50 Examples Documentation

Release 1.0

A.M. Kuchling

Apr 13, 2017

Contents

Release 1.0

Date Apr 13, 2017

Contents:

CHAPTER 1

Introduction

Welcome to "50 Examples for Teaching Python".

My goal was to collect interesting short examples of Python programs, examples that tackle a real-world problem and exercise various features of the Python language. I envision this collection as being useful to teachers of Python who want novel examples that will interest their students, and possibly to teachers of mathematics or science who want to apply Python programming to make their subject more concrete and interactive.

Readers may also enjoy dipping into the book to learn about a particular algorithm or technique, and can use the references to pursue further reading on topics that especially interest them.

Python version

All of the examples in the book were written using Python 3, and tested using Python 3.2.1. You can download a copy of Python 3 from <http://www.python.org/download/>; use the latest version available.

This book is not a Python tutorial and doesn't try to introduce features of the language, so readers should either be familiar with Python or have a tutorial available. Classes and modules are used right from the beginning, for example; the early programs do not use a subset of the language that's later expanded by the introduction of more sophisticated features such as classes.

License

This license forbids publishing and selling copies of the book, but you are free to print copies for your own private or educational use. For example, it's OK to print out a section as a handout for a class, or to include sections in documents that you write and make available under the same Creative Commons license.

The original source text and example code is available at https://github.com/akuchling/50-examples, and is formatted using the Sphinx documentation tool (http://sphinx.pocoo.org).

Please send questions, suggestions, and comments to the e-mail address below.

A.M. Kuchling
amk@amk.ca

CHAPTER 2

Convert Fahrenheit to Celsius

We'll begin with a very simple example to illustrate the format used for each example. Each section will start with a discussion of the problem being solved, giving its requirements and sometimes a brief discussion of its history.

Most of the world uses the Celsius scale to indicate temperatures, but the United States still uses the Fahrenheit scale. We'll write a little script that takes Fahrenheit temperatures and prints their corresponding values in Celsius. (The author, who's a Canadian living in the United States, first wrote this script because when listening to weather reports he could never remember if 85 degrees Fahrenheit is pleasantly warm or scorchingly hot.)

Approach

Here we'll discuss the algorithm or approach taken to solving the problem.

The calculation for the temperature conversion is straightforward and references can be found all over the place. Celsius and Fahrenheit have different zero points – 0 degrees Celsius is 32 degrees Fahrenheit – so we need to subtract 32 from the Fahrenheit temperature.

The size of the units are also different. Celsius divides the temperature span between the freezing and boiling points of water into 100 degrees, while Fahrenheit divides this range into 180 degrees, so we need to multiply the value by 5/9 to turn 180 degrees into 100.

Solution

Next we'll present the complete program listing. The source code for this book also includes test suites for each program, but the test suites won't be shown in the book.

```
#!/usr/bin/env python3

import sys

def convert_f2c(S):
    """(str): float
```

```python

    Converts a Fahrenheit temperature represented as a string
    to a Celsius temperature.
    """
    fahrenheit = float(S)
    celsius = (fahrenheit - 32) * 5 / 9
    return celsius

def main():
    # If no arguments were given, print a helpful message
    if len(sys.argv) == 1:
        print('Usage: {} temp1 temp2 ...'.format(sys.argv[0]))
        sys.exit(0)

    # Loop over the arguments
    for arg in sys.argv[1:]:
        try:
            celsius = convert_f2c(arg)
        except ValueError:
            print("{!r} is not a numeric value".format(arg),
                  file=sys.stderr)
        else:
            print('{}\N{DEGREE SIGN}F = {:g}\N{DEGREE SIGN}C'.format(
                arg, round(celsius, 0)))

if __name__ == '__main__':
    main()
```

Code Discussion

Here we will dive further into the code, discussing particularly interesting sections of code, programming techniques, or larger issues.

The conversion to Celsius done by the `convert_f2c()` function is a straightforward calculation. An input string is converted to a floating-point value. If there's a problem, the resulting `ValueError` exception is not handled here but instead is left for the caller to catch.

Notice that the `main()` function tries to print a helpful message when no command-line arguments are provided, and it also catches the `ValueError`. For pedagogical programs like this, I will try to ensure the error handling is helpful to a user experimenting with the script.

Lessons Learned

Finally, does this program demonstrate any interesting themes about Python, about programming in general, or about its subject matter?

For this celsius example, I'll discuss Python's string `format()` method and the mini-language it uses, since it will be used throughout our example programs.

The strings used with the `format()` method will contain a replacement field specification inside curly brackets (`{ }`). It's possible to leave the specification empty (just `{}`) but we can also specify the position and type of the argument to use and how the resulting value should be rendered. A few examples:

```
'{!r}'.format(v)     # Equivalent to repr(v)
'{!s}'.format(v)     # Equivalent to str(v)

# Explicitly give the position of the argument to use for
# each field. Results in the string "arg2 arg1"
'{1} {0}'.format('arg1', 'arg2')
```

We can also specify formatting details such as the number of decimal places, style of numeric display, and left- or right-alignment. This comes within the curly brackets and following a `:` character.

We can left-align, right-align, or center a string within a desired output width, optionally giving the character to be used for padding:

```
>>> '{0:<15}'.format('left-justify')
'left-justify   '
>>> '{0:>15}'.format('right-justify')
'  right-justify'
>>> '{0:*^15}'.format('centered')
'***centered****'
```

We can output a value in binary, octal, decimal, or hexadecimal:

```
>>> '{0:b}'.format(1972)
'11110110100'
>>> '{0:o}'.format(1972)
'3664'
>>> '{0:d}'.format(1972)
'1972'
>>> '{0:x}'.format(1972)
'7b4'
```

We can request rounding to a specific number of decimal places, exponential notation, or displaying as a percentage:

```
>>> '{0:d}'.format(2**32)
'4294967296'
>>> '{0:e}'.format(2**32)
'4.294967e+09'
>>> '{0:%}'.format( 45 / 70 )
'64.285714%'
>>> '{0:.2%}'.format( 45 / 70 )
'64.29%'
```

The complete syntax for format strings is documented in the Python Library Reference at <http://docs.python.org/library/string.html#format-string-syntax>.

References

The references in each section will be to useful web pages, Wikipedia entries, software libraries, and books. Generally each reference is annotated with a short explanation of what's in it and why it might be of interest, to help you in deciding which references to pursue.

http://books.google.com/books?id=lnmrSAAACAAJ "A Matter of Degrees: What Temperature Reveals About the Past and Future of Our Species, Planet and Universe", by Gino Segré, is an entertaining tour through science using the concept of temperature as the connecting thread, including: biological aspects such as the regulation of body temperature and the thermophile species that cluster around deep-sea superheated vents; the theoretical arguments underlying global warming; the discovery of the laws of thermodynamics; low-temperature phenomena such as superconductivity and helium's superfluidity; and the temperature of the cosmic microwave background.

CHAPTER 3

Background: Algorithms

An *algorithm* specifies a series of steps that perform a particular computation or task. Algorithms were originally born as part of mathematics – the word "algorithm" comes from the Arabic writer Muammad ibn Mūsā al-Khwārizmī, – but currently the word is strongly associated with computer science. Throughout this book we'll examine a number of different algorithms to perform a variety of tasks.

Algorithms resemble recipes. Recipes tell you how to accomplish a task by performing a number of steps. For example, to bake a cake the steps are: preheat the oven; mix flour, sugar, and eggs throughly; pour into a baking pan; and so forth.

However, "algorithm" is a technical term with a more specific meaning than "recipe", and calling something an algorithm means that the following properties are all true:

- An algorithm is an unambiguous description that makes clear what has to be implemented. In a recipe, a step such as "Bake until done" is ambiguous because it doesn't explain what "done" means. A more explicit description such as "Bake until the cheese begins to bubble" is better. In a computational algorithm, a step such as "Choose a large number" is vague: what is large? 1 million, 1 billion, or 100? Does the number have to be different each time, or can the same number be used on every run?
- An algorithm expects a defined set of inputs. For example, it might require two numbers where both numbers are greater than zero. Or it might require a word, or a list of zero or more numbers.
- An algorithm produces a defined set of outputs. It might output the larger of the two numbers, an all-uppercase version of a word, or a sorted version of the list of numbers.
- An algorithm is guaranteed to terminate and produce a result, always stopping after a finite time. If an algorithm could potentially run forever, it wouldn't be very useful because you might never get an answer.
- Most algorithms are guaranteed to produce the correct result. It's rarely useful if an algorithm returns the largest number 99% of the time, but 1% of the time the algorithm fails and returns the smallest number instead.[1]
- If an algorithm imposes a requirement on its inputs (called a *precondition*), that requirement must be met. For

[1] There are special situations where algorithms that are sometimes wrong can still be useful. A good example is testing whether a number is prime. There's an algorithm called the Rabin-Miller test that's always correct when it reports a number is composite, but has a 25% chance of being wrong when it reports a number is prime. One test therefore isn't enough to conclude you've found a prime, but you can perform repeated tests and reduce the chance of being wrong to as low as you like (but never zero).

example, a precondition might be that an algorithm will only accept positive numbers as an input. If preconditions aren't met, then the algorithm is allowed to fail by producing the wrong answer or never terminating.

Studying algorithms is a fundamental part of computer science. There are several different characteristics of an algorithm that are useful to know:

1. Does an algorithm actually exist to perform a given task?
2. If someone proposes an algorithm to solve a task, are we sure that the algorithm works for all possible inputs?
3. How long does the algorithm take to run? How much memory space does it require?
4. Once we know it's possible to solve a problem with an algorithm, a natural question is whether the algorithm is the best possible one. Can the problem be solved more quickly?

Most of these questions will be discussed for the algorithms covered in this book.

An Example Algorithm

Let's look at a very simple algorithm called `find_max()`.

Problem: Given a list of positive numbers, return the largest number on the list.

Inputs: A list `L` of positive numbers. This list must contain at least one number. (Asking for the largest number in a list of no numbers is not a meaningful question.)

Outputs: A number `n`, which will be the largest number of the list.

Algorithm:

1. Set `max` to 0.
2. For each number `x` in the list `L`, compare it to `max`. If `x` is larger, set `max` to `x`.
3. `max` is now set to the largest number in the list.

An implementation in Python:

```python
def find_max (L):
    max = 0
    for x in L:
        if x > max:
            max = x
    return max
```

Does this meet the criteria for being an algorithm?

- *Is it unambiguous?* Yes. Each step of the algorithm consists of primitive operations, and translating each step into Python code is very easy.
- *Does it have defined inputs and outputs?* Yes.
- *Is it guaranteed to terminate?* Yes. The list `L` is of finite length, so after looking at every element of the list the algorithm will stop.
- *Does it produce the correct result?* Yes. In a formal setting you would provide a careful proof of correctness. In the next section I'll sketch a proof for an alternative solution to this problem.

A Recursive Version of `find_max()`

There can be many different algorithms for solving the same problem. Here's an alternative algorithm for `find_max()`:

1. If `L` is of length 1, return the first item of `L`.
2. Set `v1` to the first item of `L`.
3. Set `v2` to the output of performing `find_max()` on the rest of `L`.
4. If `v1` is larger than `v2`, return `v1`. Otherwise, return `v2`.

Implementation:

```
def find_max (L):
    if len(L) == 1:
        return L[0]
    v1 = L[0]
    v2 = find_max(L[1:])
    if v1 > v2:
        return v1
    else:
        return v2
```

Let's ask our questions again.

- *Is it unambiguous?* Yes. Each step is simple and easily translated into Python.
- *Does it have defined inputs and outputs?* Yes.
- *Is it guaranteed to terminate?* Yes. The algorithm obviously terminates if `L` is of length 1. If `L` has more than one element, `find_max()` is called with a list that's one element shorter and the result is used in a computation.

  Does the nested call to `find_max()` always terminate? Yes. Each time, `find_max()` is called with a list that's shorter by one element, so eventually the list will be of length 1 and the nested calls will end.

Finally, *does it produce the correct result?* Yes. Here's a sketch of a proof.[2]

Consider a list of length 1. In this case the largest number is also the only number on the list. `find_max()` returns this number, so it's correct for lists of length 1.

Now consider a longer list of length `N+1`, where `N` is some arbitrary length. Let's assume that we've proven that `find_max()` is correct for all lists of length `N`. The value of `v2` will therefore be the largest value in the rest of the list. There are two cases to worry about.

- Case 1: `v1`, the first item of the list, is the largest item. In that case, there are no other values in the list greater than `v1`. We're assuming `find_max()` is correct when executed on the rest of the list, so the value it returns will be less than `v1`. The `if v1 > v2` comparison will therefore be true, so the first branch will be taken, returning `v1`. This is the largest item in the list, so in this case the algorithm is correct.
- Case 2: `v1`, the first item of the list, is *not* the largest item. In that case, there is at least one value in the list that's greater than `v1`. `find_max()` is correct for the shortened version of the rest of the list, returning the maximum value it contains, so this value must be greater than `v1`. The `if v1 > v2` comparison will therefore be false, so the `else` branch will be taken, returning `v2`, the largest value in the rest of the list. This case assumes that `v1` is not the largest value, so `v2` is therefore the largest value, and the algorithm is also correct in this case.

With these two cases, we've now shown that if `find_max()` is correct for lists of length `N`, it's also correct for lists of length `N+1`. In the first part of our argument, we've shown that `find_max()` is correct for lists of length 1. Therefore, it's also correct for lists that are 2 elements long, and 3 elements, and 4, 5, 6, ... up to any number.

[2] It's possible to write formal proofs of correctness for an algorithm, but the resulting proofs are lengthy even for short algorithms such as this one.

This may seem like a trick; we showed that it's correct for the trivial case of the single-element list, and then showed that it's correct on a problem of a certain size. Such proofs are called *inductive proofs*, and they're a well-known mathematical technique for proving a theorem.

Carrying out an inductive proof of some property requires two steps.

1. First, you show that the property is true for some simple case: an empty list or a list of length 1, an empty set, a single point. Usually this demonstration is very simple; often it's obviously true that the property is true. This is called the *basis case*.
2. Next, you assume the property is true for size N and show that it's true for some larger size such as N+1. This is called the *inductive step*, and is usually the more difficult one.

Once you have both demonstrations, you've proven the property is true for an infinite number of values of N; correctness for N=1 implies that the N=2 case is also correct, which in turn implies correctness for N=3, 4, 5, and every other positive integer. Not every theorem can be put into a form where an inductive proof can be used.

References

XXX something on induction

CHAPTER 4

Background: Measuring Complexity

It's obviously most important that an algorithm solves a given problem correctly. How much time an algorithm will take to solve a problem is only slightly less important. All algorithms must terminate eventually, because they wouldn't be algorithms if they didn't, but they might run for billions of years before terminating. In order to compare algorithms, we need a way to measure the time required by an algorithm.

To characterize an algorithm, we really need to know how its running time changes in relation to the size of a problem. If we solve a problem that's ten times as large, how does the running time change? If we run `find_max()` on a list that's a thousand elements long instead of a hundred elements, does it take the same amount of time? Does it take 10 times as long to run, 100 times, or 5 times? This is called the algorithm's *time complexity* or, occasionally, its *scalability*.

To measure the time complexity, we could simply implement an algorithm on a computer and time it on problems of different sizes. For example, we could run `find_max()` on lists from lengths ranging from 1 to 1000 and graph the results. This is unsatisfactory for two reasons:

- Computers perform different operations at different speeds: addition may be very fast and division very slow. Different computers may have different specialities. One machine may have very fast math but slow string operations while another might do math very slowly. Machines also vary in memory size and processor, memory, and disk speeds. Researchers would find it difficult to compare results measured on different machines.
- Measuring performance within a given range doesn't tell us if the algorithm continues to scale outside of the range. Perhaps it runs very well for problem sizes up to 1000, but at some larger size it began to run too slowly.

Instead, the measurement is done more abstractly by counting the number of basic operations required to run the algorithm, after defining what is counted as an operation. For example, if you wanted to measure the time complexity of computing a sine function, you might assume that only addition, subtraction, multiplication, and division are basic operations. On the other hand, if you were measuring the time to draw a circle, you might include sine as a basic operation.

Complexity is expressed using *big-O notation*. The complexity is written as O(<some function>), meaning that the number of operations is proportional to the given function multiplied by some constant factor. For example, if an algorithm takes 2*(n**2) operations, the complexity is written as O(n**2), dropping the constant multiplier of 2.

Some of the most commonly seen complexities are:

- O(1) is *constant-time* complexity. The number of operations for the algorithm doesn't actually change as the problem size increases.
- O(log n) is *logarithmic* complexity. The base used to take the logarithm makes no difference, since it just multiplies the operation count by a constant factor. The most common base is base 2, written as log_2 or lg.

  Algorithms with logarithmic complexity cope quite well with increasingly large problems. Doubling the problem size requires adding a fixed number of new operations, perhaps just one or two additional steps.
- O(n) time complexity means that an algorithm is *linear*; doubling the problem size also doubles the number of operations required.
- O(n**2) is *quadratic* complexity. Doubling the problem size multiplies the operation count by four. A problem 10 times larger takes 100 times more work.
- O(n**3), O(n**4), O(n**5), etc. are *polynomial* complexity.
- O(2**n) is *exponential* complexity. Increasing the problem size by 1 unit doubles the work. Doubling the problem size squares the work. The work increases so quickly that only the very smallest problem sizes are feasible.

The following graph compares the growth rates of various time complexities.

When writing down big-O notation, we can keep only the fastest-growing term and drop slower-growing terms. For example, instead of writing O(n**2 + 10n + 5), we drop the lower terms and write only O(n**2). The smaller terms don't contribute very much to the growth of the function as `n` increases. If `n` increases by a factor of 100, the `n**2` term increases the work by a factor of 10,000. The increase of 1000 operations from the `10n` term dwindles to insignificance.

After correctness, time complexity is usually the most interesting property of an algorithm, but in certain cases the amount of memory or storage space required by an algorithm is also of interest. These quantities are also expressed using big-O notation. For example, one algorithm might have O(n) time and use no extra memory while another algorithm might take only O(1) time by using O(n) extra storage space. In this case, the best algorithm to use will vary depending on the environment where you're going to be running it. A cellphone has very little memory, so you might choose the first algorithm in order to use as little memory as possible, even if it's slower. Current desktop computers usually have gigabytes of memory, so you might choose the second algorithm for greater speed and live with using more memory.

Big-O notation is an upper bound, expressing the worst-case time required to run an algorithm on various inputs. Certain inputs, however, may let the algorithm run more quickly. For example, an algorithm to search for a particular item in a list may be lucky and find a match on the very first item it tries. The work required in the best-case speed may therefore be much less than that required in the worst case.

Another notation is used for the best-case time, *big-omega notation*. If an algorithm is Omega(<some function>), the best-case time is proportional to the function multiplied by some constant factor. For example, the quicksort algorithm discussed later in this book is Omega(n lg n) and O(n**2). For most inputs quicksort requires time proportional to `n lg n`, but for certain inputs time proportional to `n**2` will be necessary.

Big-theta notation combines both upper and lower bounds; if an algorithm is both O(function) and Omega(function), it is also Theta(function). The function is therefore a tight bound on both the upper and lower limits of the running time.

Usually the worst case is what we're interested in. It's important that we know the longest possible time an algorithm might take so that we can determine if we can solve problems within a reasonable time. Occasionally encountering a particular input that can be solved more quickly may be lucky when it happens, but it can't be relied upon, so the best-case time usually isn't very relevant. For most of the algorithms in this book, only the O() bound will discussed.

References

XXX

CHAPTER 5

Simulating The Monty Hall Problem

The Monty Hall problem is a well-known puzzle in probability derived from an American game show, *Let's Make a Deal*. (The original 1960s-era show was hosted by Monty Hall, giving this puzzle its name.) Intuition leads many people to get the puzzle wrong, and when the Monty Hall problem is presented in a newspaper or discussion list, it often leads to a lengthy argument in letters-to-the-editor and on message boards.

The game is played like this:

1. The game show set has three doors. A prize such as a car or vacation is behind one door, and the other two doors hide a valueless prize called a Zonk; in most discussions of the problem, the Zonk is a goat.
2. The contestant chooses one door. We'll assume the contestant has no inside knowledge of which door holds the prize, so the contestant will just make a random choice.
3. The smiling host Monty Hall opens one of the other doors, always choosing one that shows a goat, and always offers the contestant a chance to switch their choice to the remaining unopened door.
4. The contestant either chooses to switch doors, or opts to stick with the first choice.
5. Monty calls for the remaining two doors to open, and the contestant wins whatever is behind their chosen door.

Let's say a hypothetical contestant chooses door #2. Monty might then open door #1 and offer the chance to switch to door #3. The contestant switches to door #3, and then we see if the prize is behind #3.

The puzzle is: what is the best strategy for the contestant? Does switching increase the chance of winning the car, decrease it, or make no difference?

The best strategy is to make the switch. It's possible to analyze the situation and figure this out, but instead we'll tackle it by simulating thousands of games and measuring how often each strategy ends up winning.

Approach

Simulating one run of the game is straightforward. We will write a `simulate()` function that uses Python's `random` module to pick which door hides the prize, the contestant's initial choice, and which doors Monty chooses to open. An input parameter controls whether the contestant chooses to switch, and `simulate()` will then return a Boolean telling whether the contestant's final choice was the winning door.

Part of the reason the problem fools so many people is that in the three-door case the probabilities involved are 1/3 and 1/2, and it's easy to get confused about which probability is relevant. Considering the same game with many more doors makes reasoning about the problem much clearer, so we'll make the number of doors a configurable parameter of the simulation script.

Solution

The simulation script is executed from the command line. If you supply no arguments, the script will use three doors and run 10,000 trials of both the switching and not-switching strategies. You can supply `--doors=100` to use 100 doors and `--trials=1000` to run a smaller number of trials.

```python
#!/usr/bin/env python3

"""Simulate the Monty Hall problem.

"""

import argparse, random

def simulate(num_doors, switch, verbose):
    """(int, bool): bool

    Carry out the game for one contestant.  If 'switch' is True,
    the contestant will switch their chosen door when offered the chance.
    Returns a Boolean value telling whether the simulated contestant won.
    """

    # Doors are numbered from 0 up to num_doors-1 (inclusive).

    # Randomly choose the door hiding the prize.
    winning_door = random.randint(0, num_doors-1)
    if verbose:
        print('Prize is behind door {}'.format(winning_door+1))

    # The contestant picks a random door, too.
    choice = random.randint(0, num_doors-1)
    if verbose:
        print('Contestant chooses door {}'.format(choice+1))

    # The host opens all but two doors.
    closed_doors = list(range(num_doors))
    while len(closed_doors) > 2:
        # Randomly choose a door to open.
        door_to_remove = random.choice(closed_doors)

        # The host will never open the winning door, or the door
        # chosen by the contestant.
        if door_to_remove == winning_door or door_to_remove == choice:
            continue

        # Remove the door from the list of closed doors.
        closed_doors.remove(door_to_remove)
        if verbose:
            print('Host opens door {}'.format(door_to_remove+1))

    # There are always two doors remaining.
    assert len(closed_doors) == 2

    # Does the contestant want to switch their choice?
    if switch:
        if verbose:
            print('Contestant switches from door {} '.format(choice+1), end='')

        # There are two closed doors left.  The contestant will never
        # choose the same door, so we'll remove that door as a choice.
        available_doors = list(closed_doors) # Make a copy of the list.
        available_doors.remove(choice)

        # Change choice to the only door available.
        choice = available_doors.pop()
        if verbose:
            print('to {}'.format(choice+1))

    # Did the contestant win?
    won = (choice == winning_door)
    if verbose:
        if won:
            print('Contestant WON', end='\n\n')
        else:
            print('Contestant LOST', end='\n\n')
    return won


def main():
    # Get command-line arguments
    parser = argparse.ArgumentParser(
        description='simulate the Monty Hall problem')
    parser.add_argument('--doors', default=3, type=int, metavar='int',
                        help='number of doors offered to the contestant')
    parser.add_argument('--trials', default=10000, type=int, metavar='int',
                        help='number of trials to perform')
    parser.add_argument('--verbose', default=False, action='store_true',
                        help='display the results of each trial')
    args = parser.parse_args()

    print('Simulating {} trials...'.format(args.trials))

    # Carry out the trials
    winning_non_switchers = 0
    winning_switchers = 0
    for i in range(args.trials):
        # First, do a trial where the contestant never switches.
        won = simulate(args.doors, switch=False, verbose=args.verbose)
        if won:
            winning_non_switchers += 1

        # Next, try one where the contestant switches.
        won = simulate(args.doors, switch=True, verbose=args.verbose)
        if won:
            winning_switchers += 1

    print('    Switching won {0:5} times out of {1} ({2}% of the time)'.format(
            winning_switchers, args.trials,
            (winning_switchers / args.trials * 100 ) ))
    print('Not switching won {0:5} times out of {1} ({2}% of the time)'.format(
            winning_non_switchers, args.trials,
            (winning_non_switchers / args.trials * 100 ) ))


if __name__ == '__main__':
    main()
```

A sample run:

```
-> code/monty-hall.py
Simulating 10000 trials...
    Switching won  6639 times out of 10000 (66.39% of the time)
Not switching won  3357 times out of 10000 (33.57% of the time)
->
```

Our simulation confirms the result: it's better to switch, which wins the car more often. If you switch, you have a 2/3 probability of winning the car; if you don't switch, you'll only win the car 1/3 of the time. The numbers from our simulation bear this out, though our random trials usually won't result in percentages that are *exactly* 66.6% or 33.3%.

If you supply the `--verbose` switch, the simulator will print out each step of the game so you can work through some examples. Be sure to use `--trials` to run a smaller number of trials:

```
-> code/monty-hall.py  --verbose --trials=2
Simulating 2 trials...
Prize is behind door 2
Contestant chooses door 3
Host opens door 1
Contestant LOST

Prize is behind door 3
Contestant chooses door 1
Host opens door 2
Contestant switches from door 1 to 3
Contestant WON

Prize is behind door 2
Contestant chooses door 3
Host opens door 1
Contestant LOST

Prize is behind door 3
Contestant chooses door 3
Host opens door 1
Contestant switches from door 3 to 2
Contestant LOST

    Switching won     1 times out of 2 (50.0% of the time)
Not switching won     0 times out of 2 (0.0% of the time)
->
```

Code Discussion

The command-line arguments are parsed using the `argparse` module, and the resulting values are passed into the `simulate()` function.

When there are **num_doors** doors, `simulate()` numbers the doors from 0 up to **num_doors**-1. `random.randint(a, b)()` picks a random integer from the range **a** to **b**, possibly choosing one of the endpoints, so here we use `random.randint(0, num_doors-1)()`.

To figure out which doors the host will open, the code makes a list of the currently closed doors, initially containing all the integers from 0 to **num_doors**-1. Then the code loops, picking a random door from the list to open. By our description of the problem, Monty will never open the contestant's door or the one hiding the prize, so the loop excludes those two doors and picks a different door. The loop continues until only two doors remain, so Monty will always open **num_doors**-2 doors.

To implement the contestant's switching strategy, we take the list of closed doors, which is now 2 elements long, and remove the contestant's current choice. The remaining element is therefore the door they're switching to.

Lessons Learned

This approach to answering a question, where we randomly generate many possible inputs, calculate the outcomes, and summarize the results, is called Monte Carlo simulation and has a long history, having been first developed in the 1940s by mathematicians working on the Manhattan Project to build an atomic bomb.

In the case of the Monty Hall problem, the simulation is straightforward to program and we can figure out an analytical result, so it's easy to inspect the output and verify that the program is correct. Often, though, simulations are for attacking problems too complicated to be solved beforehand and then checking for correctness is much harder. The programmers will need to carefully validate their code by unit-testing the simulation's internal functions and adding checks for internal correctness. `monty-hall.py` does this in a small way with an `assert` statement that will raise an exception if the number of closed doors is not equal to 2, which would indicate some sort of failure in the `simulate()` function's logic or input data. We could potentially add more such checks:

```
assert 0 <= winning_door < num_doors, 'Winning door is not a legal door'
assert 0 <= choice < num_doors, 'Contestant choice is not a legal door'
```

In our simple simulation, these assertions are never going to fail, but perhaps we might make changes to `simulate()` that make the function incorrect, or perhaps someday a bug in Python's `random` module will come to light.

References

http://en.wikipedia.org/wiki/Monty_Hall_problem Discusses the history of the problem and various approaches to solving it.

http://library.lanl.gov/cgi-bin/getfile?00326867.pdf "Stan Ulam, John von Neumann, and the Monte Carlo Method" (1987), by Roger Eckhardt, is a discussion of the very first Monte Carlo simulation and some of the mathematical problems encountered while implementing it. This first simulation modelled neutrons diffusing through fissionable material in an effort to determine whether a chain reaction would occur.

(The PDF also features a discussion of how random number generators work, written by Tony Warnock.)

http://www.youtube.com/watch?v=0W978thuweY A condensed episode of an episode of the original game show, showing Monty Hall's quick wit in action. Notice that the original game is more complicated than the Monty Hall puzzle as described above because Monty has many more actions available to him: he can offer the choice of an unknown prize or an unknown amount of cash, or suggest trading in what you've already won for a different unknown prize.

CHAPTER 6

Background: Drawing Graphics

Throughout the rest of this book, we'll need to display graphical output. There are many different graphical toolkits available for Python; the *References* section lists some of them. For this book I chose one of the simplest: the `turtle` module.

My reasons for selecting it are:

- `turtle` is included in the binary installers downloadable from python.org. No extra packages are required to use it.
- `turtle` can be used for drawing with Cartesian coordinates by calling the `setposition()` method, but the turtle primitives are also useful for constructing interesting examples. Most other toolkits only support Cartesian plotting.

Unfortunately the module doesn't support printed output, but I think that isn't much of a disadvantage because interactive graphics are more interesting for modern learners.

Turtle graphics are an approach with a long history. Originally the turtle was a physical object, a robot that could be placed on a large sheet of paper and directed to move. Then the turtle become a visual abstraction on a high-resolution screen, often represented as a triangle. Even in a purely graphical format, the concept of the turtle can make it easier to picture what actions are being carried out, and hence what a program is doing.

Approach

The `turtle` module provides an environment where turtles move upon a 2-dimensional grid. Turtles have a position, a heading (the direction in which the turtle is facing), and a variety of possible states (turtles can draw lines in a particular colour when they move or leave no trace) and actions (turning left or right; moving forward or backward.

Here's a brief overview of the functions and methods used in this book. Consult *Reference Card: Turtle Graphics* when you need some help remembering a method, and the Python Library Reference's documentation for the `turtle` module for a complete description of all the module's features.

(The module often has synonyms for the same action; I've chosen the one I think is the clearest. For example, the methods `backward(dist)`, `back(dist)`, and `bk(dist)` all do the same thing, moving the turtle backwards. I've chosen to use `back(dist)` consistently.)

You create a turtle by calling `turtle.Turtle()`. Doing this will automatically pop up a separate window (called a `Screen`) on your computer's display. You can call `turtle.Screen` to get an object representing this window; it has a few methods such as `title()` to set the title, `screensize()` to get the size of the canvas, and `clear()` to restore the screen's contents to their initial state.

A `Turtle` object has many methods that can be grouped into families. There are methods for controlling the turtle's motion:

- `forward(distance)` moves the turtle forward **distance** pixels, in whatever direction the turtle is pointing.
- `back(distance)` moves backward **distance** pixels.
- `left(angle)` and `right(angle)` change the turtle's orientation without moving it. By default angles are measured in degrees, but you can call the turtle's `radians()` method to use radians in future calls to `left()` and `right()`.
- The turtle's movements aren't normally performed instantly, but instead are slowed down and animated so that the eye can follow what the turtle is doing. You can change the speed of the turtle's motion by calling `speed(value)`, where **value** is a string giving a speed; "fastest" results in instananeous motion, and "fast", "normal", "slow", and "slowest" are progressively slower speeds.
- The turtle is usually drawn as an arrowhead. The `hideturtle()` method prevents the turtle from being displayed, and `showturtle()` brings it back.

To read the turtle's position and heading:

- `pos()` returns a tuple giving the **(x,y)** coordinate where the turtle is currently located. `xcor()` and `ycor()` return just the X or Y coordinate.
- `heading()` returns the turtle's heading, usually in degrees (but if you've previously called `radians()` the result will be measured in radians).

To move the turtle to a particular coordinate and orientation:

- `setpos(x, y)` moves the turtle to the given coordinate, drawing a line if the pen is down. You can also provide a pair of coordinates as a single argument.
- `setheading(angle)` sets the turtle's orientation to **angle**. Usually 0 degrees is east, 90 is north, 180 is west, and 270 is south.
- `home()` returns the turtle to position (0,0) and resets its orientation to east.

The turtle can draw a line behind it as it moves. To control this line:

- `pendown()` puts the pen down on the paper (metaphorically), so the turtle will leave a line as it moves.
- `penup()` raises the pen from the paper, so the turtle will move without leaving any trace.
- `pencolor(color)` sets the colour of the line traced. **color** is a string giving a primary colour name, such as "red" or "yellow", or an RGB colour specification such as "#33cc8c". (The database of colour names is limited, so specific names such as "crimson" or "octarine" won't work, but simple names such as "red", "blue", and "green" are understood.)
- `pensize(width)` sets the width of the line traced. The width starts out as 1 pixel, but can be changed using this method.

The turtle can also stamp its image on the display:

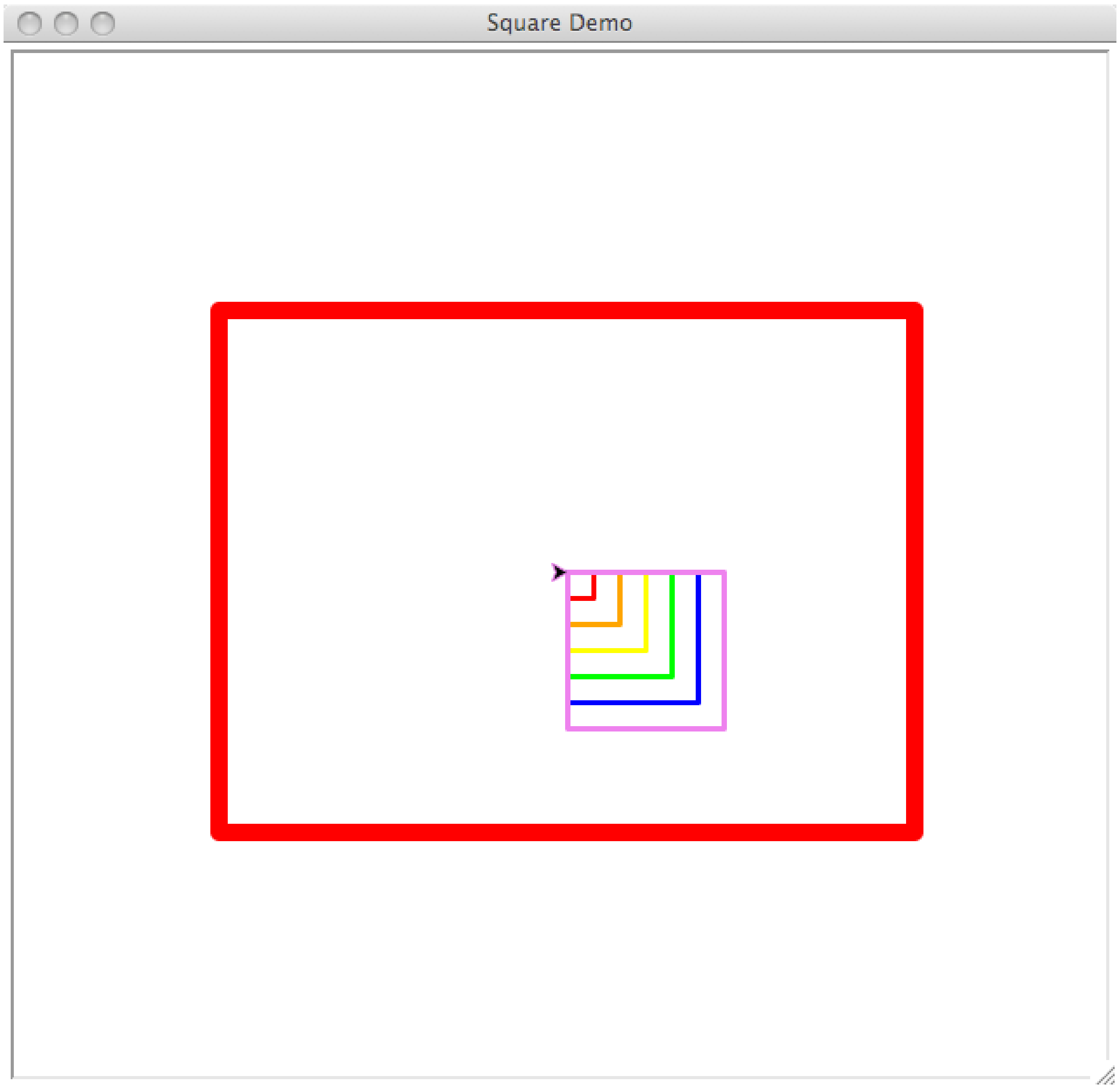
- `stamp()` records a copy of the turtle's shape onto the canvas. This method returns an integer stamp ID, so that you can remove the image later by calling `clearstamp()` and passing it the ID.
- `dot(size, color)` draws a circular dot of the given size and colour. The colour is optional; if not supplied, the turtle's current pen colour is used.
- The turtle `reset()` method clears all of the drawings made by that turtle and returns it to the home position.

Example

This program doesn't exercise every single method – that would be tediously long – but it shows what turtle graphics are like by drawing some simple graphics and then waiting for a keypress before exiting.

```python
#!/usr/bin/env python3

import sys
import turtle

def border(t, screen_x, screen_y):
    """(Turtle, int, int)

    Draws a border around the canvas in red.
    """
    # Lift the pen and move the turtle to the center.
    t.penup()
    t.home()

    # Move to lower left corner of the screen; leaves the turtle
    # facing west.
    t.forward(screen_x / 2)
    t.right(90)
    t.forward(screen_y / 2)
    t.setheading(180)           # t.right(90) would also work.

    # Draw the border
    t.pencolor('red')
    t.pendown()
    t.pensize(10)
    for distance in (screen_x, screen_y, screen_x, screen_y):
        t.forward(distance)
        t.right(90)

    # Raise the pen and move the turtle home again; it's a good idea
    # to leave the turtle in a known state.
    t.penup()
    t.home()

def square(t, size, color):
    """(Turtle, int, str)

    Draw a square of the chosen colour and size.
    """
    t.pencolor(color)
    t.pendown()
    for i in range(4):
        t.forward(size)
        t.right(90)

def main():
    # Create screen and turtle.
    screen = turtle.Screen()
    screen.title('Square Demo')
    screen_x, screen_y = screen.screensize()
    t = turtle.Turtle()

    # Uncomment to draw the graphics as quickly as possible.
    ##t.speed(0)

    # Draw a border around the canvas
    border(t, screen_x, screen_y)

    # Draw a set of nested squares, varying the color.
    # The squares are 10%, 20%, etc. of half the size of the canvas.
    colors = ['red', 'orange', 'yellow', 'green', 'blue', 'violet']
    t.pensize(3)
    for i, color in enumerate(colors):
        square(t, (screen_y / 2) / 10 * (i+1), color)

    print('Hit any key to exit')
    dummy = input()

if __name__ == '__main__':
    main()
```

The display resulting from this program is:



Code Discussion

One thing to learn from the demo program is that drawing functions such as `border()` and `square()` should be careful about the state of the turtle they expect at the beginning and the state it's left in afterwards. A frequent error is to leave the turtle pointing in an unexpected direction, causing later actions to be carried out in the wrong place. Watching the animated turtle usually makes such mistakes apparent.

References

http://cairographics.org/ Cairo is a 2D graphics library with a Python API that supports both screen and printed output.

"Turtle Geometry: The Computer as a Medium for Exploring Mathematics" By Harold Abelson and Andrea diSessa. A 1981 textbook that begins with polygons and ends with the curved spacetime of general relativity, using turtle graphics both to draw illustrative examples and as a conceptual model. ISBN 978-0-262-01063-4 (ISBN 978-0-262-51037-0 for the paperback).

CHAPTER 7

Simulating Planetary Orbits

According to Isaac Newton, the force of gravitational attraction between two objects is given by:

$$F = G\frac{m_1 m_2}{d^2}$$

where G is the gravitational constant, m_1 and m_2 are the masses of the two objects, and d is the distance between them. In SI units, G has the value $6.67428 \times 10^{-11} N(m/kg)^2$, so d is measured in meters, the masses are measured in kilograms, and the resulting F is in newtons.

Using this equation, Newton determined a formula for calculating how long it took an object to complete an orbit around a central mass. However, when dealing with three or more objects, it's generally not possible to find a tidy formula to calculate what the three bodies will do.

Instead, such problems are tackled by numeric integration, a brute-force approach where you take all the object positions and velocities at time T, calculate the forces they exert on each other, update the velocities, and calculate the new positions at time $T + \epsilon$. Then you repeat this in a loop, stepping forward through time, and output or plot the results.

Approach

To implement this in Python, we'll use the `turtle` module to provide a graphical display, subclassing the `Turtle` class to create a `Body` class that will have additional attributes: `mass` for the object's mass, `vx` and `vy` for its velocity, and `px` and `py` for its position.

An added method on `Body`, `attraction()`, will take another `Body` instance and return the X and Y components of the force exerted by the other body.

Solution

```
#!/usr/bin/env python3

import math
```

```
from turtle import *

# The gravitational constant G
G = 6.67428e-11

# Assumed scale: 100 pixels = 1AU.
AU = (149.6e6 * 1000)     # 149.6 million km, in meters.
SCALE = 250 / AU

class Body(Turtle):
    """Subclass of Turtle representing a gravitationally-acting body.

    Extra attributes:
    mass : mass in kg
    vx, vy: x, y velocities in m/s
    px, py: x, y positions in m
    """

    name = 'Body'
    mass = None
    vx = vy = 0.0
    px = py = 0.0

    def attraction(self, other):
        """(Body): (fx, fy)

        Returns the force exerted upon this body by the other body.
        """
        # Report an error if the other object is the same as this one.
        if self is other:
            raise ValueError("Attraction of object %r to itself requested"
                             % self.name)

        # Compute the distance of the other body.
        sx, sy = self.px, self.py
        ox, oy = other.px, other.py
        dx = (ox-sx)
        dy = (oy-sy)
        d = math.sqrt(dx**2 + dy**2)

        # Report an error if the distance is zero; otherwise we'll
        # get a ZeroDivisionError exception further down.
        if d == 0:
            raise ValueError("Collision between objects %r and %r"
                             % (self.name, other.name))

        # Compute the force of attraction
        f = G * self.mass * other.mass / (d**2)

        # Compute the direction of the force.
        theta = math.atan2(dy, dx)
        fx = math.cos(theta) * f
        fy = math.sin(theta) * f
        return fx, fy

def update_info(step, bodies):
    """(int, [Body])

    Displays information about the status of the simulation.
    """
    print('Step #{}'.format(step))
    for body in bodies:
        s = '{:<8}  Pos.={:>6.2f} {:>6.2f} Vel.={:>10.3f} {:>10.3f}'.format(
            body.name, body.px/AU, body.py/AU, body.vx, body.vy)
        print(s)
    print()

def loop(bodies):
    """([Body])

    Never returns; loops through the simulation, updating the
    positions of all the provided bodies.
    """
    timestep = 24*3600  # One day

    for body in bodies:
        body.penup()
        body.hideturtle()

    step = 1
    while True:
        update_info(step, bodies)
        step += 1

        force = {}
        for body in bodies:
            # Add up all of the forces exerted on 'body'.
            total_fx = total_fy = 0.0
            for other in bodies:
                # Don't calculate the body's attraction to itself
                if body is other:
                    continue
                fx, fy = body.attraction(other)
                total_fx += fx
                total_fy += fy

            # Record the total force exerted.
            force[body] = (total_fx, total_fy)

        # Update velocities based upon on the force.
        for body in bodies:
            fx, fy = force[body]
            body.vx += fx / body.mass * timestep
            body.vy += fy / body.mass * timestep

            # Update positions
            body.px += body.vx * timestep
            body.py += body.vy * timestep
            body.goto(body.px*SCALE, body.py*SCALE)
            body.dot(3)


def main():
    sun = Body()
    sun.name = 'Sun'
    sun.mass = 1.98892 * 10**30
    sun.pencolor('yellow')

    earth = Body()
    earth.name = 'Earth'
    earth.mass = 5.9742 * 10**24
    earth.px = -1*AU
    earth.vy = 29.783 * 1000            # 29.783 km/sec
    earth.pencolor('blue')

    # Venus parameters taken from
    # http://nssdc.gsfc.nasa.gov/planetary/factsheet/venusfact.html
    venus = Body()
    venus.name = 'Venus'
    venus.mass = 4.8685 * 10**24
    venus.px = 0.723 * AU
    venus.vy = -35.02 * 1000
    venus.pencolor('red')

    loop([sun, earth, venus])

if __name__ == '__main__':
    main()
```

Code Discussion

The system described in the code consists of the Sun, Earth, and Venus, so the `main()` function creates three `Body` instances for each body and passed to the `loop()` function.

The `loop()` function is the heart of the simulation, taking a list of `Body` instances and then performing simulation steps forever. The time step chosen is one day, which works well for our Sun/Earth/Venus example. When you run the program, you can see how long it takes for the plot to complete an entire orbit; for Earth it's the expected 365 days and for Venus it's 224 days.

Lessons Learned

Each simulation step requires calculating $N * (N - 1)$ distances and attractions, so the time complexity is $O(N^2)$. On a laptop or desktop, the display will be visible changing up to around 20 objects. More efficient coding would let us handle more objects; we could rewrite the calculations in C or parallelize the code to divide the work in each step among multiple threads or CPUs. You could also adjust the timestep dynamically: if objects are far apart, a larger timestep would introduce less error, and the timestep could be shortened when objects are interacting more closely.

These techniques would increase our practical limit to hundreds (10^3) or thousands (10^4) of objects, but this means we can't simulate even a small galaxy, which might contain tens of millions of stars (10^7). (Our galaxy is estimated to have around 200 billion stars, 2×10^{11}.) Entirely different approaches need to be taken for that problem size; for example, the attraction of distant particles is approximated and only nearby particles are calculated exactly. The references include a survey by Drs. Trenti and Hut that describes the techniques used for larger simulations.

References

http://www.scholarpedia.org/article/N-body_simulations This survey, by Dr. Michele Trenti and Dr. Piet Hut, describes how the serious scientific N-body simulators work, using trees to approximate the attraction at great

distances. Such programs are able to run in $O(Nlog(N))$ time.

http://ssd.jpl.nasa.gov/horizons.cgi NASA's Jet Propulsion Laboratory provides a system called HORIZONS that returns accurate positions and velocities for objects within the solar system. In the example code, the values used are only rough approximations; the orbital distances and planet velocities are set to the mean distances and their relative positions don't correspond to any actual point in time – but they produce reasonable output.

CHAPTER 8

Problem: Simulating the Game of Life

In 1970, mathematician John H. Conway proposed a simulation that he called the Game of Life. Martin Gardner wrote an column about Life in the October 1970 issue of *Scientific American* that brought widespread attention to the Game of Life. It's not what most people think of as a game; there are no players and there's no way to win or lose the game. Instead, Life is more like a model or simulation in which you can play and experiment.

Life takes place on a two-dimensional grid of square cells. Each square cell can be either alive or dead (full or empty).

Fig. 8.1: Example of a Life board. Cell **x**'s eight neighbours are numbered.

The simulation is carried out at fixed time steps; every time step, all the cells on the grid can switch from dead to alive, or alive to dead, depending on four simple rules that only depend on a given cell's eight immediate neighbours. Let's take the cell **x** in the diagram, whose neighbours have been numbered 1 through 8 in the diagram.

If the cell is dead:

1. **Birth**: if exactly three of its neighbours are alive, the cell will become alive at the next step.

If the cell is already alive:

1. **Survival**: if the cell has two or three live neighbours, the cell remains alive.

Otherwise, the cell will die:

2. **Death by loneliness**: if the cell has only zero or one live neighbours, the cell will become dead at the next step.
3. **Death by overcrowding**: if the cell alive and has more than three live neighbours, the cell also dies.

These rules are simple and readily understandable, but they lead to surprisingly complex behaviour. (The general term for a simulation carried out on a grid of cells and following some simple rules is a **cellular automaton**.) For example, here are some patterns that occur after starting with five live cells in a row.

This pattern ends up in a cycle that repeats itself endlessly. Researchers in Life have coined many terms for different types of pattern, and this one is called an **oscillator**. *The wiki at ConwayLife.com <http://www.conwaylife.com/wiki/Main_Page>* describes many such terms. For example, oscillators go through a cycle of states and return to their initial state; **still life** patterns are stable and don't change over time at all; **spaceships** return to their initial configuration but in a different position, and therefore the pattern moves through the grid.

Approach

It's possible to work out Life patterns using pencil and paper, but obviously this is boring for large or long-lived patterns, and there's also the risk of error. Computer implementations of Life were written soon after Conway described it.

A basic implementation is straightforward: store the state of the board, and loop over every cell to determine its new state. To be correct, the code has to record new states in a copy of the board's data structure and not update the original board as it's scanned.

Solution

This implementation of Life uses the `turtle` graphics to draw the board. Keystrokes are used to control the program; hit 'R' to fill the board with a random pattern, and then 'S' to step through one generation at a time.

```
#!/usr/bin/env python3

# life.py -- A turtle-based version of Conway's Game of Life.
#
# An empty board will be displayed, and the following commands are available:
#   E : Erase the board
#   R : Fill the board randomly
#   S : Step for a single generation
#   C : Update continuously until a key is struck
#   Q : Quit
#   Cursor keys :  Move the cursor around the board
#   Space or Enter : Toggle the contents of the cursor's position
#

import sys
import turtle
import random

CELL_SIZE = 10                  # Measured in pixels

class LifeBoard:
    """Encapsulates a Life board

    Attributes:
    xsize, ysize : horizontal and vertical size of the board
    state : set containing (x,y) coordinates for live cells.

    Methods:
    display(update_board) -- Display the state of the board on-screen.
    erase() -- clear the entire board
    makeRandom() -- fill the board randomly
    set(x,y) -- set the given cell to Live; doesn't refresh the screen
    toggle(x,y) -- change the given cell from live to dead, or vice
                   versa, and refresh the screen display

    """
    def __init__(self, xsize, ysize):
        """Create a new LifeBoard instance.

        scr -- curses screen object to use for display
        char -- character used to render live cells (default: '*')
        """
        self.state = set()
        self.xsize, self.ysize = xsize, ysize

    def is_legal(self, x, y):
        "Returns true if the x,y coordinates are legal for this board."
        return (0 <= x < self.xsize) and (0 <= y < self.ysize)

    def set(self, x, y):
        """Set a cell to the live state."""
        if not self.is_legal(x, y):
            raise ValueError("Coordinates {}, {} out of range 0..{}, 0..{}".format(
                    x, y, self.xsize, self.ysize))

        key = (x, y)
        self.state.add(key)

    def makeRandom(self):
        "Fill the board with a random pattern"
        self.erase()
        for i in range(0, self.xsize):
            for j in range(0, self.ysize):
                if random.random() > 0.5:
                    self.set(i, j)

    def toggle(self, x, y):
        """Toggle a cell's state between live and dead."""
        if not self.is_legal(x, y):
            raise ValueError("Coordinates {}, {} out of range 0..{}, 0..{}".format(
                    x, y, self.xsize, self.ysize))
        key = (x, y)
        if key in self.state:
            self.state.remove(key)
        else:
            self.state.add(key)

    def erase(self):
        """Clear the entire board."""
        self.state.clear()

    def step(self):
        "Compute one generation, updating the display."
        d = set()
        for i in range(self.xsize):
            x_range = range( max(0, i-1), min(self.xsize, i+2) )
            for j in range(self.ysize):
                s = 0
                live = ((i,j) in self.state)
                for yp in range( max(0, j-1), min(self.ysize, j+2) ):
                    for xp in x_range:
                        if (xp, yp) in self.state:
                            s += 1

                # Subtract the central cell's value; it doesn't count.
                s -= live
                ##print(d)
                ##print(i, j, s, live)
                if s == 3:
                    # Birth
                    d.add((i,j))
                elif s == 2 and live:
                    # Survival
                    d.add((i,j))
                elif live:
                    # Death
                    pass

        self.state = d

    #
    # Display-related methods
    #
    def draw(self, x, y):
        "Update the cell (x,y) on the display."
        turtle.penup()
        key = (x, y)
        if key in self.state:
            turtle.setpos(x*CELL_SIZE, y*CELL_SIZE)
            turtle.color('black')
            turtle.pendown()
            turtle.setheading(0)
            turtle.begin_fill()
            for i in range(4):
                turtle.forward(CELL_SIZE-1)
                turtle.left(90)
            turtle.end_fill()

    def display(self):
        """Draw the whole board"""
        turtle.clear()
        for i in range(self.xsize):
            for j in range(self.ysize):
                self.draw(i, j)
        turtle.update()


def display_help_window():
    from turtle import TK
    root = TK.Tk()
    frame = TK.Frame()
    canvas = TK.Canvas(root, width=300, height=200, bg="white")
    canvas.pack()
    help_screen = turtle.TurtleScreen(canvas)
    help_t = turtle.RawTurtle(help_screen)
    help_t.penup()
    help_t.hideturtle()
    help_t.speed('fastest')

    width, height = help_screen.screensize()
    line_height = 20
    y = height // 2 - 30
    for s in ("Click on cells to make them alive or dead.",
              "Keyboard commands:",
              " E)rase the board",
              " R)andom fill",
              " S)tep once or",
              " C)ontinuously -- use 'S' to resume stepping",
              " Q)uit"):
        help_t.setpos(-(width / 2), y)
        help_t.write(s, font=('sans-serif', 14, 'normal'))
        y -= line_height


def main():
    display_help_window()

    scr = turtle.Screen()
    turtle.mode('standard')
    xsize, ysize = scr.screensize()
    turtle.setworldcoordinates(0, 0, xsize, ysize)

    turtle.hideturtle()
    turtle.speed('fastest')
    turtle.tracer(0, 0)
    turtle.penup()

    board = LifeBoard(xsize // CELL_SIZE, 1 + ysize // CELL_SIZE)

    # Set up mouse bindings
    def toggle(x, y):
        cell_x = x // CELL_SIZE
        cell_y = y // CELL_SIZE
        if board.is_legal(cell_x, cell_y):
            board.toggle(cell_x, cell_y)
            board.display()

    turtle.onscreenclick(turtle.listen)
    turtle.onscreenclick(toggle)

    board.makeRandom()
    board.display()

    # Set up key bindings
    def erase():
        board.erase()
        board.display()
    turtle.onkey(erase, 'e')

    def makeRandom():
        board.makeRandom()
        board.display()
    turtle.onkey(makeRandom, 'r')

    turtle.onkey(sys.exit, 'q')

    # Set up keys for performing generation steps, either one-at-a-time or not.
    continuous = False
    def step_once():
        nonlocal continuous
        continuous = False
        perform_step()

    def step_continuous():
        nonlocal continuous
        continuous = True
        perform_step()

    def perform_step():
        board.step()
        board.display()
        # In continuous mode, we set a timer to display another generation
        # after 25 millisenconds.
        if continuous:
            turtle.ontimer(perform_step, 25)

    turtle.onkey(step_once, 's')
    turtle.onkey(step_continuous, 'c')

    # Enter the Tk main loop
    turtle.listen()
    turtle.mainloop()

if __name__ == '__main__':
    main()
```

Code Discussion

The `LifeBoard` class has a `state` attribute that's a set of (X,Y) tuples that contain live cells. The coordinate values can vary between 0 and an upper limit specified by the `xsize` and `ysize` attributes.

The `step()` method computes a single Life generation. It loops over the entire board, and for each cell the code counts the number of live cells surrounding it. A new set is used to record the cells that are live in the new generation, and after the whole board has been scanned, the new set replaces the existing `state`.

The size of the `state` set is therefore proportional to the number of live cells at any given time. Another approach would be just to have an N x N array representing the board, which would require a fixed amount of memory.

If there are only a few live cells on a large board, most of the time will be spent scanning empty areas of the board where we know nothing is going to happen. Cells never come alive spontaneously, without any live neighbours. Therefore, one minor optimization is to record the minimum and maximum coordinates of live cells, and then limit the scanning accordingly. An entirely different approach called Hashlife represents the board as a quadtree, a 2-dimensional tree structure, and relies on large Life patterns often containing many copies of similar structures. (See the references for an explanation of Hashlife.)

Lessons Learned

Part of what makes Life so fascinating is that it's easy to ask questions and then try to answer them. For example, if you start with a straight line of N cells, which values of N produce interesting patterns? What if you make squares, or two lines of cells? This leads to asking more theoretical questions: are there Life patterns that can make copies of themselves? How fast can a spaceship move? You can get an idea of the complexity hiding inside Life by exploring some of the references for this section.

You may wonder if there's anything special about the rules that Conway chose for Life. For example, what if live cells survived when they had four neighbours instead of dying? You can easily experiment with different rules by modifying the `life.py` program. Mirek Wojtowicz has written a list of alternate rules at http://www.mirekw.com/ca/rullex_life.html and comments on the different properties of the resulting simulations.

People have also created many other cellular automata that change other aspects such as:

- having the world be a 3-dimensional grid of cubes or a 1-dimensional line of cells instead of a 2-dimensional grid.
- use a triangular or hexagonal grid instead of a square one.
- have multiple colours for live cells; a newborn cell's can either be the majority colour of its neighbours, or
- have the cells containing decimal values between 0 and 1, instead of restricting values to only 0 and 1.

Fairly soon after Life was invented, Conway proved that there existed Life patterns that were equivalent to a Turing Machine and therefore could carry out any computable function. In April 2000, Paul Rendell actually constructed a Life pattern that behaved like a Turing machine.

References

Relevant web pages, books, with an annotation about why it's notable or worthwhile.

http://www.math.com/students/wonders/life/life.html An introductory article by Paul Callahan. The web page includes a Java applet that's similar to our Python application.

http://www.nytimes.com/1993/10/12/science/scientist-at-work-john-h-conway-at-home-in-the-elusive-world-of-mathematics.ht A *New York Times* profile of mathematician John H. Conway, who invented Life.

http://www.ibiblio.org/lifepatterns/october1970.html A copy of Martin Gardner's 1970 *Scientific American* article that started it all.

http://home.interserv.com/~mniemiec/lifeterm.htm A glossary of Life terminology.

http://www.conwaylife.com/wiki/ LifeWiki contains over a thousand Wiki articles about Life, including definitions of terms and descriptions of various patterns.

http://www.drdobbs.com/jvm/an-algorithm-for-compressing-space-and-t/184406478 An April 2006 article by Tomas G. Rokicki that explains the HashLife algorithm and implements it in Java.

CHAPTER 9

Reference Card: Turtle Graphics

XXX write a reference card of methods

- display a canvas on which turtles run around.
- turtles have various methods: (pick a fixed subset)
- forward/backward/right/left/speed
- pos()/ycor/xcor/heading
- pen/penup/pendown
- hideturtle/showturtle
- home/setpos/setheading
- dot/stamp/clearstamp/reset

CHAPTER 10

Indices and tables

- genindex
- search