Genetic Algorithms in Python with DEAP

Samir Kanaan - UOC

May 12, 2016

1 Objectives

This guide is a brief introduction to the framework **Distributed Evolutionary Algorithms in Python (DEAP)**, which at this moment is the most complete and up to date library of genetic algorithms (GA) in Python. [1]

The objectives of this guide are:

- Show how to install DEAP, which is really easy by the way.
- Create, setup and run a GA that optimizes a simple problem.
- Retrieve statistics from the execution of a GA and visualize its evolution throughout its iterations.
- Use DEAP to model and solve a combinatorial problem (the classical travel salesman problem).

Each section in this document corresponds to one of these objectives. The programs are split up in groups of instructions in order to give a proper explanation. At the end of the document there are some appendixes with the complete code of the programs.

2 DEAP setup

DEAP package is in the official Python package index (PyPI), therefore it can be installed simply using the `pip` command. On Linux you just have to open a terminal and run the following (assuming you are going to work with Python 3):

```
pip3 install deap
```

If you do not get any error message then you probably have it. You can open your Python interpreter and make `import deap` to check that it has been properly installed.

[1] Among other things it is the only GA library that works perfectly with Python 3.x.

On Windows you have to get to your Python executables folder, where **pip.exe** is, open a terminal there (probably you will have to shift+right click on the folder and select "open command console" or something similar), then type:

```
pip.exe install deap
```

You can also find a lot of useful information and a complete DEAP reference on its official website[2].

3 First GA with DEAP

DEAP is a very flexible framework for evolutionary programming, including GA, but the drawback to this flexibility is that a certain amount of coding is required to solve any optimization problem. In this example we will solve the **Onemax**[3] problem, which is a classical optimization example where each solution is a string of bits and the goal is to find the string of bits whose sum is maximum (i.e. all ones). Although its solution is trivial, the optimization algorithm does not know the "trick" (set all bits to one), therefore it has to perform a search through the whole solution space. This makes the Onemax problem a good test example where we know the optimal solution. The basic Onemax problem is defined with bits, although other numerical types can be used.

The solution of this problem with GA is pretty straightforward: each individual is a possible solution, and its genetic information is the string of bits, i.e. each bit is a gene. The code presented in this section is adapted from a DEAP example[4].

3.1 Imports

Let us go step by step. First the imports: we will use `random` to generate the initial individuals (their 0/1 bits). From DEAP we will import several packages: `creator` is the package to create individuals and populations, `base` contains the basic elements of DEAP, `tools` contains the functions to perform the crossover and mutation of individuals, among others, and finally `algorithms` contains the GA algorithms themselves.

```
import random
from deap import creator, base, tools, algorithms
```

[2] https://github.com/DEAP/deap
[3] http://tracer.lcc.uma.es/problems/onemax/onemax.html
[4] https://github.com/DEAP/deap/blob/master/examples/ga/onemax.py

3.2 Definition of the individuals and the population

The first step with DEAP (as well as with any GA approach) is to define the indivuals and to create a population with them. In order to define an individual (i.e. a solution), two instructions are required. The first one states the **goal** of the individual (usually either to maximize or to minimize a given function). We will use either `FitnessMax` or `FitnessMin` depending on the optimization goal of the problem[5]. `base.Fitness` is the class from which our goal derives. Finally, the `weights` argument is a **tuple** (that is why the final comma is there, *do not forget to place it*) of weights of the problem's objectives. The weight should be positive in maximization problems and negative in minimization problems. DEAP can be used to solve multiobjective problems, therefore several weights could be given (even mixing maximization and minimization objectives).

```
creator.create('FitnessMax', base.Fitness, weights=(1.0,))      # TUPLE!!
```

The second instruction defines the **structure** of individuals, in this case gives them a name (`Individual`, also user-defined), specifies that they will be a `list` of attributes, and finally associates them with the previously defined fitness goal.

```
creator.create('Individual', list, fitness=creator.FitnessMax)
```

Next, the `toolbox` is created so other operations can be accessed and the individuals can be completely defined. The first `register` gives a name to the bit attributes (`attrBool`, user-defined), and tells DEAP how to generate them (using `random.randint` with arguments 0 and 1, which are the limits of `randint`. If you use another function to create the random values, you will have to give it the right number and type of arguments). The second `register` gives a name to the individuals, defines how to create their attributes (in this case by repeating them), associates the previously defined `creator` and finally specifies the function that creates the attributes (`toolbox.attrBool`) and how many attributes do the individuals have (50).

Finally, the third `register` defines the population: gives it a name, tells how to create the individuals (repeating them with `tools.initRepeat`), how to store the individuals (on a list) and finally the name of the registered individual (`toolbox.individual`).

```
toolbox = base.Toolbox()
toolbox.register('attrBool', random.randint, 0, 1)
toolbox.register('individual', tools.initRepeat, creator.Individual,
                 toolbox.attrBool, n=50)
toolbox.register('population', tools.initRepeat, list, toolbox.individual)
```

[5]Note that these names are user-defined, so other names could be used instead.

3.3 Objective function

Now that the structure of an individual is defined, we have to write the objective function, whose input is an individual (i.e. a possible solution of the problem) and whose output is a value with the fitness or quality of the solution. For example, the benefit obtained, energy wasted, error value, time required... As usual, depending on the problem we may want to either maximize or to minimize the value returned by the objective function. In the Onemax example we want to maximize the sum of the bits, so the function is as follows:

```
def evalOneMax(individual):
    return sum(individual),      # COMMA HERE!!! - MUST BE A TUPLE
```

Really easy this one!

But keep in mind that a very important detail when using DEAP is that as it is designed to solve multiobjective problems, the output of the objective function can have several values. Therefore it **always has to be a tuple!!!**. That is why there is a comma at the end of the return sentence, to coerce our single value into a tuple.

3.4 Genetic algorithm strategies

The final step before running the GA itself is to configure its strategies for mutation, crossover and individual selection. In this piece of code there are four register operations. The first one register the objective function previously created. The second one defines the crossover or mating function (crossover on two points in this case). The third register defines the mutation function (flip a bit in this case, as the attributes are bits) along with the probability of each bit of being mutated. Please note that this probability may have to be adjusted depending on the number of attributes in the problem. Finally, the selection strategy, in this case a tournament between the individuals with 3 rounds which usually gives good results.

```
toolbox.register('evaluate', evalOneMax)
toolbox.register('mate', tools.cxTwoPoint)
toolbox.register('mutate', tools.mutFlipBit, indpb = 0.05)
toolbox.register('select', tools.selTournament, tournsize=3)
```

3.5 Running the genetic algorithm

Finally the time has come to run the GA itself. First the population is created, in this case with 100 individuals, then the number of generations (i.e. iterations) is set, then a `for` loop begins the iterations.

Inside the GA loop there are five steps:

1. Generate the new population (the offspring) from the previous one, with a crossover probability on each individual of 0.5 (`cxpb`) and an individual

mutation probability of 0.1 (`mutpb`, it is the probability of each individual to go through the mutation process as defined above).

2. Compute the fitness values of the new population with the objective function defined above.
3. Associate the new fitness with the new individuals (`for` loop).
4. Replace the old population with the newly created one.
5. (Optional) Select the best individual from the new population and print its fitness and attributes to screen to see the progress of the GA.

```
population = toolbox.population(100)

NGEN = 40

for gen in range(NGEN):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)
    fits      = toolbox.map(toolbox.evaluate, offspring)
    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

    # Print the best individual on each iteration with its objective value
    top = tools.selBest(population, k=1)
    print(gen, evalOneMax(top[0]), top[0])
```

If you run the complete program you will see on screen the fitness value and attributes of the best individual on each generation. With the configuration used you will probably get the optimum (sum equal to the number of bits, i.e. all ones).

```
...
36 (49,) [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, ..., 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
37 (49,) [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, ..., 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
38 (49,) [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, ..., 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
39 (49,) [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, ..., 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>
```

Ops! Well, remember that GAs are strongly random in nature, so you will have to run them several times to find out the average performance. This is specially important when trying to adjust parameters (population size, number of iterations, mutation probabilities).

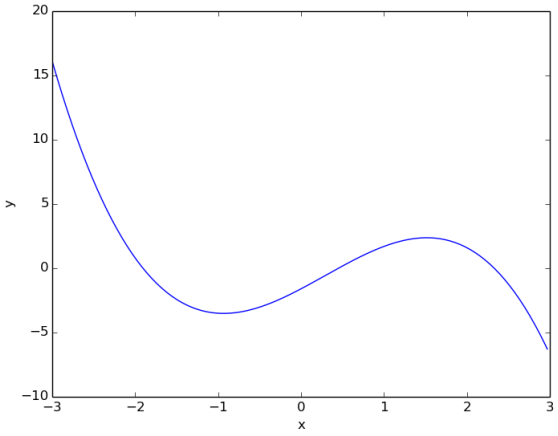

Figure 1: Example signal as read from the file.

4 Gathering Statistics

The objective of the next example is to show how to gather statistics of the execution of the GA, for example the minimum/average/maximum fitness on each iteration, in order to analyze the behaviour of the process and possibly make some fine tuning of its parameters.

This example is a very simple one: we have read a signal from an instrument and we want to approximate it using a third degree polynomial ($ax^3 + bx^2 + cx + d$). The optimization algorithm must find the best coefficients (a, b, c, d), i.e. those that produce a polynomial whose difference with the signals is minimum (ideally zero).

x takes 200 equidistant values from -3 to 3. These values are produced by the following Python expression:

```
xVal = numpy.array([x*6.0/200 - 3 for x in range(200)])
```

The signal (y) values can be read from file **signal.data**. Figure 1 shows the signal. By the way, the polynomial that generated the signal is $-0.8x^3 + 0.7x^2 + 3.4x - 1.6$, so you can check your results.

4.1 Program initialization

The setup of DEAP for this problem is basically the same as in the Onemax example, I will only remark the main differences. In this case it is a minimization problem, as the objective will be to produce a polynomial whose difference with the signal read from the file is minimum. Therefore we define a `FitnessMin` objective with negative weight. Also in this case the attributes of the individuals

are the coefficients of the third degree polynomial. Therefore they are four real numbers, that are initialized as random values in the range $[-1, +1]$.

```
import random
from deap import creator, base, tools, algorithms

creator.create('FitnessMin', base.Fitness, weights=(-1.0,))
creator.create('Individual', list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

toolbox.register('attrFloat', lambda: random.random()*2  - 1)  # Rands -1 to +1
toolbox.register('individual', tools.initRepeat, creator.Individual,
                 toolbox.attrFloat, n=4)
toolbox.register('population', tools.initRepeat, list, toolbox.individual)
```

The objective function measures the difference (mean squared error) between the signal and the candidate polynomial. It is split in two functions: one that computes the polynomial given a solution (a set of coefficients), and another one that computes the difference. Just remember to type the final comma on the return instruction so the objective function returns a tuple.

```
# Define x and objective function
import numpy

# Values in x on which the signal is defined
xVal = numpy.array([x*6.0/200 - 3 for x in range(200)])

# Read the signal data
yVal = numpy.loadtxt('signal.data')

# Define the signal simulation function (generic 3rd grade
# polynomial)
def polynomial(solution):
    a, b, c, d = solution
    return numpy.array([a*x**3 + b*x**2 + c*x + d for x in xVal])


import math
from sklearn.metrics import mean_squared_error

def objective(solution):
    return math.sqrt(mean_squared_error(yVal,
                polynomial(solution)))/(max(yVal) - min(yVal)) ,  # FINAL COMMA!
```

As for the GA configuration, we will use one point crossover as there are so few attributes, a Gaussian mutation (adds a random number to the attributes)

with mean 0 and sigma 0.5, and a tournament selection strategy. The test population will have 40 individuals and the GA will run for 60 generations.

```
toolbox.register('evaluate', objective)
toolbox.register('mate', tools.cxOnePoint)
toolbox.register('mutate', tools.mutGaussian, mu=0, sigma=0.5, indpb = 0.2)
toolbox.register('select', tools.selTournament, tournsize=3)


population = toolbox.population(40)

NGEN = 60
```

4.2 Using the Statistics module

In order to gather statistics from the execution of the GA, first we have to tell the `Statistics` module what is the parameter that should guide the statistic. In this case it is the fitness value, that is accessed on a given individual `ind` by its `fitness.values` property using the `lambda` function created on the first line.

Then we have to specify which statistical values we want to gather. In our case it will be the minimum and the average fitness of all the individuals on each generation. Other values such as the maximum or the standard deviation of the fitness values can be collected as well. The logbook is an object where the values from the population are recorded.

```
# Statistics
stats = tools.Statistics(key=lambda ind: ind.fitness.values)
stats.register('min', numpy.min)
stats.register('avg', numpy.mean)
#stats.register('std', numpy.std)
#stats.register('max', numpy.max)

logbook = tools.Logbook()
```

Then, inside the GA loop itself, the calls to `stats.compile` and `logbook.record` must be made right after the new population is generated, so their statistical values are recorded into the logbook for further use.

```
for gen in range(NGEN):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)
    fits      = toolbox.map(toolbox.evaluate, offspring)
    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))
```

```
    # Stats recording
    record = stats.compile(population)
    print(record)  # To see the values on each iteration

    logbook.record(gen=gen, **record)
```

When the GA loop ends, the data gathered can be collected using the call `logbook.select`. In the following example, such data is used to plot the minimum and average fitness along the generations of the algorithm, as shown in Figure 2.

```
# Gather the data collected and plot it
generation = logbook.select('gen')
fitnessMin = logbook.select('min')
fitnessAvg = logbook.select('avg')

import matplotlib.pyplot as plt

line1 = plt.plot(generation, fitnessMin, "b-", label="Minimum Fitness")
plt.xlabel("Generation")
plt.ylabel("Fitness (error)")

line2 = plt.plot(generation, fitnessAvg, "r-", label="Average Fitness")

lns = line1 + line2
labs = [l.get_label() for l in lns]
plt.legend(lns, labs, loc="center right")

plt.show()
```

5 Solving Combinatorial Problems

Combinatorial problems are problems where a solution is a specific ordering of a set of features, rather than a specific set of values. Therefore the solution space is the set of all possible permutations of the features in the problem. A classical combinatorial problem is known as the **travelling salesman problem** (TSP). In this problem, there is a set of cities, with a certain distance between each pair of cities, and a travelling salesman is required to visit all cities once and only once. The solutions to this problem are required to minimize the total distance traversed by the salesman (to save time, fuel, etc.). Figure 3 shows a simple instance of this problem.

In general terms, this problem tries to find the shortest path to traverse all the nodes in a weighted graph.

The solutions to the TSP are the different orderings in which the cities can be visited, which is a space of size $n!$ where n is the number of cities.

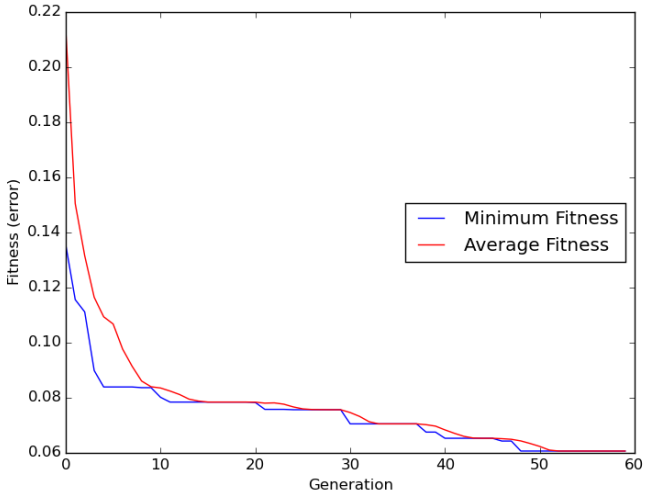

Figure 2: Evolution of the population of the GA solving the signal fitting problem.

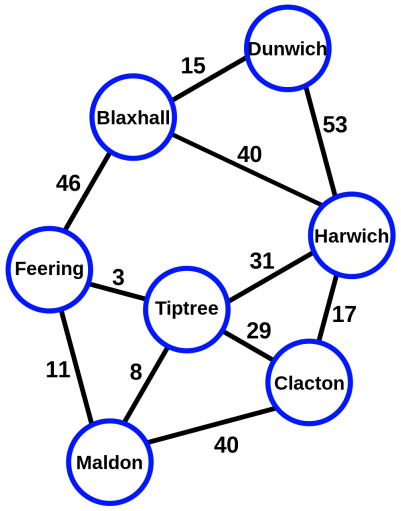

Figure 3: A travelling salesman problem example. Source: wikimedia.org

Obviously for a large number of cities an exhaustive exploration is too expensive computationally speaking.

GAs are a suitable option to find a reasonably good solution for the TSP problem. However they have to be configured in a very specific way in order to reflect the combinatorial nature of the problem. The special features for this kind of problems are (applied to TSP):

- A solution is a permutation of all the cities
- All cities must appear once and only once in the solution...
- Therefore the mutations have to be a swap between two cities...
- And the crossover must keep all cities once and only once in the new solution
- To make the solutions as general as possible, they represent the cities by their indices (0, 1, ... n-1)

Next we will see how to configure DEAP to solve this kind of problems[6].

First, the set of cities is randomly generated with coordinates (x, y). The distance between cities is the euclidean distance.

```
# -*- coding: utf-8 -*-
# Generate N cities (x,y)
import random, math

# In order to have the same problem on every run
random.seed(0)

def distance(cityA, cityB):
    return math.sqrt((cityA[0]-cityB[0])**2 + (cityA[1]-cityB[1])**2)

def generateCities(n):
    return [(random.randint(0,500), random.randint(0,500))
            for c in range(n)]

numCities = 30
cities    = generateCities(numCities)
print(cities)
```

The generation of a solution uses a permutation of the indices of the cities. The crossover strategy is named `cxOrdered`, which keeps one and only one copy of each index. The mutation just swaps a pair of indices, function `mutShuffleIndexes`.

[6] Adapted from https://github.com/DEAP/deap/blob/master/examples/ga/tsp.py

```
# Configure DEAP
from deap import algorithms, base, creator, tools
import numpy

toolbox = base.Toolbox()
creator.create('FitnessMin', base.Fitness, weights=(-1.0,))
creator.create('Individual', list, fitness=creator.FitnessMin)

toolbox.register('Indices', numpy.random.permutation, len(cities))
toolbox.register('Individual', tools.initIterate, creator.Individual,
                 toolbox.Indices)
toolbox.register('Population', tools.initRepeat, list, toolbox.Individual)

toolbox.register('mate', tools.cxOrdered)
toolbox.register('mutate', tools.mutShuffleIndexes, indpb=0.05)
```

Finally, the objective function computes the distance traversed by the given ordering of indices, and the GA is run as usual.

```
# Total distance traversed by a given solution
def totalDistance(individual):
    return sum((distance(cities[individual[i]], cities[individual[i-1]])
                for i in range(len(individual)))),


# Execute the GA
toolbox.register('evaluate', totalDistance)
toolbox.register('select', tools.selTournament, tournsize=3)

population = toolbox.Population(100)
NGEN = 100

for gen in range(NGEN):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)
    fits      = toolbox.map(toolbox.evaluate, offspring)
    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

    top = tools.selBest(population, k=1)
    print(gen, totalDistance(top[0]), top[0])

print('='*40)
top10 = tools.selBest(population, k=10)
for ind in top10:
    print(totalDistance(ind), ind)
```

6 Problems with Constraints

Optimization problems often impose certain constraints on the possible solutions. It is important to take these constraints into account in order to avoid wasting computer time exploring solutions that are not acceptable.

Let us work with a specific problem in order to discuss how to deal with constraints. In this problem, we have an amount of money (50000) that we want to invest in the commodities market, and of course we expect the maximum profit from our investment. There are ten different commodity types, each with a base buy price and a sell price, and we can buy as many units of each as we want to.

```
buyPrices  = [100, 80, 90, 130, 105, 60, 80, 40, 140, 110]
sellPrices = [112, 95, 99, 140, 113, 77, 88, 53, 146, 121]
money      = 50000
```

The question is, although the sell price is constant, the buy price is not; more specifically, if we buy n units of a given commodity, their actual buy price will be given by the expression $actualBuyPrice = buyPrice * (1 + n/1000)$. For example if we buy 60 units of the first commodity, their buy price will be $100 * (1 + 60/1000) = 106$, so we will have to spend $60 * 106 = 6360$ buying them.

So the goal of the problem is to find the amount to buy of each commodity so that the profit is maximized. We assume that a solution will be a list of integers, with the number of units bought of each commodity (i.e. one number per commodity). But we have 50000 money to invest, so the total spent cannot be above that amount (although it may be lower). Here lies the constraint of our problem.

There are three main strategies to deal with constraints in optimization problems:

1. Encode the problem's solutions in such a way that constraint violations are impossible. In this case such a codification would imply to see each number in the solution as a percentage of money spent on each commodity. But this would be rather counterintuitive and may give place to suboptimal solutions (non-integer number of units, having to drop the fractional part, and so on).

2. Add a severe penalty to the objective function to the solutions that violate the constraints, for example dividing the benefit by the excess money spent, so they naturally avoid the forbidden area outside the constraints.

3. Tell the GA to check that the new population created on each iteration (after crossover and mutation) satisfies the constraints, or fix the individuals that do not. This is the approach shown in this Section.

DEAP's toolbox allows to add *decorators* (like wrapper functions) to the crossover and mutation operators, so all the individuals are checked by a user-defined function. Inside this function, individuals that violate a constraint are

forced back to the "acceptable" area of the solution space. In this case, the strategy is to randomly substract one purchased unit from the individual until the money it spends is within the limits.

Let us see how the whole process is done in DEAP. The configuration is quite common, with individuals as list of integers randomly initialized between 0 and 100, one-point crossover, and mutations assigning a random value between 0 and 200 to an attribute.

```
import random
from deap import creator, base, tools, algorithms

creator.create('FitnessMax', base.Fitness, weights=(1.0,))
creator.create('Individual', list, fitness = creator.FitnessMax)

toolbox = base.Toolbox()

# Initially amounts purchased are 0..100, but on mutation they can be 0..200
# (so that initial population individuals are valid)
toolbox.register('attrInt', lambda: random.randint(0,100))
toolbox.register('individual', tools.initRepeat, creator.Individual,
                 toolbox.attrInt, n=len(buyPrices))

toolbox.register('population', tools.initRepeat, list, toolbox.individual)

# Set up the GA elements
toolbox.register('evaluate', objective)
toolbox.register('mate', tools.cxOnePoint)
toolbox.register('mutate', tools.mutUniformInt, low=0, up=200, indpb=0.1)
toolbox.register('select', tools.selTournament, tournsize=3)
```

The objective function computes the profit from the investments received, taking into account the increase of buying price:

```
def objective(solution):
    return sum([amt*(sp - bp*(1+amt/1000))
                for amt, bp, sp in zip(solution, buyPrices, sellPrices)]),
```

If you run the program (in this example with 100 individuals and 400 generations, file `investments.py`), you will probably get something like this:

```
499 profit= (4431.66,) spent= (59287.34,)
investments= [60, 96, 50, 38, 40, 142, 50, 163, 21, 50]
```

So the best solution is not acceptable because it spends more money than you have (50000). In order to add a constraint to the solution, you have to define a decorator function like this (yes, three nested functions, the outer one is named by you and receives any constraint-related parameters you may need):

```
def enforceBudget(budget):
    def decorator(func):
        def wrapper(*args, **kargs):
            offspring = func(*args, **kargs)
            for child in offspring:
                while spent(child) > budget:
                    pos = random.randint(0, len(child)-1)
                    # It is critical to avoid negative attributes
                    child[pos] = max(child[pos]-1, 0)
            return offspring
        return wrapper
    return decorator
```

Basically what it does is to check all the individuals (`child`) in the population (`offspring`) and randomly substract one purchased unit until the money spent is within the limits. By the way, the `spent` function simply adds the total money spent:

```
def spent(solution):
    return sum([amt*bp*(1+amt/1000) for amt, bp in zip(solution, buyPrices)])
```

Finally you have to tell DEAP to run the constraint enforcing function after crossover and mutation, so:

```
toolbox.decorate('mate',   enforceBudget(money))
toolbox.decorate('mutate', enforceBudget(money))
```

This way the output of the program would be something like this:

```
499 profit= (4360.52,) spent= 49998.48000000001
investments= [53, 85, 40, 27, 30, 133, 41, 156, 13, 43]
```

You can find the complete source code of this program at the end of this document (Appendix D) and in file `investment_constraints.py`.

Appendix A. Basic Onemax Solution

```
import random
from deap import creator, base, tools, algorithms

creator.create('FitnessMax', base.Fitness, weights=(1.0,))
creator.create('Individual', list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()

toolbox.register('attrBool', random.randint, 0, 1)
toolbox.register('individual', tools.initRepeat, creator.Individual,
```

```
                toolbox.attrBool, n=50)
toolbox.register('population', tools.initRepeat, list, toolbox.individual)

def evalOneMax(individual):
    return sum(individual),      # COMMA HERE!!! - MUST BE A TUPLE

toolbox.register('evaluate', evalOneMax)
toolbox.register('mate', tools.cxTwoPoint)
toolbox.register('mutate', tools.mutFlipBit, indpb = 0.05)
toolbox.register('select', tools.selTournament, tournsize=3)

population = toolbox.population(100)

NGEN = 40

for gen in range(NGEN):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)
    fits      = toolbox.map(toolbox.evaluate, offspring)
    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

    # Print the best individual on each iteration with its objective value
    top = tools.selBest(population, k=1)
    print(gen, evalOneMax(top[0]), top[0])
```

Appendix B. Signal Regression Solution

```
import random
from deap import creator, base, tools, algorithms

creator.create('FitnessMin', base.Fitness, weights=(-1.0,))
creator.create('Individual', list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

toolbox.register('attrFloat', lambda: random.random()*2  - 1)  # Rands -1 to +1
toolbox.register('individual', tools.initRepeat, creator.Individual,
                toolbox.attrFloat, n=4)
toolbox.register('population', tools.initRepeat, list, toolbox.individual)



# Define x and objective function
```

```
import numpy

# Values in x on which the signal is defined
xVal = numpy.array([x*6.0/200 - 3 for x in range(200)])

# Read the signal data
yVal = numpy.loadtxt('signal.data')

# Define the signal simulation function (generic 3rd grade
# polynomial)
def polynomial(solution):
    a, b, c, d = solution
    return numpy.array([a*x**3 + b*x**2 + c*x + d for x in xVal])

#yVal = polynomial([-0.8, 0.7, 3.4, -1.6])

import math
from sklearn.metrics import mean_squared_error

def objective(solution):
    return math.sqrt(mean_squared_error(yVal,
                polynomial(solution)))/(max(yVal) - min(yVal)) ,  # FINAL COMMA!



toolbox.register('evaluate', objective)
toolbox.register('mate', tools.cxOnePoint)
toolbox.register('mutate', tools.mutGaussian, mu=0, sigma=0.5, indpb = 0.2)
toolbox.register('select', tools.selTournament, tournsize=3)


population = toolbox.population(40)

NGEN = 60

# Statistics
stats = tools.Statistics(key=lambda ind: ind.fitness.values)
stats.register('avg', numpy.mean)
stats.register('std', numpy.std)
stats.register('min', numpy.min)
stats.register('max', numpy.max)

logbook = tools.Logbook()

for gen in range(NGEN):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)
```

```
    fits      = toolbox.map(toolbox.evaluate, offspring)
    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

    # Stats
    record = stats.compile(population)
    print(record)

    logbook.record(gen=gen, **record)


# Gather the data collected and plot it
generation = logbook.select('gen')
fitnessMin = logbook.select('min')
fitnessAvg = logbook.select('avg')

import matplotlib.pyplot as plt

line1 = plt.plot(generation, fitnessMin, "b-", label="Minimum Fitness")
plt.xlabel("Generation")
plt.ylabel("Fitness (error)")

line2 = plt.plot(generation, fitnessAvg, "r-", label="Average Fitness")

lns = line1 + line2
labs = [l.get_label() for l in lns]
plt.legend(lns, labs, loc="center right")

plt.show()
```

Appendix C. Travelling Salesman Problem Solution

```
# -*- coding: utf-8 -*-
# Generate N cities (x,y)
import random, math

# In order to have the same problem on every run
random.seed(0)

def distance(cityA, cityB):
    return math.sqrt((cityA[0]-cityB[0])**2 + (cityA[1]-cityB[1])**2)
```

```
def generateCities(n):
    return [(random.randint(0,500), random.randint(0,500))
            for c in range(n)]

numCities = 30
cities    = generateCities(numCities)
print(cities)

# Configure DEAP
from deap import algorithms, base, creator, tools
import numpy

toolbox = base.Toolbox()
creator.create('FitnessMin', base.Fitness, weights=(-1.0,))
creator.create('Individual', list, fitness=creator.FitnessMin)

toolbox.register('Indices', numpy.random.permutation, len(cities))
toolbox.register('Individual', tools.initIterate, creator.Individual,
                 toolbox.Indices)
toolbox.register('Population', tools.initRepeat, list, toolbox.Individual)

toolbox.register('mate', tools.cxOrdered)
toolbox.register('mutate', tools.mutShuffleIndexes, indpb=0.05)

# Total distance traversed by a given solution
def totalDistance(individual):
    return sum((distance(cities[individual[i]], cities[individual[i-1]])
                for i in range(len(individual)))),


# Execute the GA
toolbox.register('evaluate', totalDistance)
toolbox.register('select', tools.selTournament, tournsize=3)

population = toolbox.Population(100)
NGEN = 100

for gen in range(NGEN):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)
    fits      = toolbox.map(toolbox.evaluate, offspring)
    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))
```

```
    top = tools.selBest(population, k=1)
    print(gen, totalDistance(top[0]), top[0])

print('='*40)
top10 = tools.selBest(population, k=10)
for ind in top10:
    print(totalDistance(ind), ind)
```

Appendix D. Constrained Investments Problem Solution

```
# -*- coding: utf-8 -*-

# Problem data
buyPrices  = [100, 80, 90, 130, 105, 60, 80, 40, 140, 110]
sellPrices = [112, 95, 99, 140, 113, 77, 88, 53, 146, 121]
money      = 50000


# GA setup: a list with as many integer, positive attributes as
# commodity types are there in the problem. Each number is the amount of
# units purchased of that commodity.
import random
from deap import creator, base, tools, algorithms

creator.create('FitnessMax', base.Fitness, weights=(1.0,))
creator.create('Individual', list, fitness = creator.FitnessMax)

toolbox = base.Toolbox()

# Initially amounts purchased are 0..100, but on mutation they can be 0..200
# (so that initial population individuals are valid)
toolbox.register('attrInt', lambda: random.randint(0,100))
toolbox.register('individual', tools.initRepeat, creator.Individual,
                 toolbox.attrInt, n=len(buyPrices))

toolbox.register('population', tools.initRepeat, list, toolbox.individual)


# Objective function: compute total profit from amounts purchased.
# Remember the final comma (result must be a tuple)
def objective(solution):
    return sum([amt*(sp - bp*(1+amt/1000))
                for amt, bp, sp in zip(solution, buyPrices, sellPrices)]),
```

```
# Auxiliary function to compute the money spent on a given solution
def spent(solution):
    return sum([amt*bp*(1+amt/1000) for amt, bp in zip(solution, buyPrices)])


# Decorator function required to enforce individuals to limit the money spent.
# It removes 1 unit purchased (randomly) until the budget constraint is fulfilled
def enforceBudget(budget):
    def decorator(func):
        def wrapper(*args, **kargs):
            offspring = func(*args, **kargs)
            for child in offspring:
                while spent(child) > budget:
                    pos = random.randint(0, len(child)-1)
                    # It is critical to avoid negative attributes
                    child[pos] = max(child[pos]-1, 0)
            return offspring
        return wrapper
    return decorator


# Set up the GA elements
toolbox.register('evaluate', objective)
toolbox.register('mate', tools.cxOnePoint)
toolbox.register('mutate', tools.mutUniformInt, low=0, up=200, indpb=0.1)
toolbox.register('select', tools.selTournament, tournsize=3)

# Tell DEAP to check budget use after mutations and crossovers
toolbox.decorate('mate',   enforceBudget(money))
toolbox.decorate('mutate', enforceBudget(money))

# Create the population and run the GA
population = toolbox.population(100)
NGEN = 500

for gen in range(NGEN):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)
    fits      = toolbox.map(toolbox.evaluate, offspring)

    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

    # Display info on each iteration
```

```
top = tools.selBest(population, k=1)
print(gen,'profit=',objective(top[0]),'spent=',spent(top[0]),'investments=',top[0])
```