DEAP: **D**ISTRIBUTED **E**VOLUTIONARY **A**LGORITHMS IN **P**YTHON

DISTRIBUTED
EVOLUTIONARY
ALGORITHMS IN
PYTHON

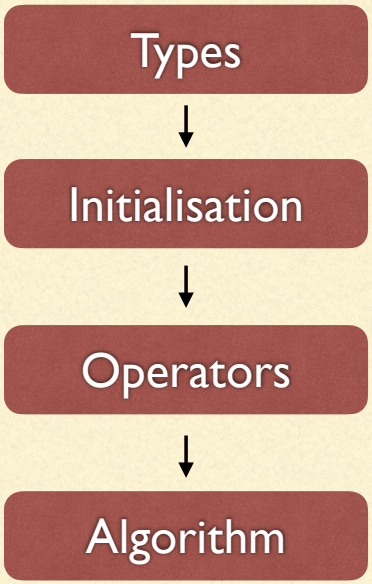
- An evolutionary computation framework for Python

  Enables developers to define their own types, initialisers, operators, and algorithms!

https://github.com/deap/deap

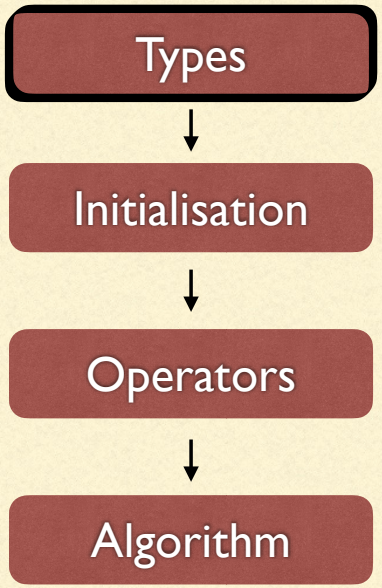
DEAP: **D**ISTRIBUTED **E**VOLUTIONARY **A**LGORITHMS IN **P**YTHON

- Four main steps of DEAP



DEAP: **D**ISTRIBUTED **E**VOLUTIONARY **A**LGORITHMS IN **P**YTHON

- Four main steps of DEAP



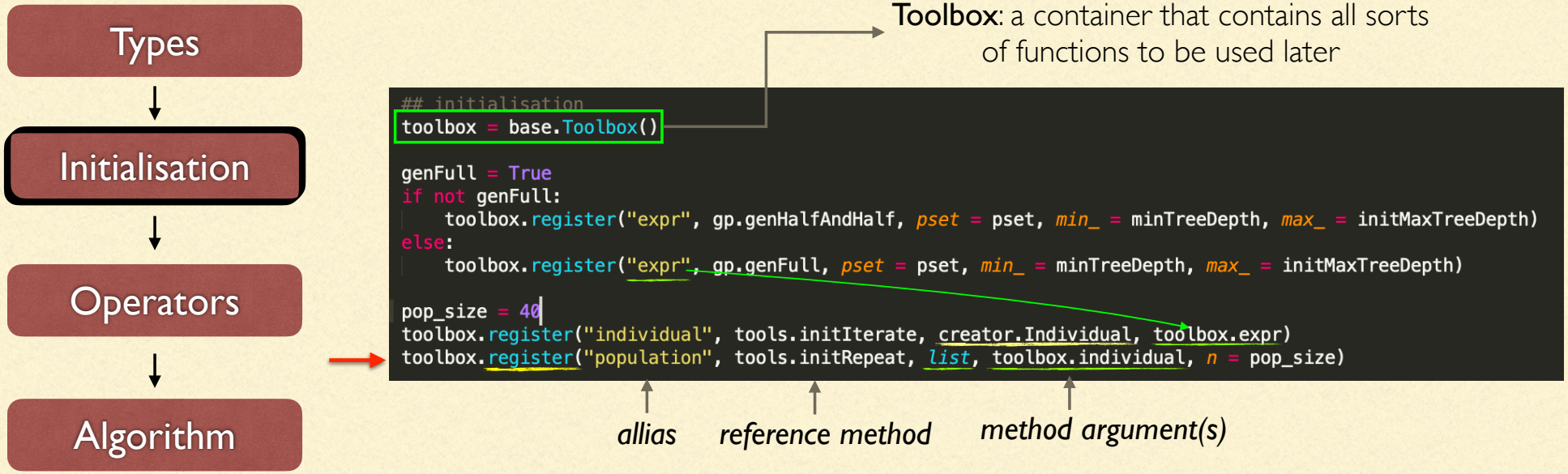
creator: allow to create class that suites the user's need

```
## create types
creator.create("FitnessMax", self.base.Fitness, weights = (1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness = creator.Fitness, pset = pset)
```

allias *base class* *attribute(s)*

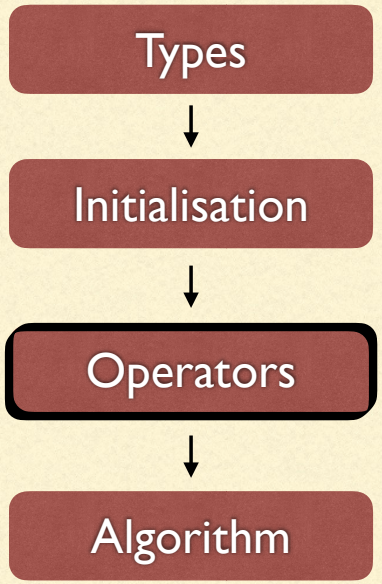
DEAP: **D**ISTRIBUTED **E**VOLUTIONARY **A**LGORITHMS IN **P**YTHON

- Four main steps of DEAP



DEAP: **D**ISTRIBUTED **E**VOLUTIONARY **A**LGORITHMS IN **P**YTHON

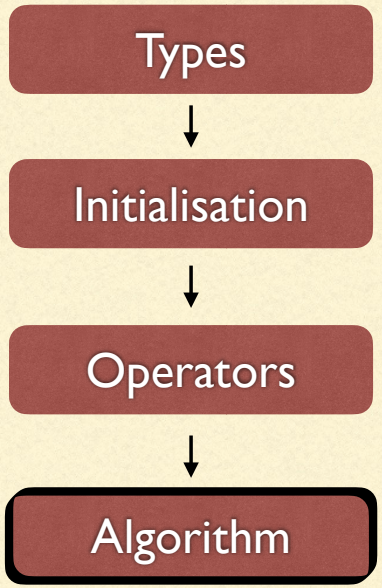
- Four main steps of DEAP




```
## operators
toolbox.register("compile", gp.compile, pset = pset)
toolbox.register("evaluate", self.eval_func)

# for the parent selection
tournsize = int(pop_size * 0.2) if int(pop_size * 0.2) % 2 == 0 else int(pop_size * 0.2) + 1
toolbox.register("select", tools.selTournament, tournsize = tournsize, fit_attr = "fitness")

toolbox.register("mate", gp.cxOnePoint)
toolbox.register("mutate", gp.mutUniform, expr = toolbox.expr, pset = pset)
```

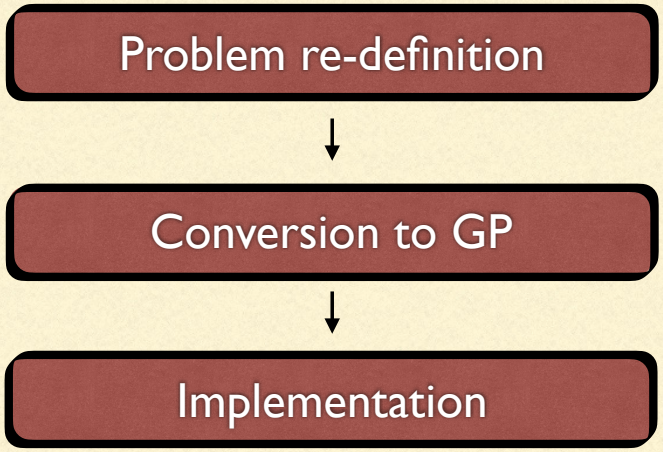
DEAP: **D**ISTRIBUTED **E**VOLUTIONARY **A**LGORITHMS IN **P**YTHON

- Four main steps of DEAP




```
# generate the initial population
pop = toolbox.population(n = pop_size)
fitness_per_ind = toolbox.map(toolbox.evaluate, pop)
for fit, ind in zip(fitness_per_ind, pop):
    ind.fitness.values = fit

# contains the best individual ever lived in the population during the evolution
num_best = 1
best = tools.HallOfFame(num_best, similar = self.check_duplicate)
best.update(pop)

# main algorithm
ngen = 100
for idx_to_gen in tqdm(range(1, ngen + 1)):
    next_pop = []
    while len(next_pop) < pop_size:
        # select a pair of individuals that will become the parent
        parents = toolbox.select(pop, 2)
        offsprings = algorithms.varAnd(parents, toolbox, cxpb, mutpb)
        for offspring in offsprings:
            if len(next_pop) == 0 or not self.check_duplicate(offspring, next_pop):
                next_pop.append(offspring)

    #select & evaluate the offspring
    fitness_per_ind = toolbox.map(toolbox.evaluate, next_pop)
    for fit, ind in zip(fitness_per_ind, next_pop):
        ind.fitness.values = fit

    # select the nex population
    pop[:]= tools.selBest(pop + next_pop, pop_size)
    best.update(pop)
```

CONVERT TO A GENETIC PROGRAMMING PROBLEM



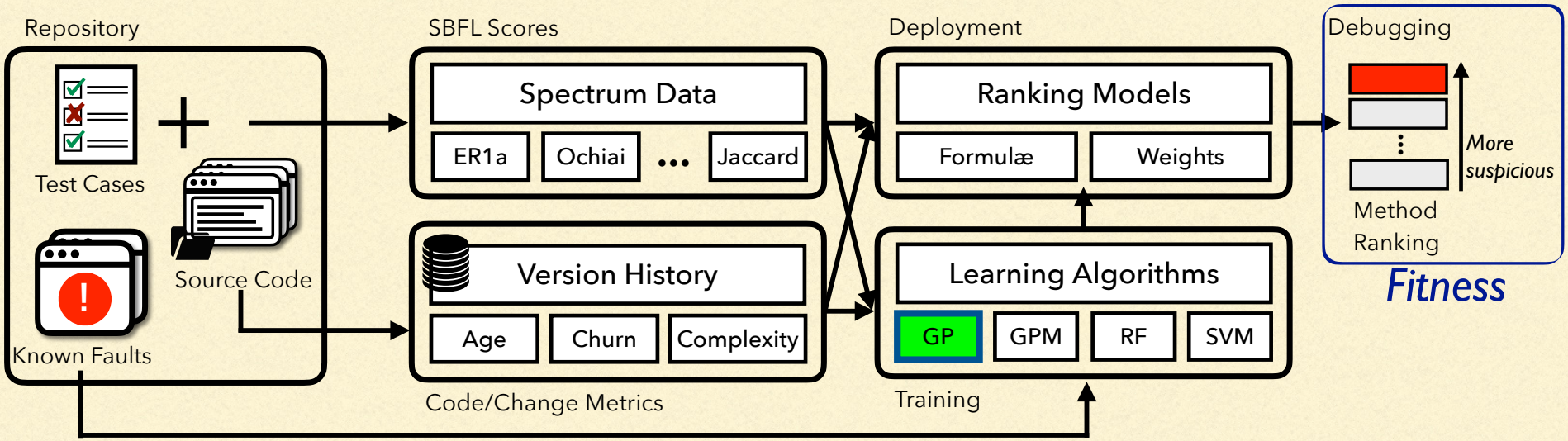
=> redefine to a search problem

=> Define fitness, operators, operands, etc.

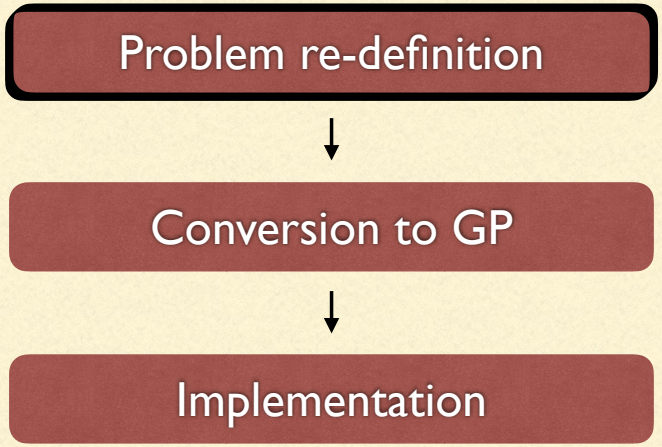
CONVERSION TO A GP PROBLEM (IN FLUCCS)

GENETIC PROGRAMMING IN FLUCCS

FLUCCS: Using Code and Change Metrics to Improve Fault Localization



GENETIC PROGRAMMING IN FLUCCS

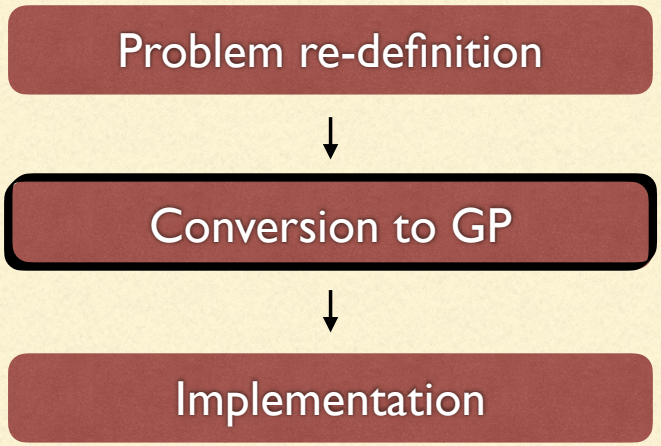


Generate an effective fault localisation model

↓

Search for an effective fault localisation model among the space of all fault localisation models through the evolution

GENETIC PROGRAMMING IN FLUCCS

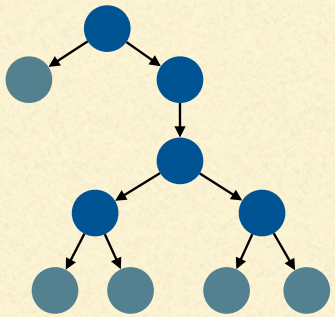


1. *Decide how we represent a fault localisation model (individual)*
2. *Define a fitness function to guide the evolution*

GENETIC PROGRAMMING IN FLUCCS

Conversion to GP

1. *Decide how we represent a fault localisation model (individual)*



primitive (GP operators)
: terminal

GP operators: addition, subtraction, multiplication, division, negation, square root

Terminals: 40 features (33 SBFL scores & 7 code and change metric features)

→ *Primitiv*

Strongly Typed Primitive Set

```
sbfl_scores = ["ochiai", "jaccard", "gp13", "wong1", "wong2", "wong3",\
                    "tarantula", "ample", "RussellRao", "SorensenDice", "Kulczynski1",\
                    "SimpleMatching", "M1", "RogersTanimoto", "Hamming", "Ochiai2",\
                    "Hamann", "dice", "Kulczynski2", "Sokal", "M2", "Goodman",\
                    "Euclid", "Anderberg", "Zoltar", "ER1a", "ER1b", "ER5a", "ER5b",\
                    "ER5c", "gp02", "gp03", "gp19"]

ccmetrics = ["churn", "max_age", "min_age","num_args", "num_vars", "b_length", "loc"]
features = sbfl_scores + ccmetrics
```

```
def gen_primitives(features):
    pset = gp.PrimitiveSetTyped('main', [float] * len(features), float)

    pset.addPrimitive(np.add, [float,float], float, name = "gp_add")
    pset.addPrimitive(np.subtract, [float,float], float, name = "gp_sub")
    pset.addPrimitive(np.multiply, [float,float], float, name = "gp_mul")
    pset.addPrimitive(np.negative, [float], float, name = "gp_neg")
    pset.addPrimitive(protect_div, [float, float], float, name = "gp_div")
    pset.addPrimitive(protect_sqrt, [float], float, name = "gp_sqrt")

    pset.addTerminal(1.0, float)
```

```
pset = gen_primitives(features)
## create types
creator.create("Fitness", base.Fitness, weights = (-1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness = creator.Fitness, pset = pset)
```

GENETIC PROGRAMMING IN FLUCCS

Conversion to GP

2. Define a fitness function to guide the evolution

→*Minimisation*

Fitness function (*the effectiveness of* τ):

$$F(\tau, B, p) = \frac{1}{|B|} \sum_{b} wef(\tau, b, p)$$

```python
def evaluate(individual, data_df, features):
    num_data = len(data_df)
    vs = np.float32(data_df.feature.values.tolist())
    sps = np.zeros(num_data, dtype = np.float32)

    _features = dict()
    for featureIdx, featureName in enumerate(features):
        _features[featureName] = vs[:,featureIdx]
        individual = individual.replace(featureName, "_features['" + featureName + "'][idx]")

    for idx in range(num_data):
        sps[idx] = -eval(individual)

    return sps
```

```python
def eval_func(individual, data_df_lst, features):
    from scipy.stats import rankdata

    fitness_values = []
    for i, data_df in enumerate(data_df_lst):
        fault_indices = data_df.index[data_df.label == 1].values

        core_values = evaluate(str(individual), data_df, features)
```

```python
pset = gen_primitives(features)
## create types
creator.create("Fitness", base.Fitness, weights = (-1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness = creator.Fitness, pset = pset)
```

GENETIC PROGRAMMING IN FLUCCS

Problem re-definition

↓

Conversion to GP

↓

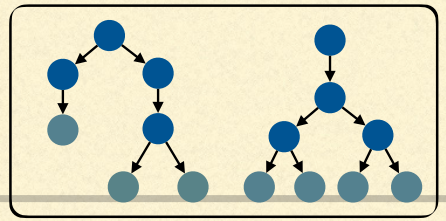
Implementation

Initilisation

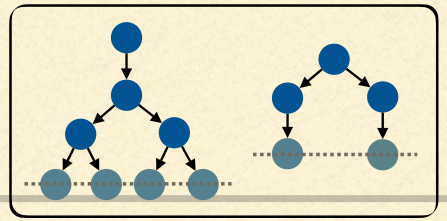
```
## initialisation
minTreeDepth = 1; maxTreeDepth = 8; initMaxTreeDepth = 6
pop_size = 40; num_best = 1
cxpb, mutpb = 1.0, 0.1
ngen = 100
genFull = True
```

```
toolbox = base.Toolbox()
if not genFull:
    toolbox.register("expr", gp.genHalfAndHalf, pset = pset, min_ = minTreeDepth, max_ = initMaxTreeDepth)
else:
    toolbox.register("expr", gp.genFull, pset = pset, min_ = minTreeDepth, max_ = initMaxTreeDepth)

pop_size = 40
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr)
toolbox.register("population", tools.initRepeat, list, toolbox.individual, n = pop_size)
```

genHalfAndHalf



genFull



GENETIC PROGRAMMING IN FLUCCS

Problem re-definition

↓

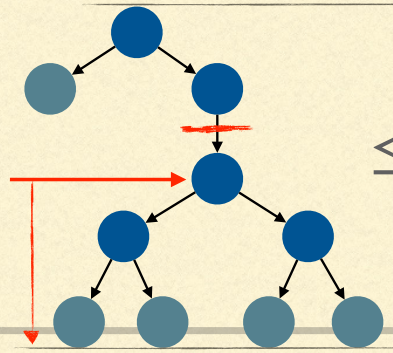
Conversion to GP

↓

Implementation

Operator Registration

```
## operators
toolbox.register("evaluate", eval_func, data_df_lst = data_df_lst, features = features)

# for the parent selection
tournsize = int(pop_size * 0.2) if int(pop_size * 0.2) % 2 == 0 else int(pop_size * 0.2) + 1
toolbox.register("select", tools.selTournament, tournsize = tournsize, fit_attr = "fitness")

toolbox.register("mate", gp.cxOnePoint)
toolbox.register("mutate", gp.mutUniform, expr = toolbox.expr, pset = pset)

# decorate Individual(PrimitiveTree).height with max_value
# for bloat control
toolbox.decorate("mate", gp.staticLimit(key = operator.attrgetter("height"), max_value = maxTreeDepth))
toolbox.decorate("mate", gp.staticLimit(key = operator.attrgetter("height"), max_value = maxTreeDepth))
```

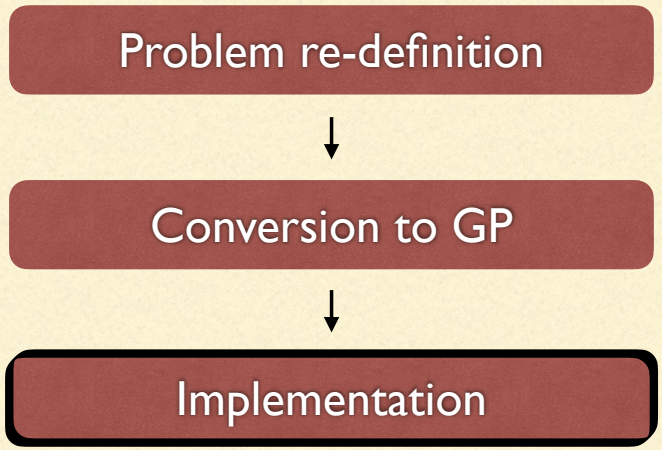


$\leq maxTreeDep$
$= 8$

Logging

```
# set Statistics
stats = tools.Statistics(lambda ind: ind.fitness.values[0])
stats.register("average", np.mean, axis = 0)
stats.register("max", np.max, axis = 0)
stats.register("min", np.min, axis = 0)

logbook = tools.Logbook()
logbook.header = ['gen', 'evals'] + stats.fields
```

GENETIC PROGRAMMING IN FLUCCS



Generate and evaluate an initial population

```
# generate the initial population
pop = toolbox.population(n = pop_size)
fitness_per_ind = toolbox.map(toolbox.evaluate, pop)
for fit, ind in zip(fitness_per_ind, pop):
    ind.fitness.values = fit

# contains the best individual ever lived in the populuation during the evolution
best = tools.HallOfFame(num_best, similar = check_duplicate)
best.update(pop)
```

GENETIC PROGRAMMING IN FLUCCS

Problem re-definition

↓

Conversion to GP

↓

Implementation

Generate & evaluate the offspring population

Compose the population of the next generation

Status logging

Check the population convergence

```
# main algorithm
for idx_to_gen in range(1, ngen + 1):
    next_pop = []

    while len(next_pop) < pop_size:
        # select a pair of individuals that will become the parent
        parents = toolbox.select(pop, 2)
        offsprings = algorithms.varAnd(parents, toolbox, cxpb, mutpb)
        for offspring in offsprings:
            if len(next_pop) == 0 or not check_duplicate(offspring, next_pop):
                next_pop.append(offspring)

    #evaluate the fitness for the offspring
    fitness_per_ind = toolbox.map(toolbox.evaluate, next_pop)
    #update next_pop, with invalid fitness, with the new(valid) fitness
    for fit, ind in zip(fitness_per_ind, next_pop):
        ind.fitness.values = fit

    # select the nex population
    pop[:]= tools.selBest(pop + next_pop, pop_size)
    #update current best for this new pop
    best.update(pop)

    #Logging current status the pop
    stats_result = stats.compile(pop)
    print ("\t\tGeneration {}: {}(Max), {}(AVG), {}(MIN)".format(
        idx_to_gen, stats_result["max"], stats_result["average"], stats_result["min"]))

    # logging
    logbook.record(gen = idx_to_gen, evals = len(pop), **stats_result)
    print (logbook)

    # check population
    num_same = 0
    for ind_1, ind_2 in zip(pop[::2], pop[1::2]):
        num_same += int(check_duplicate(ind_1, ind_2))
    if num_same == pop_size:
        print ("All individuals in the population are the same")
        break
```

GENETIC PROGRAMMING IN FLUCCSS

Problem re-definition

↓

Conversion to GP

↓

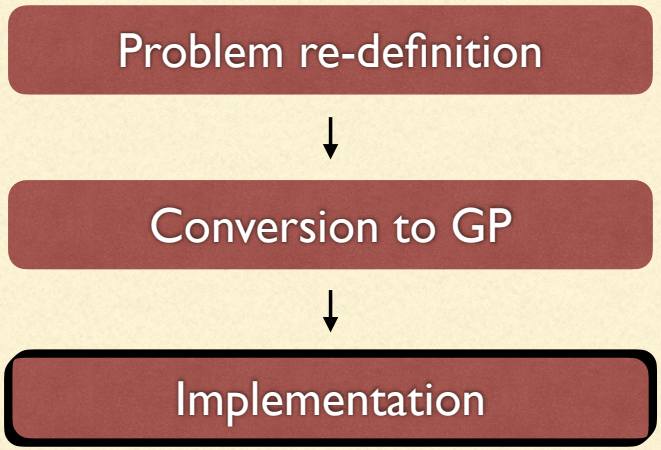
Implementation

Generate & evaluate the offspring population

```
# main algorithm
for idx_to_gen in range(1, ngen + 1):
    next_pop = []

    while len(next_pop) < pop_size:
        # select a pair of individuals that will become the parent
        parents = toolbox.select(pop, 2)
        offsprings = algorithms.varAnd(parents, toolbox, cxpb, mutpb)
        for offspring in offsprings:
            if len(next_pop) == 0 or not check_duplicate(offspring, next_pop):
                next_pop.append(offspring)

    #evaluate the fitness for the offspring
    fitness_per_ind = toolbox.map(toolbox.evaluate, next_pop)
    #update next_pop, with invalid fitness, with the new(valid) fitness
    for fit, ind in zip(fitness_per_ind, next_pop):
        ind.fitness.values = fit
```

varAnd: *apply both crossover and mutation on individuals*

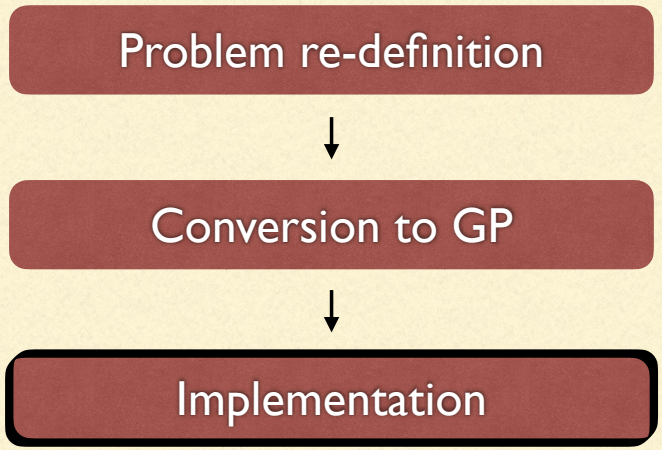
GENETIC PROGRAMMING IN FLUCCS



Compose the population of the next generation (+ update the set of best solutions)

```
# select the nex population
pop[:]= tools.selBest(pop + next_pop, pop_size)
#update current best for this new pop
best.update(pop)
```

GENETIC PROGRAMMING IN FLUCCS



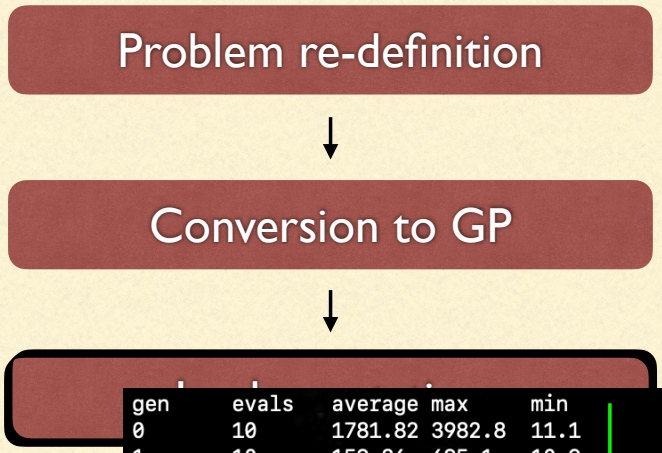
Status logging

```
#Logging current status the pop
stats_result = stats.compile(pop)
print ("\t\tGeneration {}: {}(Max), {}(AVG), {}(MIN)".format(
    idx_to_gen, stats_result["max"], stats_result["average"], stats_result["min"]))

# logging
logbook.record(gen = idx_to_gen, evals = len(pop), **stats_result)
print (logbook)
```

```
                Generation 3: 11.0(Max), 10.179999923706054(AVG), 8.5(MIN)
gen     evals   average max     min
0       10      1781.82 3982.8  11.1
1       10      159.86  695.1   10.9
2       10      10.71   11.2    8.5
3       10      10.18   11      8.5
```

At the third generation

GENETIC PROGRAMMING IN FLUCCS



Check the population convergence

```
# check population
num_same = 0
for ind_1, ind_2 in zip(pop[::2], pop[1::2]):
    num_same += int(check_duplicate(ind_1, ind_2))
if num_same == pop_size:
    print ("All individuals in the population are the same")
    break
```

```
gen     evals   average max     min
0       10      1781.82 3982.8  11.1
1       10      159.86  695.1   10.9
2       10      10.71   11.2    8.5
3       10      10.18   11      8.5
4       10      9.67    10.9    7
5       10      8.89    10.2    7

The best individual
                expression : gp_sub(gp_mul(gp_sqrt(gp_sqrt(gp_mul(gp13, SorensenDice))), gp_add(gp_neg(churn), gp_sqrt(b_length))), gp_neg(Anderberg))
                fitness    : 7.0
```