```
horaInicio = datetime.datetime.now()
mejorPadre = generar_padre(len(objetivo))
mejorAptitud = obtener_aptitud(mejorPadre)
mostrar(mejorPadre)
```

La pieza final es el corazón del motor genético. Es un bucle que:

genera una conjetura, solicita la *aptitud* para esa conjetura, luego compara la *aptitud* con la de la mejor conjetura anterior, y se queda con la conjetura con la mejor aptitud.

Este ciclo se repite hasta que se presenta una condición de parada, en cuyo caso todas las letras en la conjetura coinciden con las del objetivo.

```
while True:
    niño = mutar(mejorPadre)
    niñoAptitud = obtener_aptitud(niño)
    if mejorAptitud >= niñoAptitud:
        continue
    mostrar(niño)
    if niñoAptitud >= len(mejorPadre):
        break
    mejorAptitud = niñoAptitud
    mejorPadre = niño
```

Ejecuta el código y verás una salida similar a la siguiente:

```
IDR!,VfWNjTM    0    0,0
IDR!,VfuNjTM    1    0,0
IDRl,VfuNjTM    2    0,0
IDol,VfuNjTM    3    0,0
IDol,VfuNdTM    4    0,0
IDol,VfundTM    5    0,0
IDolaVfundTM    6    0,0
IDolaVfundT!    7    0,0
¡DolaVfundT!    8    0,015481
¡HolaVfundT!    9    0,015481
¡HolaVMundT!    10    0,015481
¡Hola MundT!    11    0,031101
¡Hola Mundo!    12    0,046732
```

¡Conseguido!

Extraer un motor reutilizable

Tenemos un motor funcional pero actualmente está estrechamente vinculado al proyecto de la contraseña, por lo que la siguiente tarea es extraer el código del motor genético del programa específico para adivinar la contraseña de manera que pueda ser reutilizado en otros proyectos. Empieza creando un nuevo archivo llamado `genetic.py`.

Después, mueve las funciones *mutar* y *generar_padre* al nuevo archivo y renómbralas a _mutar y _generar_padre. Así es como se nombran

las funciones protegidas en Python. Las funciones protegidas sólo son accesibles a otras funciones en el mismo módulo.

Generación y mutación

Los proyectos futuros necesitarán personalizar el conjunto de genes, por lo que es necesario que éste sea un parámetro para _generar_padre y _mutar.

```
import random

def _generar_padre(longitud, geneSet):
    genes = []
    while len(genes) < longitud:
        tamañoMuestral = min(longitud - len(genes),
len(geneSet))
        genes.extend(random.sample(geneSet,
tamañoMuestral))
    return ''.join(genes)

def _mutar(padre, geneSet):
    índice = random.randrange(0, len(padre))
    genesDelNiño = list(padre)
    nuevoGen, alterno = random.sample(geneSet, 2)
    genesDelNiño[índice] = alterno if nuevoGen ==
genesDelNiño[
        índice] else nuevoGen
    return ''.join(genesDelNiño)
```

obtener_mejor

El siguiente paso es mover el bucle principal a una nueva función pública llamada obtener_mejor en el módulo `genetic`. Sus parámetros son:

la función a la que llama para solicitar la aptitud de una conjetura, el número de genes que se usarán al crear una nueva secuencia de genes, el valor de aptitud óptimo, el conjunto de genes que se usarán para crear y mutar las secuencias de genes, y la función a la que debería llamar para mostrar (o informar sobre) cada mejora encontrada.

```
def obtener_mejor(obtener_aptitud, longitudObjetivo,
aptitudÓptima, geneSet,
                  mostrar):
    random.seed()
    mejorPadre = _generar_padre(longitudObjetivo,
geneSet)
    mejorAptitud = obtener_aptitud(mejorPadre)
    mostrar(mejorPadre)
    if mejorAptitud >= aptitudÓptima:
        return mejorPadre

    while True:
        niño = _mutar(mejorPadre, geneSet)
        niñoAptitud = obtener_aptitud(niño)
```

```
    if mejorAptitud >= niñoAptitud:
        continue
    mostrar(niño)
    if niñoAptitud >= aptitudÓptima:
        return niño
    mejorAptitud = niñoAptitud
    mejorPadre = niño
```

Ten en cuenta que las funciones _mostrar y _obtener_aptitud se llaman con sólo un parámetro - la secuencia de genes hija. Esto es debido a que un motor genético no necesita acceder al valor objetivo, ni le importa el tiempo transcurrido, por lo que no se le pasan esos parámetros.

El resultado es un módulo reutilizable llamado `genetic` que se puede usar en otros programas mediante `import genetic`.

Usar el módulo

El código restante en `contraseña.py` es específico del proyecto de adivinar la contraseña. Para que funcione otra vez, primero importa el módulo `genetic`.

contraseña.py

```
import datetime
import genetic
```

Después, hay que definir las funciones auxiliares que sólo aceptan un parámetro para que sean compatibles con lo que espera el motor. Cada función auxiliar tomará la secuencia de genes candidata que reciba y llamará a las funciones locales con los parámetros adicionales según sea necesario. Hay que tener en cuenta que las funciones auxiliares están anidadas dentro de la función *adivine_contraseña*, por lo que tienen acceso a las variables objetivo y hora de inicio.

```
def test_Hola_Mundo():
    objetivo = "¡Hola Mundo!"
    adivine_contraseña(objetivo)


def adivine_contraseña(objetivo):
    geneset = "
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
¡!.,"
    horaInicio = datetime.datetime.now()

    def fnObtenerAptitud(genes): ①
        return obtener_aptitud(genes, objetivo)

    def fnMostrar(genes): ②
        mostrar(genes, objetivo, horaInicio)
```

```
    aptitudÓptima = len(objetivo)
    genetic.obtener_mejor(fnObtenerAptitud,
len(objetivo), aptitudÓptima,
                          geneset, fnMostrar)
```

Función de visualización

Ahora cambia la función mostrar para que tome la contraseña objetivo como parámetro. Podríamos implementarla como una variable global en el archivo del algoritmo, pero este cambio facilita probar diferentes contraseñas sin efectos secundarios.

```
def mostrar(genes, objetivo, horaInicio):
    diferencia = (datetime.datetime.now() -
horaInicio).total_seconds()
    aptitud = obtener_aptitud(genes, objetivo)
    print("{}\t{}\t{}".format(genes, aptitud,
diferencia))
```

La función aptitud

La función aptitud también necesita recibir la contraseña objetivo como parámetro.

```
def obtener_aptitud(genes, objetivo):
    return sum(1 for esperado, real in zip(objetivo,
genes)
               if esperado == real)
```

La función principal

Hay muchas maneras de estructurar el código principal, pero la más flexible es una prueba unitaria. Para hacer posible la ejecución del código desde la línea de comandos, añade:

contraseña.py

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

Si estás siguiendo los pasos en un editor, asegúrate de ejecutar tu código para verificar que funciona en este punto.

Usa el framework `unittest` de Python

El siguiente paso es hacer que el código funcione con el framework de pruebas incorporado de Python.

```
import unittest
```

Para hacerlo, hay que mover la función de prueba principal a una *clase* que herede de

`unittest.TestCase`. También puedes mover las otras funciones dentro de la *clase* si quieres, pero si lo haces debes añadir `self` como primer parámetro de cada una porque entonces pasarán a formar parte de la *clase* de prueba.

```
# `nosetests` no admite caracteres como ñ en el
nombre de la clase
class PruebasDeContrasena(unittest.TestCase):
    geneSet = "
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
¡¡.,"

    def test_Hola_Mundo(self):
        objetivo = "¡Hola Mundo!"
        self.adivine_contraseña(objetivo)

    def adivine_contraseña(self, objetivo):
...
        aptitudÓptima = len(objetivo)
        mejor =
genetic.obtener_mejor(fnObtenerAptitud,
len(objetivo),
                              aptitudÓptima,
self.geneSet,
                              fnMostrar)
        self.assertEqual(mejor, objetivo)
```

Al llamar a la función `main` del módulo `unittest`, ésta ejecuta automáticamente cada función cuyo nombre empieza por `test`.

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

Esto permite que la prueba se ejecute desde la línea de comandos y, además, sin la salida de su función mostrar.

```
python -m unittest -b contraseña
.
----------------------------------------
Ran 1 test in 0,020s

OK
```

Si obtienes un error como `'module' object has no attribute 'py'` entonces es que usaste el nombre del archivo `contraseña.py` en lugar del nombre del módulo `contraseña'.

Una contraseña más larga

"¡Hola Mundo!" no demuestra suficientemente el poder del motor genético, por lo que prueba una contraseña más larga:

```
def test_Porque_me_formaste_de_una_manera_formidable
_y_maravillosa(self):
    objetivo = "Porque me formaste de una manera
formidable y " \
               "maravillosa."
    self.adivine_contraseña(objetivo)
```

Ejecuta

```
...
PorqueZweSformaste de una manerc formidable y
maraviXlosa.    53    0,109355
PorqueZmeSformaste de una manerc formidable y
maraviXlosa.    54    0,109355
PorqueZmeSformaste de una manerc formidable y
maravillosa.    55    0,109355
PorqueZme formaste de una manerc formidable y
maravillosa.    56    0,109355
PorqueZme formaste de una manera formidable y
maravillosa.    57    0,171862
Porque me formaste de una manera formidable y
maravillosa.    58    0,187489
```

¡Bien!

Introduce una *clase* Cromosoma

El siguiente cambio es introducir una *clase* Cromosoma que tenga los atributos *Genes* y *Aptitud*. Esto hará que el motor genético sea más flexible permitiendo que pase esos valores como una unidad.

genetic.py

```
class Cromosoma:
    def __init__(self, genes, aptitud):
        self.Genes = genes
        self.Aptitud = aptitud

def _mutar(padre, geneSet, obtener_aptitud):
    índice = random.randrange(0, len(padre.Genes))
    genesDelNiño = list(padre.Genes)
...
    genes = ''.join(genesDelNiño)
    aptitud = obtener_aptitud(genes)
    return Cromosoma(genes, aptitud)

def _generar_padre(longitud, geneSet,
obtener_aptitud):
...
    genes = ''.join(genes)
    aptitud = obtener_aptitud(genes)
    return Cromosoma(genes, aptitud)

def obtener_mejor(obtener_aptitud, longitudObjetivo,
aptitudÓptima, geneSet,
                  mostrar):
    random.seed()
    mejorPadre = _generar_padre(longitudObjetivo,
```

```
geneSet, obtener_aptitud)
    mostrar(mejorPadre)
    if mejorPadre.Aptitud >= aptitudÓptima:
        return mejorPadre

    while True:
        niño = _mutar(mejorPadre, geneSet,
obtener_aptitud)

        if mejorPadre.Aptitud >= niño.Aptitud:
            continue
        mostrar(niño)
        if niño.Aptitud >= aptitudÓptima:
            return niño
        mejorPadre = niño
```

Esto requiere cambios compensatorios en las funciones del archivo del algoritmo, pero esos cambios también eliminan un poco de trabajo doble (a saber, recalcular la aptitud).

contraseña.py

```
def mostrar(candidato, horaInicio):
    diferencia = (datetime.datetime.now() -
horaInicio).total_seconds()
    print("{}\t{}\t{}".format(
        candidato.Genes, candidato.Aptitud,
diferencia))

class PruebasDeContrasena(unittest.TestCase):
...
    def adivine_contraseña(self, objetivo):
...
        def fnMostrar(candidato):
            mostrar(candidato, horaInicio) ①

        aptitudÓptima = len(objetivo)
        mejor =
genetic.obtener_mejor(fnObtenerAptitud,
len(objetivo),
                                        aptitudÓptima,
self.geneSet,
                                        fnMostrar)
        self.assertEqual(mejor.Genes, objetivo) ②
```

Evaluación comparativa

La siguiente mejora es añadir soporte para evaluaciones comparativas (benchmarking) porque es útil saber cuánto tarda el motor en encontrar una solución de media y la desviación estándar. Eso se puede hacer con otra *clase* del siguiente modo:

genetic.py

```
class Comparar:
    @staticmethod
    def ejecutar(función):
        cronometrajes = []
        for i in range(100):
```

```
            horaInicio = time.time()
            función()
            segundos = time.time() - horaInicio
            cronometrajes.append(segundos)
            promedio =
statistics.mean(cronometrajes)
            print("{} {:3.2f} {:3.2f}".format(
                1 + i, promedio,
                statistics.stdev(cronometrajes,
promedio) if i > 1 else 0))
```

Esta función ejecuta la función proporcionada 100 veces e informa de cuánto tarda cada ejecución, la media y la desviación estándar. Para calcular la desviación estándar, usaremos un módulo de terceros llamado `statistics`:

genetic.py

```
import statistics
import time
```

Puede que necesites instalar el módulo `statistics` en tu sistema. Puedes hacerlo desde la línea de comandos con `python -m pip install statistics`.

Ahora, para usar la capacidad de benchmarking, simplemente añade una prueba y pasa la función que quieres evaluar.

contraseña.py

```
def test_comparativa(self):

 genetic.Comparar.ejecutar(self.test_Porque_me_forma
ste_de_una_manera_formidable_y_maravillosa)
```

Al ejecutarse, esta función funciona genial pero es un poco verbosa porque también muestra la salida de la función mostrar para las 100 ejecuciones. Esto se puede corregir en la función benchmark redirigiendo temporalmente la salida estándar a ninguna parte.

genetic.py

```
import sys
...
class Comparar:
    @staticmethod
    def ejecutar(función):
...
        cronometrajes = []
        stdout = sys.stdout ①
        for i in range(100):
            sys.stdout = None ②
            horaInicio = time.time()
            función()
            segundos = time.time() - horaInicio
            sys.stdout = stdout ③
            cronometrajes.append(segundos)
```

```
...
```

Si obtienes un error como el siguiente al ejecutar la prueba de benchmark:

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

Entonces es que probablemente estés usando Python 2.7, que no permite redirigir `stdout` a `None`, ya que necesita escribir los datos en algún lugar. Una solución a este problema es añadir la siguiente *clase*:

genetic.py

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

Luego sustituye lo siguiente en la función de ejecución:

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

por:

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

Ese cambio te permite sortear la diferencia entre Python 2.7 y 3.5 por esta vez. Sin embargo, el código en proyectos futuros usa otras características de Python 3.5, por lo que te sugiero que te pases a Python 3.5 para poder centrarte en aprender sobre los algoritmos genéticos sin estos problemas adicionales. Si quieres usar herramientas de aprendizaje automático que estén vinculadas a Python 2.7, espera hasta que tengas una comprensión sólida de los algoritmos y luego cambia a Python 2.7.

La salida también se puede mejorar mostrando sólo las estadísticas de las primeras diez ejecuciones y luego cada décima ejecución después de eso.

genetic.py

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

Ahora la salida de la prueba de benchmark tiene el siguiente aspecto.

salida de muestra

```
1 0,30 0,00
2 0,29 0,00
3 0,27 0,04
...
9 0,28 0,07
10 0,27 0,07
20 0,27 0,07
...
90 0,26 0,07
100 0,26 0,07
```

Esto significa que, con un promedio de 100 ejecuciones, se tardan 0,26 segundos en adivinar la contraseña, y el 68% del tiempo (una desviación estándar) se tarda entre 0,19 (0,26 - 0,07) y 0,33 (0,26 + 0,07) segundos. Por desgracia, eso es probablemente demasiado rápido para determinar si un cambio es debido a una mejora en el código o a otra cosa que se ejecuta en el ordenador en ese momento. Este problema se puede solucionar haciendo que el algoritmo genético adivine una secuencia aleatoria que tarda 1-2 segundos en ejecutarse.

contraseña.py

```
import random
...
    def test_aleatorio(self):
        longitud = 150
        objetivo =
''.join(random.choice(self.geneSet)
                              for _ in range(longitud))
        self.adivine_contraseña(objetivo)

    def test_comparativa(self):

 genetic.Comparar.ejecutar(self.test_aleatorio)
```

Es probable que tu CPU sea diferente de la mía, así que ajusta la longitud según sea necesario. En mi sistema, eso da como resultado:

```
         Benchmarks actualizados
  media (segundos)  | desviación estándar
----------------------------------------
      1,36          |          0,30
```

Resumen

En este proyecto, creamos un motor genético sencillo que utiliza la mutación aleatoria para producir mejores resultados. Este motor fue capaz de adivinar una contraseña secreta dada sólo su longitud, un conjunto de caracteres que podrían estar en la contraseña, y una función de aptitud que