Using The Genetics Library to Successfully Guess Passwords Using A Gene Set

Name:	Jordan Nelson
Date:	22/05/21

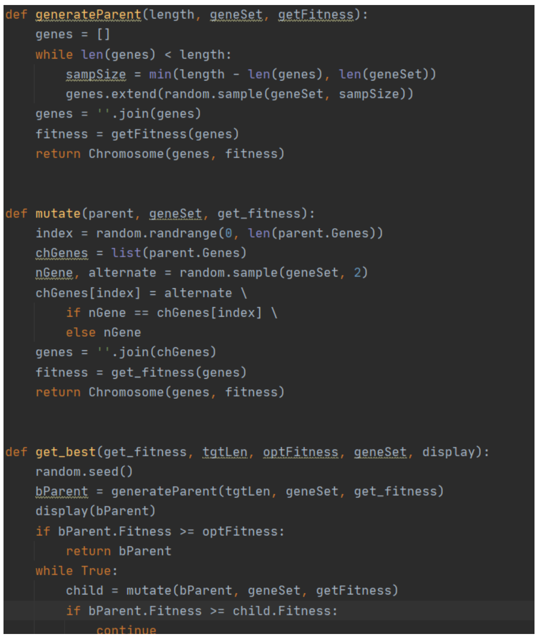
Table of Contents

Introduction

The idea of the program is to determine whether a gene set could crack a password. I will be using a gene set to test the algorithm(s), with the goal of eventually evaluating how accurately they are able to correctly guess a given password. I will be focusing on overall accuracy as an average; rather than assessing individual scores, to determine if the algorithm is more accurate ranging over multiple tests rather than individual results.

How it works (Overview)

Using Python and the Genetics Library, the program works by randomly generating a sequence of letters and then gradually one by one manipulating a different letter at a time until it reaches the optimal result.

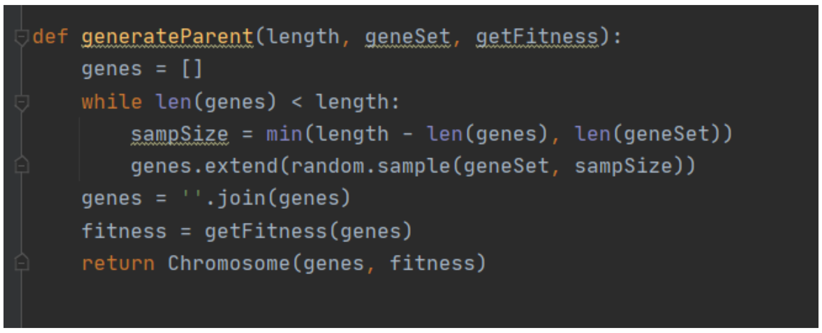


```python
def generateParent(length, geneSet, getFitness):
    genes = []
    while len(genes) < length:
        sampSize = min(length - len(genes), len(geneSet))
        genes.extend(random.sample(geneSet, sampSize))
    genes = ''.join(genes)
    fitness = getFitness(genes)
    return Chromosome(genes, fitness)


def mutate(parent, geneSet, get_fitness):
    index = random.randrange(0, len(parent.Genes))
    chGenes = list(parent.Genes)
    nGene, alternate = random.sample(geneSet, 2)
    chGenes[index] = alternate \
        if nGene == chGenes[index] \
        else nGene
    genes = ''.join(chGenes)
    fitness = get_fitness(genes)
    return Chromosome(genes, fitness)


def get_best(get_fitness, tgtLen, optFitness, geneSet, display):
    random.seed()
    bParent = generateParent(tgtLen, geneSet, get_fitness)
    display(bParent)
    if bParent.Fitness >= optFitness:
        return bParent
    while True:
        child = mutate(bParent, geneSet, getFitness)
        if bParent.Fitness >= child.Fitness:
            continue
```

We can see from the mutate function how this works. By starting from a random index in the parent we can then take a random sample from the gene set to manipulate it.

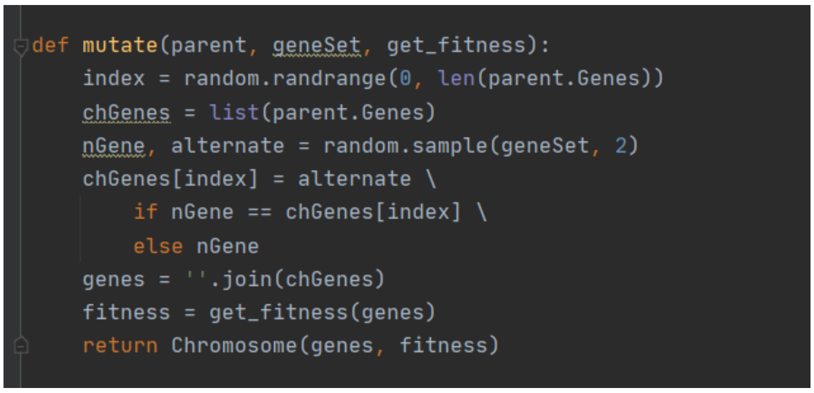
In short, the input is the geneset, and the output is how accurately the algorithms can correctly predict a given password based on the gene set. So what is the point of a password cracker in the real world? Well for one it can give us an indication of how strong a password is. While this is simple and doesn't for instance represent a brute force attack it does give us an idea of how machines are able to decipher a given string.

How it Works - The Details



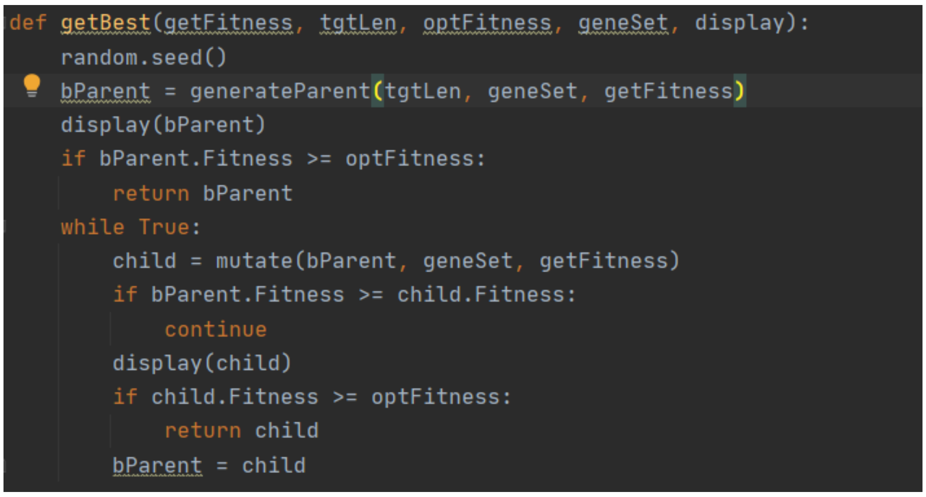
```python
def generateParent(length, geneSet, getFitness):
    genes = []
    while len(genes) < length:
        sampSize = min(length - len(genes), len(geneSet))
        genes.extend(random.sample(geneSet, sampSize))
    genes = ''.join(genes)
    fitness = getFitness(genes)
    return Chromosome(genes, fitness)
```

Random sample will take sampSize values from the input without replacing anything. What does this mean? It means that the parent will have no duplicate values within it unless the gene set itself contains duplicates or it exceeds the length of the gene set. Why is this useful? Well it allows us to create a long string while using a relatively small set of genes but also as many unique genes as is possible too.



```python
def mutate(parent, geneSet, get_fitness):
    index = random.randrange(0, len(parent.Genes))
    chGenes = list(parent.Genes)
    nGene, alternate = random.sample(geneSet, 2)
    chGenes[index] = alternate \
        if nGene == chGenes[index] \
        else nGene
    genes = ''.join(chGenes)
    fitness = get_fitness(genes)
    return Chromosome(genes, fitness)
```

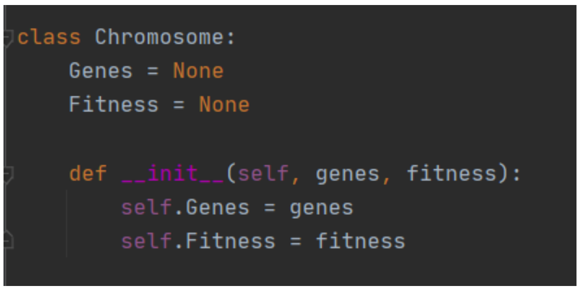
To guess the password we need to be able to be able to create a new guess each time. This is done by mutating individual elements at a time. As you can see we transform the parent string into an array. By doing this it allows us to focus on one character or index of the string at a time. We will randomly mutate each element of the array with one that is in our gene set. We will then reconstruct the result into a string before testing.



```python
def getBest(getFitness, tgtLen, optFitness, geneSet, display):
    random.seed()
    bParent = generateParent(tgtLen, geneSet, getFitness)
    display(bParent)
    if bParent.Fitness >= optFitness:
        return bParent
    while True:
        child = mutate(bParent, geneSet, getFitness)
        if bParent.Fitness >= child.Fitness:
            continue
        display(child)
        if child.Fitness >= optFitness:
            return child
        bParent = child
```

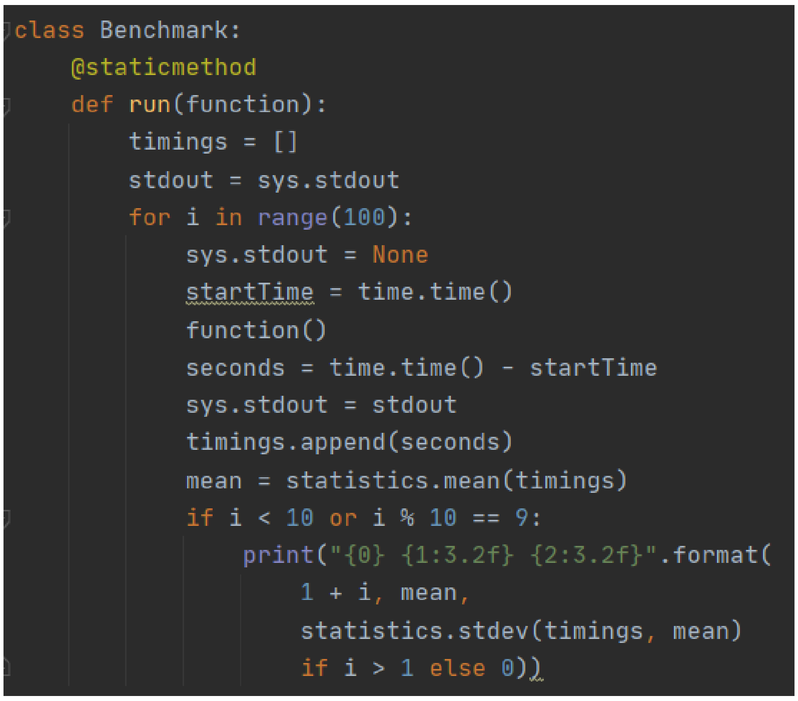
Now we will make use of the genetics library. We have here a function getBest. Notice that its parameters are the different functions we will need, such as getFitness to test our guess, the number of genes we will use to generate our guess, the optimal fitness, the set of genes we are going to be using and the display. The display being the least important as the algorithm doesn't need this to work but as humans, we do want to be able to see and make sure it did.

One thing of note here is that we often deal with the child rather than the parent. Such as when displaying it or testing the sequence. This is because we don't want the program to see the target value we provide - as this would defeat the point of the entire program.



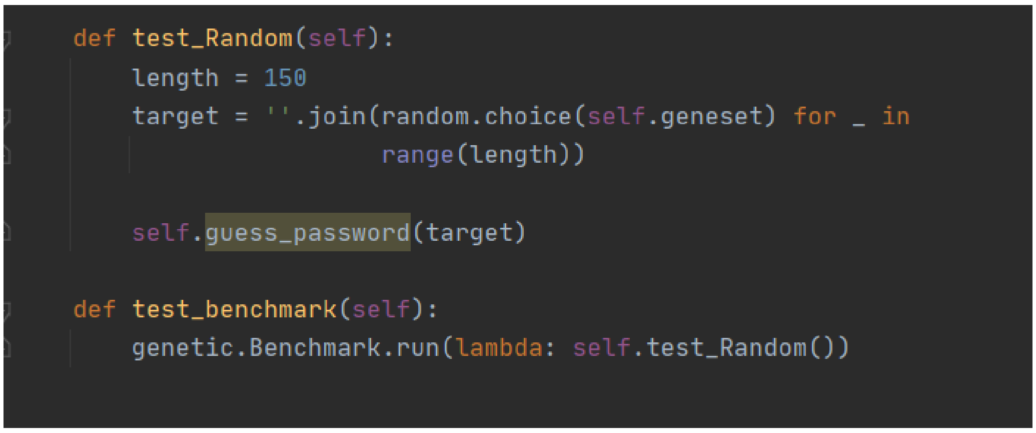
```python
class Chromosome:
    Genes = None
    Fitness = None

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

Then we have the Chromosome class. The whole purpose of this class is simply to allow the values to be passed around as one object rather than as separate entities.



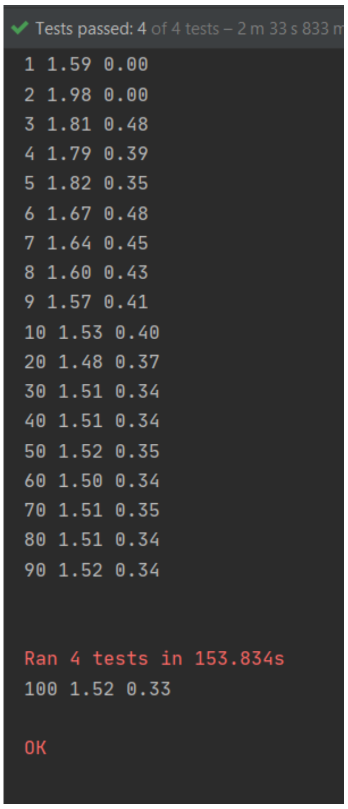
```python
class Benchmark:
    @staticmethod
    def run(function):
        timings = []
        stdout = sys.stdout
        for i in range(100):
            sys.stdout = None
            startTime = time.time()
            function()
            seconds = time.time() - startTime
            sys.stdout = stdout
            timings.append(seconds)
            mean = statistics.mean(timings)
            if i < 10 or i % 10 == 9:
                print("{0} {1:3.2f} {2:3.2f}".format(
                    1 + i, mean,
                    statistics.stdev(timings, mean)
                    if i > 1 else 0))
```

Finally in this file we have the benchmark class. This is so we can visually see our results on how long it takes for us to find the correct gene sequence (guess the password). As previously stated we will only be looking at the average performance to a standard deviation rather than individual performance. As we are running this 100 times it might be a good idea to cut down the amount of output we receive from the program to every 10th after the first ten have run. This is simply to save screen time as we won't be needing to review all 100 tests.



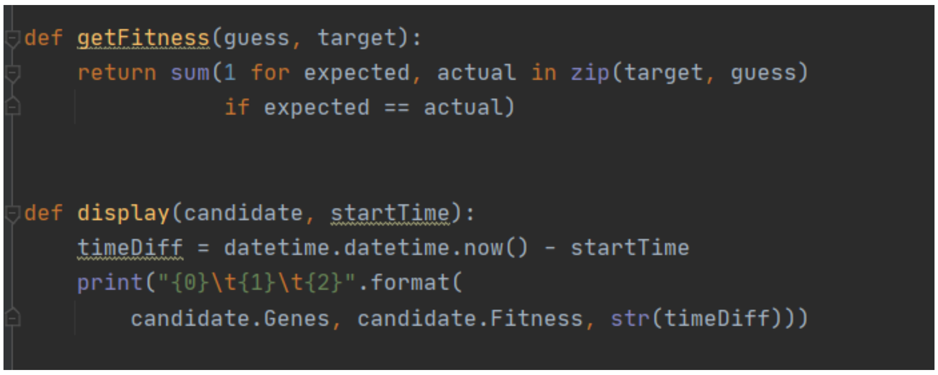
```python
    def test_Random(self):
        length = 150
        target = ''.join(random.choice(self.geneset) for _ in
                         range(length))

        self.guess_password(target)

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

To create a readable and usable output we will also create a random method. This is to deter the uncertainty of our results. What do I mean by this? Should we not have the random method it produces an output of 0.XXs. This is incredibly quick but it is uncertain to us, if this output is a true indicator of the speed the computer and program was able to perform this task as factors such as other programs running would have a minute affect on the result - the problem being the results are also minute so it would be hard to say.



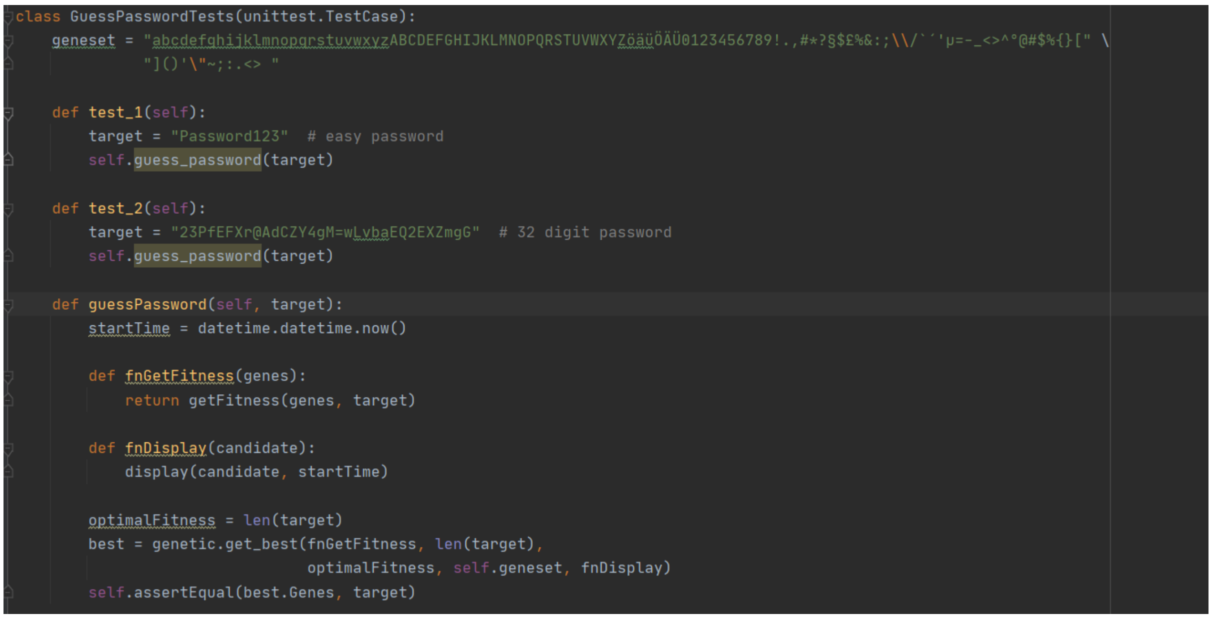
```
Tests passed: 4 of 4 tests – 2 m 33 s 833 m
1 1.59 0.00
2 1.98 0.00
3 1.81 0.48
4 1.79 0.39
5 1.82 0.35
6 1.67 0.48
7 1.64 0.45
8 1.60 0.43
9 1.57 0.41
10 1.53 0.40
20 1.48 0.37
30 1.51 0.34
40 1.51 0.34
50 1.52 0.35
60 1.50 0.34
70 1.51 0.35
80 1.51 0.34
90 1.52 0.34


Ran 4 tests in 153.834s
100 1.52 0.33

OK
```

In my tests these were the results. What does this mean exactly? These results indicate that on an average of 100 tests that ran it takes an average speed of 1.52s to guess the password. With a standard deviation of .33, this means it takes between 1.19 and 1.85 seconds to guess the password.



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


def display(candidate, startTime):
    timeDiff = datetime.datetime.now() - startTime
    print("{0}\t{1}\t{2}".format(
        candidate.Genes, candidate.Fitness, str(timeDiff)))
```

Display here takes the target password as a parameter, thus allowing us to test different passwords. The display is completely optional and it is up to you what you want to be displayed on screen when testing. The fitness the genetics algorithm provides is the only way that the program can guide itself to a solution we want. The value of it is the number of letters in the guess that match that of the letters in the password.



```
class GuessPasswordTests(unittest.TestCase):
    geneset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZöäüÖÄÜ0123456789!.,#*?§$£%&:;\\/`´'µ=-_<>^°@#$%{}[" \
              "]()'\"~;:.<> "

    def test_1(self):
        target = "Password123"  # easy password
        self.guess_password(target)

    def test_2(self):
        target = "23PfEFXr@AdCZY4gM=wLvbaEQ2EXZmgG"  # 32 digit password
        self.guess_password(target)

    def guessPassword(self, target):
        startTime = datetime.datetime.now()

        def fnGetFitness(genes):
            return getFitness(genes, target)

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

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

In GuessPasswords is where everything ties up and makes sense. We pass the potentially correct gene sequence as a parameter and then we can call the other methods we have discussed to execute the program as required. I chose to use a geneset that consisted of the alphabet both upper and lowercase as well as common symbols you may find on a english keyboard. As I use a german keyboard I also included some special characters such as öäüß and their capitalised counterparts.