is 50 in decimal, Wolfram calls this CA "Rule 50."

Figure 6.1 shows the effect of Rule 50 over 10 time steps. The first row shows the state of the system during the first time step; it starts with one cell "on" and the rest "off". The second row shows the state of the system during the next time step, and so on.

The triangular shape in the figure is typical of these CAs; is it a consequence of the shape of the neighborhood. In one time step, each cell influences the state of one neighbor in either direction. During the next time step, that influence can propagate one more cell in each direction. So each cell in the past has a "triangle of influence" that includes all of the cells that can be affected by it.

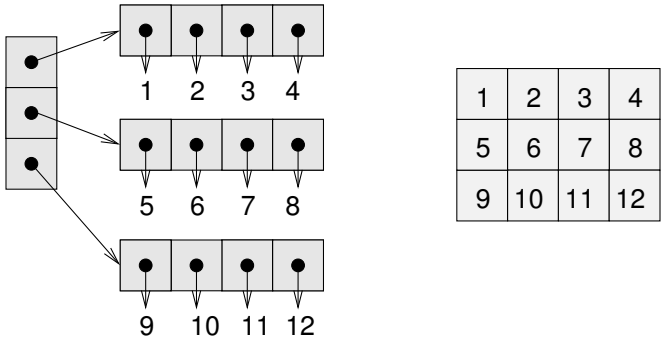

Figure 6.2: A list of lists (left) and a Numpy array (right).

6.2 Implementing CAs

To generate the previous figure, I wrote a Python program that implements and draws CAs. You can download my code from `thinkcomplex.com/CA.py`. and `thinkcomplex.com/CADrawer.py`.

To store the state of the CA, I use a NumPy array. An array is a multi-dimensional data structure whose elements are all the same type. It is similar to a nested list, but usually smaller and faster. Figure 6.2 shows why. The diagram on the left shows a list of lists of integers; each dot represents a reference, which takes up 4–8 bytes. To access one of the integers, you have to follow two references.

The diagram on the right shows an array of the same integers. Because the elements are all the same size, they can be stored contiguously in memory. This arrangement saves space because it doesn't use references, and it saves time because the location of an element can be computed directly from the indices; there is no need to follow a series of references.

Here is a CA object that uses a NumPy array:

```
import numpy

class CA(object):

    def __init__(self, rule, n=100, ratio=2):
        self.table = make_table(rule)
        self.n = n
        self.m = ratio*n + 1
        self.array = numpy.zeros((n, self.m), dtype=numpy.int8)
        self.next = 0
```

`rule` is an integer in the range 0-255, which represents the CA rule table using Wolfram's encoding. `make_table` converts the rule to a dictionary that maps from neighborhood states to cell states. For example, in Rule 50 the table maps from $(1, 1, 1)$ to 0.

`n` is the number of rows in the array, which is the number of time steps we will compute. `m` is the number of columns, which is the number of cells. To get started, I'll implement a finite array of cells.

`zeros` is provided by NumPy; it creates a new array with the given dimensions, `n` by `m`; `dtype` stands for "data type," and it specifies the type of the array elements. `int8` is an 8-bit integer, so we are limited to 256 states, but that's no problem: we only need two.

`next` is the index of the next time step.

There are two common starting conditions for CAs: a single cell, or a row of random cells. `start_single` initializes the first row with a single cell and increments `next`:

```
    def start_single(self):
        """Starts with one cell in the middle of the top row."""
        self.array[0, self.m/2] = 1
        self.next += 1
```

The array index is a tuple that species the row and column of the cell, in that order.

`step` computes the next state of the CA:

```
    def step(self):
        i = self.next
        self.next += 1

        a = self.array
        t = self.table
        for j in xrange(1,self.m-1):
            a[i,j] = t[tuple(a[i-1, j-1:j+2])]
```

`i` is the time step and the index of the row we are about to compute. `j` loops through the cells, skipping the first and last, which are always off.

Arrays support slice operations, so `self.array[i-1, j-1:j+2]` gets three elements from row `i-1`. Then we look up the neighborhood tuple in the table, get the next state, and store it in the array.

Array indexing is constant time, so `step` is linear in n. Filling in the whole array is $O(nm)$.

You can read more about NumPy and arrays at `scipy.org/Tentative_NumPy_Tutorial`.

6.3 CADrawer

An **abstract class** is a class definition that specifies the interface for a set of methods without providing an implementation. Child classes extend the abstract class and implement the incomplete methods. See `http://en.wikipedia.org/wiki/Abstract_type`.

As an example, `CADrawer` defines an interface for drawing CAs; here is the definition:

```
class Drawer(object):
    """Drawer is an abstract class that should not be instantiated.
    It defines the interface for a CA drawer; child classes of Drawer
    should implement draw, show and save.

    If draw_array is not overridden, the child class should provide
    draw_cell.
```

```
    """
    def __init__(self):
        msg = 'CADrawer is an abstract type and should not be instantiated.'
        raise UnimplementedMethodException, msg

    def draw(self, ca):
        """Draws a representation of cellular automaton (CA).
        This function generally has no visible effect."""
        raise UnimplementedMethodException

    def draw_array(self, a):
        """Iterate through array (a) and draws any non-zero cells."""
        for i in xrange(self.rows):
            for j in xrange(self.cols):
                if a[i,j]:
                    self.draw_cell(j, self.rows-i-1)

    def draw_cell(self, ca):
        """Draws a single cell.
        Not required for all implementations."""
        raise UnimplementedMethodException

    def show(self):
        """Displays the representation on the screen, if possible."""
        raise UnimplementedMethodException

    def save(self, filename):
        """Saves the representation of the CA in filename."""
        raise UnimplementedMethodException
```

Abstract classes should not be instantiated; if you try, you get an `UnimplementedMethodException`, which is a simple extension of `Exception`:

```
class UnimplementedMethodException(Exception):
    """Used to indicate that a child class has not implemented an
    abstract method."""
```

To instantiate a `CADrawer` you have to define a child class that implements the methods, then instantiate the child.

`CADrawer.py` provides three implementations, one that uses `pyplot`, one that uses the Python Imaging Library (PIL), and one that generates Encapsulated Postscript (EPS).

Here is an example that uses `PyplotDrawer` to display a CA on the screen:

```
    ca = CA(rule, n)
    ca.start_single()
    ca.loop(n-1)

    drawer = CADrawer.PyplotDrawer()
    drawer.draw(ca)
    drawer.show()
```

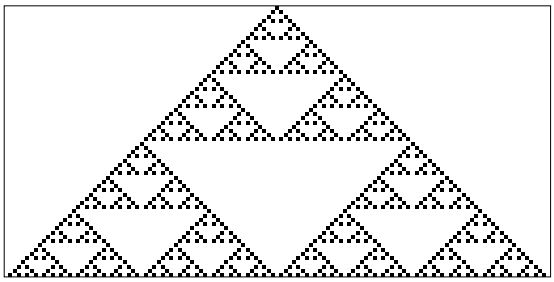

Figure 6.3: Rule 18 after 64 steps.

Exercise 6.1. *Download* `thinkcomplex.com/CA.py` *and* `thinkcomplex.com/CADrawer.py` *and confirm that they run on your system; you might have to install additional Python packages.*

Create a new class called `CircularCA` *that extends* `CA` *so that the cells are arranged in a ring. Hint: you might find it useful to add a column of "ghost cells" to the array.*

You can download my solution from `thinkcomplex.com/CircularCA.py`

6.4 Classifying CAs

Wolfram proposes that the behavior of CAs can be grouped into four classes. Class 1 contains the simplest (and least interesting) CAs, the ones that evolve from almost any starting condition to the same uniform pattern. As a trivial example, Rule 0 always generates an empty pattern after one time step.

Rule 50 is an example of Class 2. It generates a simple pattern with nested structure; that is, the pattern contains many smaller versions of itself. Rule 18 makes the nested structure even clearer; Figure 6.3 shows what it looks like after 64 steps.

This pattern resembles the Sierpiński triangle, which you can read about at `http://en.wikipedia.org/wiki/Sierpinski_triangle`.

Some Class 2 CAs generate patterns that are intricate and pretty, but compared to Classes 3 and 4, they are relatively simple.

6.5 Randomness

Class 3 contains CAs that generate randomness. Rule 30 is an example; Figure 6.4 shows what it looks like after 100 time steps.

Along the left side there is an apparent pattern, and on the right side there are triangles in various sizes, but the center seems quite random. In fact, if you take the center column and treat it as a sequence of bits, it is hard to distinguish from a truly random sequence. It passes many of the statistical tests people use to test whether a sequence of bits is random.

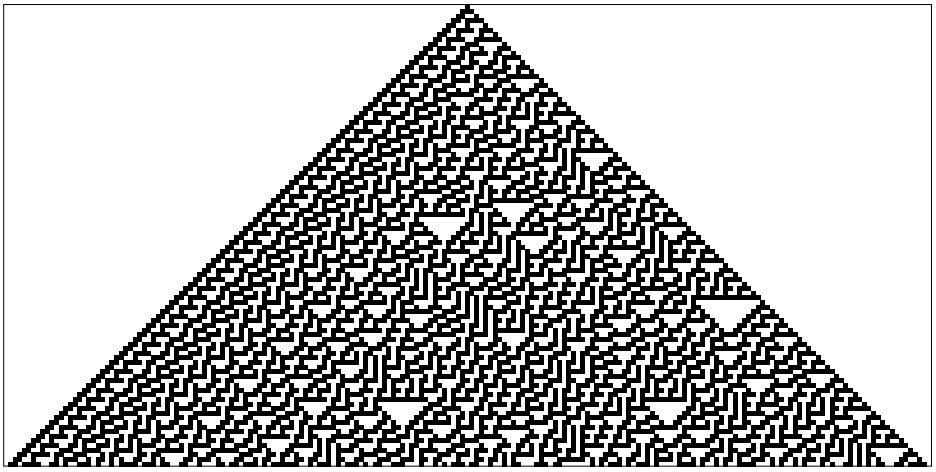

Figure 6.4: Rule 30 after 100 time steps.

Programs that produce random-seeming numbers are called **pseudo-random number generators** (PRNGs). They are not considered truly random because

- Many of them produce sequences with regularities that can be detected statistically. For example, the original implementation of `rand` in the C library used a linear congruential generator that yielded sequences with easily detectable serial correlations.

- Any PRNG that uses a finite amount of state (that is, storage) will eventually repeat itself. One of the characteristics of a generator is the **period** of this repetition.

- The underlying process is fundamentally deterministic, unlike some physical processes, like radioactive decay and thermal noise, that are considered to be fundamentally random.

Modern PRNGs produce sequences that are statistically indistinguishable from random, and they can be implemented with with periods so long that the universe will collapse before they repeat. The existence of these generators raises the question of whether there is any real difference between a good quality pseudo-random sequence and a sequence generated by a "truly" random process. In *A New Kind of Science*, Wolfram argues that there is not (pages 315–326).

Exercise 6.2. *This exercise asks you to implement and test several PRNGs.*

1. *Write a program that implements one of the linear congruential generators described at* `http://en.wikipedia.org/wiki/Linear_congruential_generator`*).*

2. *Download* `DieHarder`*, a random number test suite, from* `http://phy.duke.edu/~rgb/General/rand_rate.php` *and use it to test your PRNG. How does it do?*

3. *Read the documentation of Python's* `random` *module. What PRNG does it use? Test it using DieHarder.*

4. *Implement a Rule 30 CA on a ring with a few hundred cells, run it for as many time steps as you can in a reasonable amount of time, and output the center column as a sequence of bits. Test it using DieHarder.*

6.6 Determinism

The existence of Class 3 CAs is surprising. To understand how surprising, it is useful to consider philosophical **determinism** (see `http://en.wikipedia.org/wiki/Determinism`). Most philosophical stances are hard to define precisely because they come in a variety of flavors. I often find it useful to define them with a list of statements ordered from weak to strong:

D1: Deterministic models can make accurate predictions for some physical systems.

D2: Many physical systems can be modeled by deterministic processes, but some are intrinsically random.

D3: All events are caused by prior events, but many physical systems are nevertheless fundamentally unpredictable.

D4: All events are caused by prior events, and can (at least in principle) be predicted.

My goal in constructing this range is to make D1 so weak that virtually everyone would accept it, D4 so strong that almost no one would accept it, with intermediate statements that some people accept.

The center of mass of world opinion swings along this range in response to historical developments and scientific discoveries. Prior to the scientific revolution, many people regarded the working of the universe as fundamentally unpredictable or controlled by supernatural forces. After the triumphs of Newtonian mechanics, some optimists came to believe something like D4; for example, in 1814 Pierre-Simon Laplace wrote

> We may regard the present state of the universe as the effect of its past and the cause of its future. An intellect which at a certain moment would know all forces that set nature in motion, and all positions of all items of which nature is composed, if this intellect were also vast enough to submit these data to analysis, it would embrace in a single formula the movements of the greatest bodies of the universe and those of the tiniest atom; for such an intellect nothing would be uncertain and the future just like the past would be present before its eyes.

This "intellect" came to be called "Laplace's Demon". See `http://en.wikipedia.org/wiki/Laplace's_demon`. The word "demon" in this context has the sense of "spirit," with no implication of evil.

Discoveries in the 19th and 20th centuries gradually dismantled this hope. The thermodynamic concept of entropy, radioactive decay, and quantum mechanics posed successive challenges to strong forms of determinism.

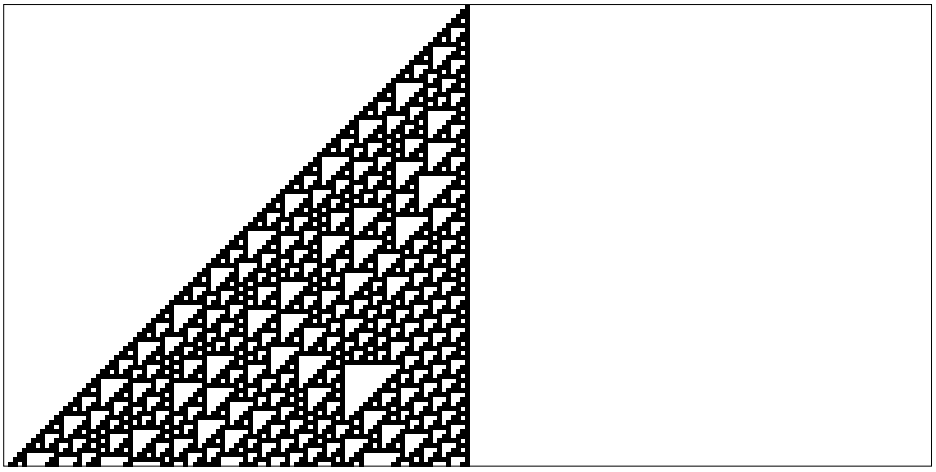

Figure 6.5: Rule 110 after 100 time steps.

In the 1960s chaos theory showed that in some deterministic systems prediction is only possible over short time scales, limited by the precision of measurement of initial conditions.

Most of these systems are continuous in space (if not time) and nonlinear, so the complexity of their behavior is not entirely surprising. Wolfram's demonstration of complex behavior in simple cellular automata is more surprising—and disturbing, at least to a deterministic world view.

So far I have focused on scientific challenges to determinism, but the longest-standing objection is the conflict between determinism and human free will. Complexity science provides a possible resolution of this apparent conflict; we come back to this topic in Section 10.7.

6.7 Structures

The behavior of Class 4 CAs is even more surprising. Several 1-D CAs, most notably Rule 110, are **Turing complete**, which means that they can compute any computable function. This property, also called **universality**, was proved by Matthew Cook in 1998. See `http://en.wikipedia.org/wiki/Rule_110`.

Figure 6.5 shows what Rule 110 looks like with an initial condition of a single cell and 100 time steps. At this time scale it is not apparent that anything special is going on. There are some regular patterns but also some features that are hard to characterize.

Figure 6.6 shows a bigger picture, starting with a random initial condition and 600 time steps:

After about 100 steps the background settles into a simple repeating pattern, but there are a number of persistent structures that appear as disturbances in the background. Some

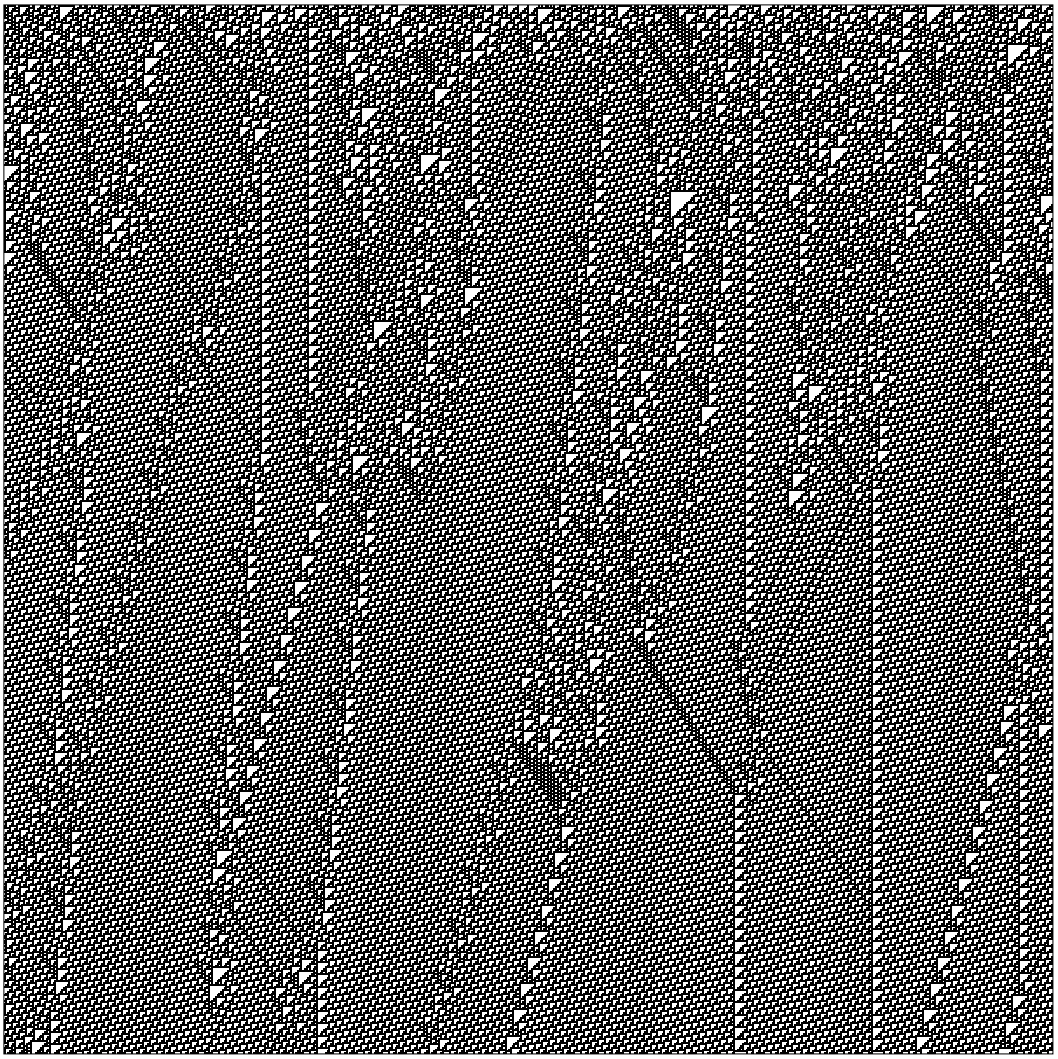

Figure 6.6: Rule 110 with random initial conditions and 600 time steps.

of these structures are stable, so they appear as vertical lines. Others translate in space, appearing as diagonals with different slopes, depending on how many time steps they take to shift by one column. These structures are called **spaceships**.

Collisions between spaceships yield different results depending on the types of the spaceships and the phase they are in when they collide. Some collisions annihilate both ships; others leave one ship unchanged; still others yield one or more ships of different types.

These collisions are the basis of computation in a Rule 110 CA. If you think of spaceships as signals that propagate on wires, and collisions as gates that compute logical operations like AND and OR, you can see what it means for a CA to perform a computation.

Exercise 6.3. *This exercise asks you to experiment with Rule 110 and see how many spaceships you can find.*

1. *Modify your program from the previous exercises so it starts with an initial condition that yields the stable background pattern.*
2. *Modify the initial condition by adding different patterns in the center of the row and see which ones yield spaceships. You might want to enumerate all possible patterns of n bits, for some reasonable value of n. For each spaceship, can you find the period and rate of translation? What is the biggest spaceship you can find?*

6.8 Universality

To understand universality, we have to understand computability theory, which is about models of computation and what they compute.

One of the most general models of computation is the Turing machine, which is an abstract computer proposed by Alan Turing in 1936. A Turing machine is a 1-D CA, infinite in both directions, augmented with a read-write head. At any time, the head is positioned over a single cell. It can read the state of that cell (usually there are only two states) and it can write a new value into the cell.

In addition, the machine has a register, which records the state of the machine (one of a finite number of states), and a table of rules. For each machine state and cell state, the table specifies an action. Actions include modifying the cell the head is over and moving one cell to the left or right.

A Turing machine is not a practical design for a computer, but it models common computer architectures. For a given program running on a real computer, it is possible (at least in principle) to construct a Turing machine that performs an equivalent computation.

The Turing machine is useful because it is possible to characterize the set of functions that can be computed by a Turing machine, which is what Turing did. Functions in this set are called Turing computable.

To say that a Turing machine can compute any Turing-computable function is a **tautology**: it is true by definition. But Turing-computability is more interesting than that.

It turns out that just about every reasonable model of computation anyone has come up with is Turing complete; that is, it can compute exactly the same set of functions as the

Turing machine. Some of these models, like lamdba calculus, are very different from a Turing machine, so their equivalence is surprising.

This observation led to the Church-Turing Thesis, which is essentially a definition of what it means to be computable. The "thesis" is that Turing-computability is the right, or at least natural, definition of computability, because it describes the power of such a diverse collection of models of computation.

The Rule 110 CA is yet another model of computation, and remarkable for its simplicity. That it, too, turns out to be universal lends support to the Church-Turing Thesis.

In *A New Kind of Science*, Wolfram states a variation of this thesis, which he calls the "principle of computational equivalence:"

> Almost all processes that are not obviously simple can be viewed as computations of equivalent sophistication.
>
> More specifically, the principle of computational equivalence says that systems found in the natural world can perform computations up to a maximal ("universal") level of computational power, and that most systems do in fact attain this maximal level of computational power. Consequently, most systems are computationally equivalent (see `mathworld.wolfram.com/PrincipleofComputationalEquivalence.html`).

Applying these definitions to CAs, Classes 1 and 2 are "obviously simple." It may be less obvious that Class 3 is simple, but in a way perfect randomness is as simple as perfect order; complexity happens in between. So Wolfram's claim is that Class 4 behavior is common in the natural world, and that almost all of the systems that manifest it are computationally equivalent.

Exercise 6.4. *The goal of this exercise is to implement a Turing machine. See* `http://en.wikipedia.org/wiki/Turing_machine`.

1. *Start with a copy of* `CA.py` *named* `TM.py`. *Add attributes to represent the location of the head, the action table and the state register.*
2. *Override* `step` *to implement a Turing machine update.*
3. *For the action table, use the rules for a 3-state busy beaver.*
4. *Write a class named* `TMDrawer` *that generates an image that represents the state of the tape and the position and state of the head. For one example of what that might look like, see* `http://mathworld.wolfram.com/TuringMachine.html`.

6.9 Falsifiability

Wolfram holds that his principle is a stronger claim than the Church-Turing Thesis because it is about the natural world rather than abstract models of computation. But saying that natural processes "can be viewed as computations" strikes me as a statement about theory choice more than a hypothesis about the natural world.

Also, with qualifications like "almost" and undefined terms like "obviously simple," his hypothesis may be **unfalsifiable**. Falsifiability is an idea from the philosophy of science, proposed by Karl Popper as a demarcation between scientific hypotheses and pseudoscience. A hypothesis is falsifiable if there is an experiment, at least in the realm of practicality, that would contradict the hypothesis if it were false.

For example, the claim that all life on earth is descended from a common ancestor is falsifiable because it makes specific predictions about similarities in the genetics of modern species (among other things). If we discovered a new species whose DNA was almost entirely different from ours, that would contradict (or at least bring into question) the theory of universal common descent.

On the other hand, "special creation," the claim that all species were created in their current form by a supernatural agent, is unfalsifiable because there is nothing that we could observe about the natural world that would contradict it. Any outcome of any experiment could be attributed to the will of the creator.

Unfalsifiable hypotheses can be appealing because they are impossible to refute. If your goal is never to be proved wrong, you should choose hypotheses that are as unfalsifiable as possible.

But if your goal is to make reliable predictions about the world—and this is at least one of the goals of science—then unfalsifiable hypotheses are useless. The problem is that they have no consequences (if they had consequences, they would be falsifiable).

For example, if the theory of special creation were true, what good would it do me to know it? It wouldn't tell me anything about the creator except that he has an "inordinate fondness for beetles" (attributed to J. B. S. Haldane). And unlike the theory of common descent, which informs many areas of science and bioengineering, it would be of no use for understanding the world or acting in it.

Exercise 6.5. *Falsifiability is an appealing and useful idea, but among philosophers of science it is not generally accepted as a solution to the demarcation problem, as Popper claimed.*

Read `http://en.wikipedia.org/wiki/Falsifiability` *and answer the following questions.*

1. *What is the demarcation problem?*

2. *How, according to Popper, does falsifiability solve the demarcation problem?*

3. *Give an example of two theories, one considered scientific and one considered unscientific, that are successfully distinguished by the criterion of falsifiability.*

4. *Can you summarize one or more of the objections that philosophers and historians of science have raised to Popper's claim?*

5. *Do you get the sense that practicing philosophers think highly of Popper's work?*

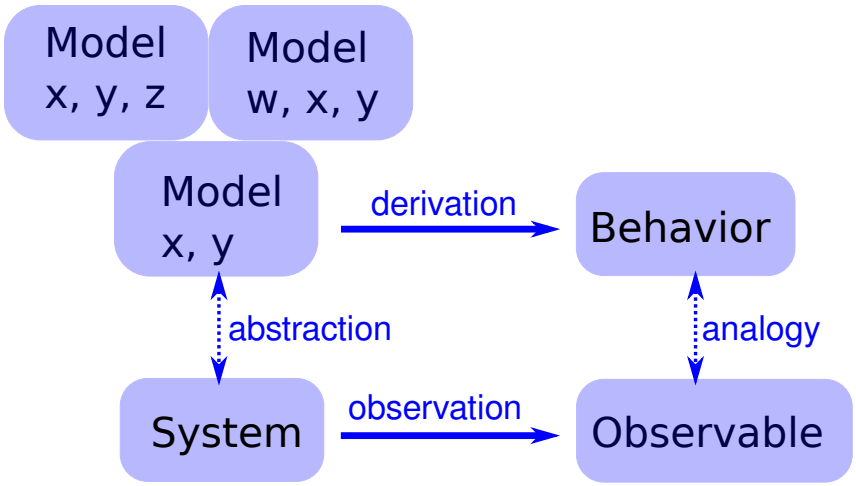

Figure 6.7: The logical structure of a simple physical model.

6.10 What is this a model of?

Some cellular automata are primarily mathematical artifacts. They are interesting because they are surprising, or useful, or pretty, or because they provide tools for creating new mathematics (like the Church-Turing thesis).

But it is not clear that they are models of physical systems. And if they are, they are highly abstracted, which is to say that they are not very detailed or realistic.

For example, some species of cone snail produce a pattern on their shells that resembles the patterns generated by cellular automata (seee `en.wikipedia.org/wiki/Cone_snail`). So it is natural to suppose that a CA is a model of the mechanism that produces patterns on shells as they grow. But, at least initially, it is not clear how the elements of the model (so-called cells, communication between neighbors, rules) correspond to the elements of a growing snail (real cells, chemical signals, protein interaction networks).

For conventional physical models, being realistic is a virtue, at least up to a point. If the elements of a model correspond to the elements of a physical system, there is an obvious analogy between the model and the system. In general, we expect a model that is more realistic to make better predictions and to provide more believable explanations.

Of course, this is only true up to a point. Models that are more detailed are harder to work with, and usually less amenable to analysis. At some point, a model becomes so complex that it is easier to experiment with the system.

At the other extreme, simple models can be compelling exactly because they are simple.

Simple models offer a different kind of explanation than detailed models. With a detailed model, the argument goes something like this: "We are interested in physical system S, so we construct a detailed model, M, and show by analysis and simulation that M exhibits a behavior, B, that is similar (qualitatively or quantitatively) to an observation of the real

system, O. So why does O happen? Because S is similar to M, and B is similar to O, and we can prove that M leads to B."

With simple models we can't claim that S is similar to M, because it isn't. Instead, the argument goes like this: "There is a set of models that share a common set of features. Any model that has these features exhibits behavior B. If we make an observation, O, that resembles B, one way to explain it is to show that the system, S, has the set of features sufficient to produce B."

For this kind of argument, adding more features doesn't help. Making the model more realistic doesn't make the model more reliable; it only obscures the difference between the essential features that cause O and the incidental features that are particular to S.

Figure 6.7 shows the logical structure of this kind of model. The features x and y are sufficient to produce the behavior. Adding more detail, like features w and z, might make the model more realistic, but that realism adds no explanatory power.

Chapter 7

Game of Life

One of the first cellular automata to be studied, and probably the most popular of all time, is a 2-D CA called "The Game of Life," or GoL for short. It was developed by John H. Conway and popularized in 1970 in Martin Gardner's column in *Scientific American*. See `http://en.wikipedia.org/wiki/Conway_Game_of_Life`.

The cells in GoL are arranged in a 2-D **grid**, either infinite in both directions or wrapped around. A grid wrapped in both directions is called a **torus** because it is topographically equivalent to the surface of a doughnut. See `http://en.wikipedia.org/wiki/Torus`.

Each cell has two states—live and dead—and 8 neighbors—north, south, east, west, and the four diagonals. This set of neighbors is sometimes called a Moore neighborhood.

The rules of GoL are **totalistic**, which means that the next state of a cell depends on the number of live neighbors only, not on their arrangement. The following table summarizes the rules:

Number of neighbors	Current state	Next state
2–3	live	live
0–1,4–8	live	dead
3	dead	live
0–2,4–8	dead	dead

This behavior is loosely analogous to real cell growth: cells that are isolated or overcrowded die; at moderate densities they flourish.

GoL is popular because:

- There are simple initial conditions that yield surprisingly complex behavior.
- There are many interesting stable patterns: some oscillate (with various periods) and some move like the spaceships in Wolfram's Rule 110 CA.
- Like Rule 110, GoL is Turing complete.

- Conway posed an intriguing conjecture—that there is no initial condition that yields unbounded growth in the number of live cells—and offered $50 to anyone who could prove or disprove it.
- The increasing availability of computers made it possible to automate the computation and display the results graphically. That turns out to be more fun than Conway's original implementation using a checkerboard.

7.1 Implementing Life

To implement GoL efficiently, we can take advantage of the multi-dimensional convolution function in SciPy. SciPy is a Python package that provides functions related to scientific computing. You can read about it at `http://www.scipy.org/`; if it is not already on your system, you might have to install it.

Convolution is an operation common in digital image processing, where an image is an array of pixels, and many operations involve computing a function of a pixel and its neighbors.

The neighborhood is described by a smaller array, called a **kernel** that specifies the location and **weight** of the neighbors. For example, this array:

```
kernel = numpy.array([[1,1,1],
                      [1,0,1],
                      [1,1,1]])
```

represents a neighborhood with eight neighbors, all with weight 1.

Convolution computes the weighted sum of the neighbors for each element of the array. So this kernel computes the sum of the neighbors (not including the center element).

For example, if `array` represents a GoL grid with 1s for live cells and 0s for dead cells we can use convolution to compute the number of neighbors for each cell.

```
import scipy.ndimage
neighbors = scipy.ndimage.filters.convolve(array, kernel)
```

Here's an implementation of GoL using `convolve`:

```
import numpy
import scipy.ndimage

class Life(object):

    def __init__(self, n, mode='wrap'):
        self.n = n
        self.mode = mode
        self.array = numpy.random.random_integers(0, 1, (n, n))
        self.weights = numpy.array([[1,1,1],
                                    [1,10,1],
                                    [1,1,1]])
```

```
    def step(self):
        con = scipy.ndimage.filters.convolve(self.array,
                                             self.weights,
                                             mode=self.mode)

        boolean = (con==3) | (con==12) | (con==13)
        self.array = numpy.int8(boolean)
```

The attributes of the `Life` object are `n`, the number of rows and columns in the grid, `mode`, which controls the behaviors of the boundary cells, `array`, which represents the grid, and `weights` which is the kernel used to count the neighbors.

The weight of the center cell is 10, so the number of neighbors is 0-8 for dead cells and 10-18 for live cells.

In `step`, `boolean` is a boolean array with `True` for live cells; `numpy.int8` converts it to integers.

To display an animated sequence of grids, I use `pyplot`. Animation in `pyplot` is a little awkward, but here's a class that manages it:

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as pyplot

class LifeViewer(object):

    def __init__(self, life, cmap=matplotlib.cm.gray_r):
        self.life = life
        self.cmap = cmap

        self.fig = pyplot.figure()
        pyplot.axis([0, life.n, 0, life.n])
        pyplot.xticks([])
        pyplot.yticks([])

        self.pcolor = None
        self.update()
```

`life` is a `Life` object. `cmap` is a color map provided by `matplotlib`; you can see the other color maps at `http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps`.

`self.fig` is a reference to the `matplotlib` figure, and `self.pcolor` is a reference to the **pseudocolor plot** created by `update`:

```
    def update(self):
        if self.pcolor:
            self.pcolor.remove()

        a = self.life.array
        self.pcolor = pyplot.pcolor(a, cmap=self.cmap)
        self.fig.canvas.draw()
```

If there is already a plot, we have to remove it; then we create a new one and invoke `draw` to update the display.

To run the animation, we need two methods:

```
def animate(self, steps=10):
    self.steps = steps
    self.fig.canvas.manager.window.after(1000, self.animate_callback)
    pyplot.show()

def animate_callback(self):
    for i in range(self.steps):
        self.life.step()
        self.update()
```

`animate` gets the animation started. It invokes `pyplot.show`, which sets up the GUI and waits for user events, but *first* it has to invoke `window.after` to set up a callback, so that `animate_callback` gets invoked after the window is set up. The first argument is the delay in milliseconds. The second argument is a bound method (see Chapter 19 of *Think Python*).

`animate_callback` invokes `step` to update the `Life` object and `update` to update the display.

Exercise 7.1. *Download my implementation of GoL from `thinkcomplex.com/Life.py`.*

Start the CA in a random state and run it until it stabilizes. What stable patterns can you identify?

7.2 Life patterns

If you run GoL from a random starting state, a number of stable patterns are likely to appear. Blocks, boats, beehives, blinkers and gliders are among the most common.

People have spent embarrassing amounts of time finding and naming these patterns. If you search the web, you will find many collections.

From most initial conditions, GoL quickly reaches a stable state where the number of live cells is nearly constant (usually with a small amount of oscillation).

But there are a some simple starting conditions that take a long time to settle down and yield a surprising number of live cells. These patterns are called "Methuselahs" because they are so long-lived.

One of the simplest is the r-pentomino, which has only five cells in the shape of a "r," hence the name. It runs for 1103 steps and yields 6 gliders, 8 blocks, 4 blinkers, 4 beehives, 1 boat, 1 ship, and 1 loaf. One of the longest-lived small patterns is rabbits, which starts with 9 live cells and takes 17 331 steps to stabilize.

Exercise 7.2. *Start with an r-pentomino as an initial condition and confirm that the results are consistent with the description above. You might have to adjust the size of the grid and the boundary behavior.*

7.3 Conway's conjecture

The existence of long-lived patterns brings us back to Conway's original question: are there initial patterns that never stabilize? Conway thought not, but he described two kinds of pattern that would prove him wrong, a "gun" and a "puffer train." A gun is a stable pattern that periodically produces a spaceship—as the stream of spaceships moves out from the source, the number of live cells grows indefinitely. A puffer train is a translating pattern that leaves live cells in its wake.

It turns out that both of these patterns exist. A team led by Bill Gosper discovered the first, a glider gun now called Gosper's Gun. Gosper also discovered the first puffer train. You can find descriptions and animations of these patterns in several places on the Web.

There are many patterns of both types, but they are not easy to design or find. That is not a coincidence. Conway chose the rules of GoL so that his conjecture would not be obviously true or false. Of all the possible rules for a 2-D CA, most yield simple behavior; most initial conditions stabilize quickly or grow unboundedly. By avoiding uninteresting CAs, Conway was also avoiding Wolfram's Class 1 and Class 2 behavior, and probably Class 3 as well.

If we believe Wolfram's Principle of Computational Equivalence, we expect GoL to be in Class 4. And it is. The Game of Life was proved Turing complete in 1982 (and again, independently, in 1983). Since then several people have constructed GoL patterns that implement a Turing machine or another machine known to be Turing complete.

Exercise 7.3. *Many named patterns are available in portable file formats. Modify* `Life.py` *to parse one of these formats and initialize the grid.*

7.4 Realism

Stable patterns in GoL are hard not to notice, especially the ones that move. It is natural to think of them as persistent entities, but remember that a CA is made of cells; there is no such thing as a toad or a loaf. Gliders and other spaceships are even less real because they are not even made up of the same cells over time. So these patterns are like constellations of stars. We perceive them because we are good at seeing patterns, or because we have active imaginations, but they are not real.

Right?

Well, not so fast. Many entities that we consider "real" are also persistent patterns of entities at a smaller scale. Hurricanes are just patterns of air flow, but we give them personal names. And people, like gliders, are not made up of the same cells over time. But even if you replace every cell in your body, we consider you the same person.

This is not a new observation—about 2500 years ago Heraclitus pointed out that you can't step in the same river twice—but the entities that appear in the Game of Life are a useful test case for thinking about **philosophical realism**.

In the context of philosophy, realism is the view that entities in the world exist independent of human perception and conception. By "perception" I mean the information that we get

from our senses, and by "conception" I mean the mental model we form of the world. For example, our vision systems perceive something like a 2-D projection of a scene, and our brains use that image to construct a 3-D model of the objects in the scene.

Scientific realism pertains to scientific theories and the entities they postulate. A theory postulates an entity if it is expressed in terms of the properties and behavior of the entity. For example, Mendelian genetics postulates a "gene" as a unit that controls a heritable characteristic. Eventually we discovered that genes are encoded in DNA, but for about 50 years, a gene was just a postulated entity. See `http://en.wikipedia.org/wiki/Gene`.

Again, I find it useful to state philosophical positions in a range of strengths, where SR1 is a weak form of scientific realism and SR4 is a strong form:

SR1: Scientific theories are true or false to the degree that they approximate reality, but no theory is exactly true. Some postulated entities may be real, but there is no principled way to say which ones.

SR2: As science advances, our theories become better approximations of reality. At least some postulated entities are known to be real.

SR3: Some theories are exactly true; others are approximately true. Entities postulated by true theories, and some entities in approximate theories, are real.

SR4: A theory is true if it describes reality correctly, and false otherwise. The entities postulated by true theories are real; others are not.

SR4 is so strong that it is probably untenable; by such a strict criterion, almost all current theories are known to be false. Most realists would accept something in the space between SR1 and SR3.

7.5 Instrumentalism

But SR1 is so weak that it verges on **instrumentalism**, which is the view that we can't say whether a theory is true or false because we can't know whether a theory corresponds to reality. Theories are instruments that we use for our purposes; a theory is useful, or not, to the degree that it is fit for its purpose.

To see whether you are comfortable with instrumentalism, consider the following statements:

> "Entities in the Game of Life aren't real; they are just patterns of cells that people have given cute names."
>
> "A hurricane is just a pattern of air flow, but it is a useful description because it allows us to make predictions and communicate about the weather."
>
> "Freudian entities like the Id and the Superego aren't real, but they are useful tools for thinking and communicating about psychology (or at least some people think so)."

> "Electrons are postulated entities in our best theories of electro-magnetism, but they aren't real. We could construct other theories, without postulating electrons, that would be just as useful."
>
> "Many of the things in the world that we identify as objects are arbitrary collections like constellations. For example, a mushroom is just the fruiting body of a fungus, most of which grows underground as a barely-contiguous network of cells. We focus on mushrooms for practical reasons like visibility and edibility."
>
> "Some objects have sharp boundaries, but many are fuzzy. For example, which molecules are part of your body: Air in your lungs? Food in your stomach? Nutrients in your blood? Nutrients in a cell? Water in a cell? Structural parts of a cell? Hair? Dead skin? Dirt? Bacteria on your skin? Bacteria in your gut? Mitochondria? How many of those molecules do you include when you weigh yourself. Conceiving the world in terms of discrete objects is useful, but the entities we identify are not real."

Give yourself one point for each statement you agree with. If you score 4 or more, you might be an instrumentalist!

If you are more comfortable with some of these statements than others, ask yourself why. What are the differences in these scenarios that influence your reaction? Can you make a principled distinction between them?

Exercise 7.4. *Read* `http://en.wikipedia.org/wiki/Instrumentalism` *and construct a sequence of statements that characterize instrumentalism in a range of strengths.*

7.6 Turmites

If you generalize the Turing machine to two dimensions, or add a read-write head to a 2-D CA, the result is a cellular automaton called a Turmite. It is named after a termite because of the way the read-write head moves, but spelled wrong as an homage to Alan Turing.

The most famous Turmite is Langton's Ant, discovered by Chris Langton in 1986. See `http://en.wikipedia.org/wiki/Langton_ant`.

The ant is a read-write head with four states, which you can think of as facing north, south, east or west. The cells have two states, black and white.

The rules are simple. During each time step, the ant checks the color of the cell is it on. If it is black, the ant turns to the right, changes the cell to white, and moves forward one space. If the cell is white, the ant turns left, changes the cell to black, and moves forward.

Given a simple world, a simple set of rules, and only one moving part, you might expect to see simple behavior—but you should know better by now. Starting with all white cells, Langton's ant moves in a seemingly random pattern for more than 10 000 steps before it enters a cycle with a period of 104 steps. After each cycle, the ant is translated diagonally, so it leaves a trail called the "highway."

If you start with multiple Turmites, they interact with each other in seemingly complex ways. If one Turmite is on the highway, another can follow it, overtake it, and cause it to reverse its pattern, moving back up the highway and leaving only white cells behind.