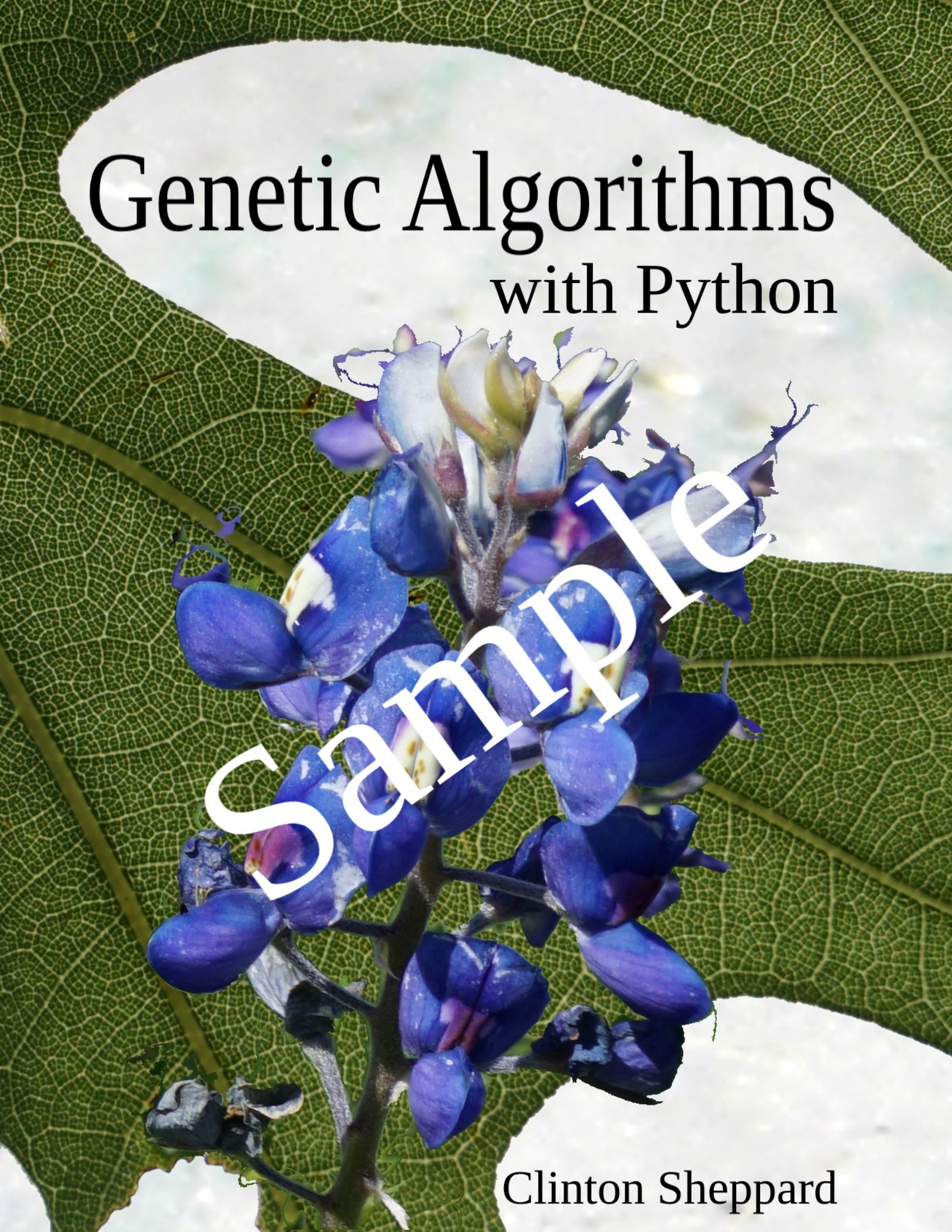
Genetic Algorithms with Python



Clinton Sheppard

Genetic Algorithms with Python

Clinton Sheppard

2017-04-29

First Printing: 2016

```
Clinton Sheppard <fluentcoder@gmail.com>
Austin, Texas, USA
www.cs.unm.edu/~sheppard
```

The final code from each chapter is available at https://github.com/handcraftsman/GeneticAlgorithmsWithPython, licensed under the Apache License, Version 2.0.

The text of this book was written in AsciiDoc using Atom and AsciiDoc-Preview for editing and converted to PDF using AsciiDoctor 1.5.5. The code was written and tested using JetBrains' PyCharm IDE for Python. Some images were produced using GraphViz and Paint.Net. The fonts are Liberation Serif v2.00.1, M+ MN Type-1, and FontAwesome icons included by AsciiDoc.

Preface

Genetic Algorithms with Python distills more than 5 years of experience using genetic algorithms and helping others learn how to apply genetic algorithms, into a graduated series of lessons that will impart to you a powerful life-long skill. To get the most out of this book you need to do more than simply read along, you need to train your mind by experimenting with the code. That means typing in the code changes as they are introduced so you can try different things at each checkpoint before moving on to the next. Enjoy!

Clinton Sheppard <fluentcoder@gmail.com>

A brief introduction to genetic algorithms

Genetic algorithms are one of the tools we can use to apply machine learning to finding good, sometimes even optimal, solutions to problems that have billions of potential solutions. They use biological processes in software to find answers to problems that have really large search spaces by continuously generating candidate solutions, evaluating how well the solutions fit the desired outcome, and refining the best solutions.

When solving a problem with a genetic algorithm, instead of asking for a specific solution, you provide characteristics that the solution must have or rules its solution must pass to be accepted. For example, when filling a moving truck you provide a set of rules like: load big things first, distribute the weight to both sides, put light things on top but not loose, interlock things with odd shapes so they don't move around. The more constraints you add the more potential solutions are blocked. Suppose you say: put the refrigerator, washer and dryer along the front left wall, load boxes of books along the front right wall, mattresses down the middle, and clothes on top of the mattresses. These more specific rules do not work if you are loading packages, or Christmas trees, but the previous goal oriented ones still do.

Goal oriented problem solving

Imagine you are given 10 chances to guess a number between 1 and 1000 and the only feedback you get is whether your guess is right or wrong. Could you reliably guess the number? With only *right* or *wrong* as feedback, you have no way to improve your guesses so you have at best a 1 in 100 chance of guessing the number. A fundamental aspect of solving problems using genetic algorithms is that they must provide feedback that helps the engine select the better of two guesses. That feedback is called the fitness, for how closely the guess fits the desired result. More importantly it implies a general progression.

If instead of *right* or *wrong* as feedback you receive *higher* or *lower* indicating that the number is higher or lower than your guess, you can always find the number because 10 guesses are sufficient to binary search your way to any number in the 1 to 1000 range.

Now imagine multiplying the size of this problem so that instead of trying to find 1 number you are simultaneously trying to find a set of 100 numbers, all in the range 1 to 1000, your only receive back a fitness value indicating how closely that set of numbers matches the desired outcome. Your goal would be to maximize or minimize that fitness. Could you find the right set of 100 numbers? You might be able to do better than random guessing if you have problem-specific knowledge that helps you eliminate certain number combinations. Using problem-specific knowledge to guide the genetic algorithm's creation and modification of potential solutions can help them find a solution orders of magnitude faster.

Genetic algorithms and genetic programming are very good at finding solutions to very large problems. They do it by taking millions of samples from the search space, making small changes, possibly recombining parts of the best solutions, comparing the resultant fitness against that of the current best solution, and keeping the better of the two. This process repeats until a stop condition like one of the following occurs: the known solution is found, a solution meeting all requirements is found, a certain number of generations has passed, a specific amount of time has passed, etc.

First project

Imagine you are asked to guess a 3-letter password; what kind of feedback would you want? If the password is *'aaa*' and you guess *'abc*' what should the fitness value be? Would something simple like how many of the letters in your guess are correct be sufficient? Should *'bab*', which has one correct letter, get a better fitness value than *'zap*', also one correct letter but the wrong letters are alphabetically farther away, or should the fitness be the same? These are some of the first decisions you have to make when planning to implement a genetic algorithm to find a solution to your problem. Genetic algorithms are good at finding good solutions to problems with large search spaces because they can quickly find the parts of the guesses that improve fitness values or lead to better solutions.

In the project above, let's say the fitness function returns a count of the number of letters that match the password. That means *'abc*', *'bab*' and *'zba*' all get a fitness value of one, because they each have one letter correct. The genetic algorithm might then combine the first two letters of *'abc*' with the last letter of *'zba*' through crossover, to create the guess *'aba*'. The

fitness for that guess would be two because two letters match the password. The algorithm might also mutate the last letter of *'zba'* to get *'zbc'* and a fitness value of zero. Depending on how the engine is configured it might throw out *'zbc'* immediately, it might keep it in order to maintain genetic diversity, or perhaps it would only keep it if it is better than some cutoff fitness value when compared with all the other guesses tried by the engine.

We will look more at the password project in the first chapter, and go on to explore a variety of projects to learn different ways of solving problems with genetic algorithms. However, this book is not about showing you a hand-picked set of problems you can solve with genetic algorithms. It is about giving you experience making genetic algorithms work for you using sample projects that you understand and can fall back upon when learning to use other machine learning tools and techniques, or applying genetic algorithms in your own field of expertise.

Genetic programming with Python

This book uses the Python programming language to explore genetic algorithms. Why Python? Because Python is a low ceremony, powerful and easy-to-read language whose code can be understood by entry-level programmers. I explain the occasional Python feature but should you encounter a programming construct you've never seen before and can't intuit, Python.org and StackOverflow.com are great places to find explanations. If you have experience with another programming language then you should have no difficulty learning Python by induction while also exploring genetic algorithms.

example Python syntax

```
# this is a comment
import math  # imports make code from other modules available

# code blocks are initiated by a trailing colon followed by indented lines
class Circle:                     # define a class
    def __init__(self, radius):   # constructor with parameter radius
        self.radius = radius      # store the parameter in a class variable

    def get_area(self):           # define a function that belongs to the class
        return math.pi \
               * self.radius \
               * self.radius      # trailing \ continues the expression on the next line

# code that is not in a class is executed immediately
for i in range(1, 10):
    if (i & 1) == 0:
        continue
    circle = Circle(i)            # create an instance
    print("A circle with radius {0} has area {1:0.2f}".format(
        i, circle.get_area()      # `print` writes output to the console
    ))
```

You can run the code above in your browser at https://repl.it/C0uK.

Like blacksmiths, programmers create their own tools. We frequently prototype a solution by using tools we already have available, not unlike using a pair of pliers to pull a nail. Once we get a good understanding of the problem, however, we usually restart with a better combination of tools or build a problem-specific one. In this book we will co-evolve a genetic engine while examining increasingly difficult projects with the engine. Why not just use one of the genetic programming packages already available for Python like Pyvolution, DEAP, Pyevolve, pySTEP, etc? Because they all have different

interfaces and options, some of which may not be applicable to a problem, and we're trying to learn about genetic algorithms not specific engines. By co-evolving the engine you'll know exactly how it works so you'll be able to use its features effectively to solve the next problem with a genetic algorithm of your own design. The engine will be a by-product of applying genetic algorithms to the different projects in this book. If you were to co-evolve an engine with a different set of projects, or even the projects in this book in a different order, you would end up with a different engine. But, by co-evolving an engine you will gain experience with some of the features available in commonly used packages, and see how they can affect the performance of your code.

About the author

I am a polyglot programmer with more than 15 years of professional programming experience. Occasionally I step out of my comfort zone and learn a new language to see what that development experience is like and to keep my skills sharp. This book grew out of my experiences while learning Python, but it isn't about Python.

When learning a new programming language, I start with a familiar project and try to learn enough of the new language to solve it. For me, writing a genetic engine is that familiar project. Why a genetic engine? For one thing, it is a project where I can explore interesting puzzles, and where even a child's game like Tic-Tac-Toe can be viewed on a whole new level. Also, I can select increasingly complex puzzles to drive evolution in the capabilities of the engine. This allows me to discover the expressiveness of the language, the power of its tool chain, and the size of its development community as I work through the idiosyncrasies of the language.

About the text

The Python 3.5 code snippets in this book were programmatically extracted from **working code files** using the tags feature of AsciiDoc's include directive.

Chapter 1. Hello World!

1.1. Guess my number

Let's begin by learning a little bit about genetic algorithms. Reach way back in your memories to a game we played as kids. It is a simple game for two people where one picks a secret number between 1 and 10 and the other has to guess that number.

```
Is it 2?  No
Is it 3?  No
Is it 7?  No
Is it 1?  Yes
```

That works reasonably well for 1..10 but quickly becomes frustrating or boring as we increase the range to 1..100 or 1..1000. Why? Because we have no way to improve our guesses. There's no challenge. The guess is either right or wrong, so it quickly becomes a mechanical process.

```
Is it 1?  No
Is it 2?  No
Is it 3?  No
Is it 4?  No
Is it 5?  No
...
```

So, to make it more interesting, instead of *no* let's say *higher* or *lower*.

```
1?  Higher
7?  Lower
6?  Lower
5?  Lower
4?  Correct
```

That might be reasonably interesting for a while for 1..10 but soon you'll increase the range to 1..100. Because people are competitive, the next revision is to see who is a better guesser by trying to find the number in the fewest guesses. At this point the person who evolves the most efficient guessing strategy wins.

However, one thing we automatically do when playing the game is make use of domain knowledge. For example, after this sequence:

```
1?  Higher
7?  Lower
```

Why wouldn't we guess 8, 9, or 10? The reason is, of course, because we know that those numbers are not *lower* than 7. Why wouldn't we guess 1? Because we already tried it. We use our memory of what we've tried, our successes and failures, and our *knowledge of the domain*, number relationships, to improve our guesses.

When playing a card game inexperienced players build a mental map using the cards in their hand and those on the table. More experienced players also take advantage of their knowledge of the problem space, the entire set of cards in the deck. This means they may also keep track of cards that have not yet been played, and may know they can win the rest of the rounds without having to play them out. Highly experienced card players also know the probabilities of various winning combinations. Professionals, who earn their living playing the game, also pay attention to the way their competitors play… whether they bluff in certain situations, play with their chips when they think they have a good hand, etc.

A genetic algorithm does not know what *lower* means. It has no intelligence. It does not learn. It will make the same mistakes every time. It will only be as good at solving a problem as the person who writes the code. And yet, it can be used to find solutions to problems that humans would struggle to solve or could not solve at all. How is that possible?

Genetic algorithms use random exploration of the problem space combined with evolutionary processes like mutation and crossover (exchange of genetic information) to improve guesses. But also, because they have no experience in the problem domain, they *try things a human would never think to try*. Thus, a person using a genetic algorithm may learn more about the problem space and potential solutions. This gives them the ability to make improvements to the algorithm, in a virtuous cycle.

What can we learn from this?

Technique

The genetic algorithm should make informed guesses.

1.2. Guess the Password

Now let's see how this applies to guessing a password. Start with a randomly generated initial sequence of letters, then mutate/change one random letter at a time until the sequence of letters is "Hello World!". Conceptually:

pseudo code

```
_letters = [a..zA..Z !]
target = "Hello World!"
guess = get 12 random letters from _letters
while guess != target:
    index = get random value from [0..length of target]
    guess[index] = get 1 random value from _letters
```

If you try this in your favorite programming language you'll find that it performs worse than playing the number guessing game with only *yes* and *no* answers because it cannot tell when one guess is better than another.

One solution is to help it make an informed guess by telling it how many of the letters from the guess are in the correct locations. For example "World!Hello?" would get 2 because only the 4th letter of each word is correct. The 2 indicates how close the answer is to correct. This is called the fitness value. "hello world?" would get a fitness value of 9 because 9 letters are correct. Only the h, w, and question mark are wrong.

1.3. First Program

It is time to start writing Python. By the way, if you do not already have a favorite Python development environment, I highly recommend JetBrains' PyCharm IDE.

1.3.1. Genes

To begin with, the genetic algorithm needs a gene set to use for building guesses. For this project that will be a generic set of letters. It also needs a target password to guess:

guessPassword.py

```
geneSet = " abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!."
target = "Hello World!"
```

You can run the code for this section in your browser at https://repl.it/CZL1/1

1.3.2. Generate a guess

Next the algorithm needs a way to generate a random string from the gene set.

```
import random

def generate_parent(length):
    genes = []
    while len(genes) < length:
        sampleSize = min(length - len(genes), len(geneSet))
        genes.extend(random.sample(geneSet, sampleSize))
    return ''.join(genes)
```

`random.sample` takes `sampleSize` values from the input without replacement. This means there will be no duplicates in the generated parent unless `geneSet` contains duplicates, or `length` is greater than `len(geneSet)`. The implementation above can generate a long string with a small set of genes and uses as many unique genes as possible.

1.3.3. Fitness

The `fitness` value the genetic algorithm provides is the **only** feedback the engine gets to guide it toward a solution. In this project the `fitness` value is the total number of letters in the guess that match the letter in the same position of the password.

```
def get_fitness(guess):
    return sum(1 for expected, actual in zip(target, guess)
               if expected == actual)
```

1.3.4. Mutate

Next, the engine needs a way to produce a new guess by mutating the current one. The following implementation converts the parent string to an array with `list(parent)`, then replaces 1 letter in the array with a randomly selected one from `geneSet`, and finally recombines the result into a string with `''.join(childGenes)`.

```
def mutate(parent):
    index = random.randrange(0, len(parent))
    childGenes = list(parent)
    newGene, alternate = random.sample(geneSet, 2)
    childGenes[index] = alternate \
        if newGene == childGenes[index] \
        else newGene
    return ''.join(childGenes)
```

This implementation uses an alternate replacement if the randomly selected `newGene` is the same as the one it is supposed to replace, which can prevent a significant number of wasted guesses.

1.3.5. Display

Next, it is important to monitor what is happening so that the engine can be stopped if it gets stuck. Having a visual representation of the gene sequence, which may not be the literal gene sequence, is often critical to identifying what works and what does not so that the algorithm can be improved.

Normally the display function also outputs the fitness value and how much time has elapsed.

```
import datetime
...
def display(guess):
    timeDiff = datetime.datetime.now() - startTime
    fitness = get_fitness(guess)
    print("{}\t{}\t{}".format(guess, fitness, timeDiff))
```

1.3.6. Main

The main program begins by initializing `bestParent` to a random sequence of letters and calling the display function.

```
random.seed()
startTime = datetime.datetime.now()
bestParent = generate_parent(len(target))
bestFitness = get_fitness(bestParent)
display(bestParent)
```

The final piece is the heart of the genetic engine. It is a loop that:

- generates a guess,
- requests the `fitness` for that guess, then
- compares the `fitness` to that of the previous best guess, and
- keeps the guess with the better fitness.

This cycle repeats until a stop condition occurs, in this case when all the letters in the guess match those in the target.

```
while True:
    child = mutate(bestParent)
    childFitness = get_fitness(child)
    if bestFitness >= childFitness:
        continue
    display(child)
    if childFitness >= len(bestParent):
        break
    bestFitness = childFitness
    bestParent = child
```

Run the code and you'll see output similar to the following. Success!

```
ftljCDPvhasn    1   0:00:00
ftljC Pvhasn    2   0:00:00
ftljC Pohasn    3   0:00:00.001000
HtljC Pohasn    4   0:00:00.002000
HtljC Wohasn    5   0:00:00.004000
Htljo Wohasn    6   0:00:00.005000
Htljo Wohas!    7   0:00:00.008000
Htljo Wohls!    8   0:00:00.010000
Heljo Wohls!    9   0:00:00.013000
Hello Wohls!    10  0:00:00.013000
Hello Wohld!    11  0:00:00.013000
Hello World!    12  0:00:00.015000
```

1.4. Extract a reusable engine

We have a working engine but it is currently tightly coupled to the Password project, so the next task is to extract the genetic engine code from that specific to guessing the password so it can be reused for other projects. Start by creating a new file named `genetic.py`.

Next move the `mutate` and `generate_parent` functions to the new file and rename them to `_mutate` and `_generate_parent`. This is how protected functions are named in Python. Protected functions are only accessible to other functions in the same module.

1.4.1. Generate and Mutate

Future projects will need to be able to customize the gene set, so that needs to become a parameter to `_generate_parent` and `_mutate`.

```
import random

def _generate_parent(length, geneSet):
    genes = []
    while len(genes) < length:
        sampleSize = min(length - len(genes), len(geneSet))
        genes.extend(random.sample(geneSet, sampleSize))
    return ''.join(genes)
```

```
def _mutate(parent, geneSet):
    index = random.randrange(0, len(parent))
    childGenes = list(parent)
    newGene, alternate = random.sample(geneSet, 2)
    childGenes[index] = alternate \
        if newGene == childGenes[index] \
        else newGene
    return ''.join(childGenes)
```

1.4.2. `get_best`

The next step is to move the main loop into a new public function named `get_best` in the `genetic` module. Its parameters are:

- the function it calls to request the fitness for a guess,
- the number of genes to use when creating a new gene sequence,
- the optimal fitness value,
- the set of genes to use for creating and mutating gene sequences, and
- the function it should call to display, or report, each improvement found.

```
def get_best(get_fitness, targetLen, optimalFitness, geneSet, display):
    random.seed()
    bestParent = _generate_parent(targetLen, geneSet)
    bestFitness = get_fitness(bestParent)
    display(bestParent)
    if bestFitness >= optimalFitness:
        return bestParent

    while True:
        child = _mutate(bestParent, geneSet)
        childFitness = get_fitness(child)

        if bestFitness >= childFitness:
            continue
        display(child)
        if childFitness >= optimalFitness:
            return child
        bestFitness = childFitness
        bestParent = child
```

Notice that `display` and `get_fitness` are called with only one parameter - the child gene sequence. This is because a generic engine does not need access to the target value, and it does not care about how much time has passed, so those are not passed to it.

The result is a reusable module named `genetic` that can be used in other programs via `import genetic`.

1.4.3. Use the `genetic` module

The code remaining in `guessPassword.py` is specific to the password guessing project. To get it working again first import the `genetic` module.

guessPassword.py

```
import datetime
import genetic
```

Next, helper functions that take only one parameter must be defined so they are compatible with what the `genetic` engine expects. Each helper function will take the candidate gene sequence it receives and call the local functions with additional required parameters as necessary.

```
def test_Hello_World():
    target = "Hello World!"
    guess_password(target)


def guess_password(target):
    geneset = " abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!."
    startTime = datetime.datetime.now()

    def fnGetFitness(genes):                ①
        return get_fitness(genes, target)

    def fnDisplay(genes):                   ②
        display(genes, target, startTime)

    optimalFitness = len(target)
    genetic.get_best(fnGetFitness, len(target), optimalFitness, geneset, fnDisplay)
```

1.4.4. Display

Now change `display` to take the target password as a parameter. It could remain a global variable in the algorithm file but this change facilitates trying different passwords without side effects.

```
def display(genes, target, startTime):
    timeDiff = datetime.datetime.now() - startTime
    fitness = get_fitness(genes, target)
    print("{}\t{}\t{}".format(genes, fitness, timeDiff))
```

1.4.5. Fitness

The fitness function also needs to receive the target password as a parameter.

```
def get_fitness(genes, target):
    return sum(1 for expected, actual in zip(target, genes)
               if expected == actual)
```

1.4.6. Main

There are many ways to structure the main code, the most flexible is as a unit test. To start that transition first rename `guessPassword.py` to `guessPasswordTests.py`. Next, to make it possible to execute the code from the command line add:

guessPasswordTests.py

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

If you are following along in an editor be sure to run your code to verify it works at this point.

1.5. Use Python's `unittest` framework

The next step is to make the code work with Python's built in test framework.

```
import unittest
```

To do that the main test function must be moved into a class that inherits from `unittest.TestCase`. The other functions can be moved into the class as well if you want, but if they are then `self` must be added as the first parameter to each because they will then belong to the test class.

```
class GuessPasswordTests(unittest.TestCase):
    geneset = " abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!.,"

    def test_Hello_World(self):
        target = "Hello World!"
        self.guess_password(target)

    def guess_password(self, target):
...
        optimalFitness = len(target)
        best = genetic.get_best(fnGetFitness, len(target),
                                optimalFitness, self.geneset,
                                fnDisplay)
        self.assertEqual(best, target)
```

When the `unittest` module's `main` function is called, it automatically executes each function whose name starts with "test".

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

This allows the test to be run from the command line and, incidentally, without its `display` function output.

```
python -m unittest -b guessPasswordTests
.
----------------------------------------
Ran 1 test in 0.020s

OK
```

If you get an error like `'module' object has no attribute 'py'` then you used the filename `guessPasswordTests.py` instead of the module name `guessPasswordTests`.

1.6. A longer password

"Hello World!" doesn't sufficiently demonstrate the power of the genetic engine so try a longer password:

```
def test_For_I_am_fearfully_and_wonderfully_made(self):
    target = "For I am fearfully and wonderfully made."
    self.guess_password(target)
```

1.6.1. Run

```
...
ForMI am feabaully and wWndNyfulll made.    33  0:00:00.047094
For I am feabaully and wWndNyfulll made.    34  0:00:00.047094
For I am feabfully and wWndNyfulll made.    35  0:00:00.053111
For I am feabfully and wondNyfulll made.    36  0:00:00.064140
For I am feabfully and wondNyfully made.    37  0:00:00.067148
For I am feabfully and wondeyfully made.    38  0:00:00.095228
For I am feabfully and wonderfully made.    39  0:00:00.100236
For I am fearfully and wonderfully made.    40  0:00:00.195524
```

Nice!

1.7. Introduce a `Chromosome` object

The next change is to introduce a `Chromosome` object that has `Genes` and `Fitness` attributes. This will make the genetic engine more flexible by making it possible to pass those values around as a unit.

genetic.py

```
class Chromosome:
    def __init__(self, genes, fitness):
        self.Genes = genes
        self.Fitness = fitness
```

```
def _mutate(parent, geneSet, get_fitness):
    index = random.randrange(0, len(parent.Genes))
    childGenes = list(parent.Genes)
...
    genes = ''.join(childGenes)
    fitness = get_fitness(genes)
    return Chromosome(genes, fitness)
```

```
def _generate_parent(length, geneSet, get_fitness):
...
    genes = ''.join(genes)
    fitness = get_fitness(genes)
    return Chromosome(genes, fitness)
```

```
def get_best(get_fitness, targetLen, optimalFitness, geneSet, display):
    random.seed()
    bestParent = _generate_parent(targetLen, geneSet, get_fitness)
    display(bestParent)
    if bestParent.Fitness >= optimalFitness:
        return bestParent

    while True:
        child = _mutate(bestParent, geneSet, get_fitness)

        if bestParent.Fitness >= child.Fitness:
            continue
        display(child)
        if child.Fitness >= optimalFitness:
            return child
        bestParent = child
```

It does require compensating changes to the algorithm file functions but those changes remove some double work.

guessPasswordTests.py

```
def display(candidate, startTime):
    timeDiff = datetime.datetime.now() - startTime
    print("{}\t{}\t{}".format(
        candidate.Genes, candidate.Fitness, timeDiff))
```

```
class GuessPasswordTests(unittest.TestCase):
...
    def guess_password(self, target):
...
        def fnDisplay(candidate):
            display(candidate, startTime)    ①

        optimalFitness = len(target)
        best = genetic.get_best(fnGetFitness, len(target),
                                optimalFitness, self.geneset,
                                fnDisplay)
        self.assertEqual(best.Genes, target) ②
```

1.8. Benchmarking

The next improvement is to add support for benchmarking to `genetic` because it is useful to know how long the engine takes to find a solution on average and the standard deviation. That can be done with another class as follows:

genetic.py

```
class Benchmark:
    @staticmethod
    def run(function):
        timings = []
        for i in range(100):
            startTime = time.time()
            function()
            seconds = time.time() - startTime
            timings.append(seconds)
            mean = statistics.mean(timings)
            print("{} {:3.2f} {:3.2f}".format(
                1 + i, mean,
                statistics.stdev(timings, mean)
                if i > 1 else 0))
```

That requires the following imports:

genetic.py

```
import statistics
import time
```

You may need to install the `statistics` module on your system. This can be accomplished from the command line with `python -m pip install statistics`

Now, to use the benchmarking capability simply add a test and pass the function to be benchmarked.

guessPasswordTests.py

```
def test_benchmark(self):
    genetic.Benchmark.run(self.test_For_I_am_fearfully_and_wonderfully_made)
```

When run, this function works great but is a bit chatty because it also shows the `display` output for all 100 runs. That can be fixed in the `run` function by temporarily redirecting standard output to nowhere while running the function being benchmarked.

genetic.py

```
import sys
...
class Benchmark:
    @staticmethod
    def run(function):
...
        timings = []
        stdout = sys.stdout          ①
        for i in range(100):
            sys.stdout = None        ②
            startTime = time.time()
            function()
            seconds = time.time() - startTime
            sys.stdout = stdout      ③
            timings.append(seconds)
...
```

If you get an error like the following when you run the benchmark test:

```
AttributeError: 'NoneType' object has no attribute 'write'
```

Then you are probably using Python 2.7. It does not support redirecting `stdout` to `None`. It must have someplace to write the data. One solution to that problem is to add the following class:

genetic.py

```
class NullWriter():
    def write(self, s):
        pass
```

Then replace the following in `Benchmark.run`:

```
for i in range(100):
    sys.stdout = None
```

with:

```
for i in range(100):
    sys.stdout = NullWriter()
```

That change allows you work around the difference between Python 2.7 and 3.5 this time. However, the code in this book uses other features of Python 3.5 so I suggest switching to Python 3.5 so that you can focus on learning about genetic algorithms without these additional issues. If you want to use machine learning tools that are tied to Python 2.7, wait until you have a solid understanding of genetic algorithms then switch to Python 2.7.

The output can also be improved by only displaying statistics for the first ten runs and then every 10th run after that.

genetic.py

```
...
            timings.append(seconds)
            mean = statistics.mean(timings)
            if i < 10 or i % 10 == 9:
                print("{} {:3.2f} {:3.2f}".format(
                    1 + i, mean,
                    statistics.stdev(timings, mean)
                    if i > 1 else 0))
```

Now the benchmark test output looks like the following.

sample output

```
1 0.19 0.00
2 0.17 0.00
3 0.18 0.02
...
9 0.17 0.03
10 0.17 0.03
20 0.18 0.04
...
90 0.16 0.05
100 0.16 0.05
```

This means that, averaging 100 runs, it takes .16 seconds to guess the password, and 68 percent of the time (one standard deviation) it takes between .11 (.16 - .05) and .21 (.16 + .05) seconds. Unfortunately that is probably too fast to determine if a change is due to a code improvement or due to something else running on the computer at the same time. That problem can be solved by making the genetic algorithm guess a random sequence that takes 1-2 seconds to run. Your processor likely is different from mine so adjust the length as necessary.

guessPasswordTests.py

```
import random
...
    def test_Random(self):
        length = 150
        target = ''.join(random.choice(self.geneset) for _ in range(length))
        self.guess_password(target)

    def test_benchmark(self):
        genetic.Benchmark.run(self.test_Random)
```

On my system that results in:

Benchmarks

average (seconds)	standard deviation
1.46	0.35

1.9. Summary

In this chapter we built a simple genetic engine that makes use of random mutation to produce better results. This engine was able to guess a secret password given only its length, a set of characters that might be in the password, and a fitness function that returns a count of the number characters in the guess that match the secret. This is a good benchmark project for the engine because as the target string gets longer the engine wastes more and more guesses trying to change positions that are already correct. As the engine evolves in later projects, we'll try to keep this benchmark fast. Also, as you work your way through this book you'll learn ways to improve the performance of the code in this project.

1.10. Final Code

The final code for this chapter is available from:

https://github.com/handcraftsman/GeneticAlgorithmsWithPython/tree/master/ch01

Problems?

Did you encounter a problem with this chapter? Please let me know at fluentcoder@gmail.com so I can try to help.

Chapter 2. One Max Problem

Our second project involves maximizing the number of 1's in an array of 100 1-bit numbers. For this project the gene set will be only 0 or 1.

2.1. Test class

oneMaxTests.py

```
import unittest
import datetime
import genetic

class OneMaxTests(unittest.TestCase):
    def test(self, length=100):
        geneset = [0, 1]
```

2.2. Change `genetic` to work with lists

The `genetic` module is currently hard coded to work with strings, so we need to modify it to work with lists instead. We can start by using Python's array slicing feature to copy the genes in `_mutate` instead of using the list constructor.

```
def _mutate(parent, geneSet, get_fitness):
    childGenes = parent.Genes[:]
...
```

Then remove the following line from `_mutate` and `_generate_parent` since we no longer need to convert the list back to a string:

```
genes = ''.join(childGenes)
```

Next, update those functions to use the list, as follows:

genetic.py

```
def _mutate(parent, geneSet, get_fitness):
    childGenes = parent.Genes[:]
    index = random.randrange(0, len(parent.Genes))
    newGene, alternate = random.sample(geneSet, 2)
    childGenes[index] = alternate \        ①
        if newGene == childGenes[index] \  ②
        else newGene
    fitness = get_fitness(childGenes)      ③
    return Chromosome(childGenes, fitness)    ④
```

```
def _generate_parent(length, geneSet, get_fitness):
...
    fitness = get_fitness(genes)      ①
    return Chromosome(genes, fitness) ②
```

To keep `guessPasswordTests.py` working, we must recombine the genes into a string in its `display` function

guessPasswordTests.py

```
def display(candidate, startTime):
    timeDiff = datetime.datetime.now() - startTime
    print("{}\t{}\t{}".format(
        ''.join(candidate.Genes),     ①
        candidate.Fitness,
        timeDiff))
```

and the assertion in `guess_password`.

```
class GuessPasswordTests(unittest.TestCase):
    def guess_password(self, target):
...
        self.assertEqual(''.join(best.Genes), target)
```

2.3. Genes

OK, back to the One Max problem. The fitness will be the number of 1's in the array.

oneMaxTests.py

```
def get_fitness(genes):
    return genes.count(1)
```

2.4. Display

Since 100 numbers would be a lot to display, we'll just show the first and last 15 along with the fitness and elapsed time.

```
def display(candidate, startTime):
    timeDiff = datetime.datetime.now() - startTime
    print("{}...{}\t{:3.2f}\t{}".format(
          ''.join(map(str, candidate.Genes[:15])),
           ''.join(map(str, candidate.Genes[-15:])),
          candidate.Fitness,
          timeDiff))
```

This uses `str` to convert the integers in `candidate.Genes` to strings. Without `map` the code would need a loop like the following to convert the candidate genes to strings:

```
result = []
for i in candidate.Genes
  result += str(candidate[i])
return result
```

2.5. Test

And here's the full test harness.

```
def test(self, length=100):
    geneset = [0, 1]
    startTime = datetime.datetime.now()

    def fnDisplay(candidate):
        display(candidate, startTime)

    def fnGetFitness(genes):
        return get_fitness(genes)

    optimalFitness = length
    best = genetic.get_best(fnGetFitness, length, optimalFitness,
                            geneset, fnDisplay)
    self.assertEqual(best.Fitness, optimalFitness)
```

2.6. Run

Now it can find the solution very quickly.

sample output

```
010101101010010...101010110100101   50.00   0:00:00.001000
010101101010010...101010110100101   51.00   0:00:00.001000
...
110111101111111...111111110111111   95.00   0:00:00.008000
110111111111111...111111110111111   96.00   0:00:00.008000
110111111111111...111111110111111   97.00   0:00:00.009000
110111111111111...111111111111111   98.00   0:00:00.009000
111111111111111...111111111111111   99.00   0:00:00.010000
111111111111111...111111111111111   100.00  0:00:00.011000
```

2.7. Benchmarks

Since it runs so fast we'll benchmark this project with a longer array. As with the Guess Password benchmark I'm choosing an array length that takes between 1 and 2 seconds on average on my box. You may want to select a different length.

```
def test_benchmark(self):
    genetic.Benchmark.run(lambda: self.test(4000))
```

We can see in the updated benchmarks that eliminating the string conversion may also have given us a tiny performance improvement in Guess Password.

Updated Benchmarks

project	average (seconds)	standard deviation
Guess Password	1.21	0.25
One Max	1.25	0.17

2.8. Aside

In this project the genetic engine randomly chooses an array index to change, even if the array already contains a 1 at that index, because the engine doesn't know the change it is making is useless until after it has called the fitness function. A physical equivalent is to take 100 coins and put green sticky dots on one side and yellow sticky dots on the other. Then have a partner blindfold you and drop the coins on a table top. Your goal is to turn all the coins yellow-side up. If you turn one yellow-side up they tell you it was a success. Otherwise, they undo the change and tell you it was a failure. To keep you from building a mental map they could optionally move the coin somewhere else afterward. Tough game right?

Now think about possible changes to the coin game that would make it solvable. For example, what if they were to remove the coin from the table if you turn it yellow-side up. That would be extremely useful right? Assuming every coin started yellow-side up, you would at most turn each coin twice, meaning you could turn all coins yellow-side up in at most 200 turns no matter which way they started. If the genetic algorithm, instead of the engine, had control of choosing the next index, then the equivalent would be for it to record the index it changes. Then, if `display` is called next by the engine instead of `get_index`, the algorithm would know that the array contains a 1 at that index. It could then simply add the index to a list of indexes it ignores when guessing. A side benefit of this solution is that, like the human player in the coin game, the genetic algorithm does not need to know the state of the entire array. It can begin with zero knowledge and start tracking indexes that improve the fitness, just like the classic Battleship game.

2.9. Summary

In this chapter the flexibility of the `genetic` module was increased by allowing the genes to be any type instead of only characters in a string. This is a critical feature, as before long we will want to use something other than just keyboard characters for genes.

2.10. Final Code

The final code for this chapter is available from:

https://github.com/handcraftsman/GeneticAlgorithmsWithPython/tree/master/ch02

Ready for more?

Genetic Algorithms with Python is available from major stores including Amazon, Apple and Barnes & Noble, in paperback, ePub, Kindle and PDF formats.

- https://www.amazon.com/Genetic-Algorithms-Python-Clinton-Sheppard/dp/1540324001/ (paperback)
- https://itunes.apple.com/us/book/genetic-algorithms-with-python/id1231392098 (ePub)
- https://www.amazon.com/dp/B01MYOWVJ2/ (Kindle Edition)
- https://leanpub.com/genetic_algorithms_with_python (PDF)

Thank you for your purchase.

Contents