```
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from random import choice, sample
from math import ceil, inf
from inspect import isfunction, ismethod
from re import match
from operator import itemgetter

from lib.eventdispatcher import EventDispatcher
from .ProcessusState import ProcessusState
from .event_names import *


class Generator:
    """Handle the generation proccess

    The generator, at the heart of the generation process, has three charges:
        - create a population of individuals
        - select a subset of the population, based on their performances
        - breed individuals of the selection to form a new population
    Individuals are represented by root GeneticElement instances.
    Use a Graduator to grade performances.
    Extending it is strongly adviced.

    The generator dispatches several events through its internal dispatcher:
        processus.start,
        processus.done,
        creation.start,
        creation.done,
        generation.start,
        generation.done,
        grading.start,
        grading.process,
        grading.done,
        selection.start,
        selection.done,
        breeding.start,
        breeding.process,
        breeding.done,
    and processus.resume
    See ProcessusState.py for informations carried by events.
    In particular, the population is available through creation.done and
    generation.start/done.
    """


    def __init__(self, factory, graduator, listeners = [], end_statement = None):
        """Init

        Expects:
            factory to be a class inheriting of GeneticElementFactory
            graduator to be a instance inheriting of Graduator
            listeners to be a list of listeners (see below)
            end_statement to be a boolean function
        Listeners can be:
            - couples (event_name, listener)
            - tuples (event_name, listener, priority)
            - functions and methods if their names follow the format
              'onEventName'. For example, listener 'onProcessusStart' will
              listen on 'processus.start'.
              If they have a priority attribute, it will be used as priority.
            - objects: every method following the format above is added to
            listeners.
        factory and graduator are automatically added to listeners.
        Priorities has to be strictly smaller than 1000.
        """

        self.factory = factory
        self.graduator = graduator
        self.end_statement = end_statement

        self.state = ProcessusState()
        self.iterating = False

        self.dispatcher = EventDispatcher()
        listeners.append(factory)
        listeners.append(graduator)

        listeners.extend([
            (PROCESSUS.START, self.create, 1000),
            (CREATION.DONE, self.initGeneration, 1000),
            (GENERATION.START, self.grade, 1000),
            (GRADING.DONE, self.select, 1000),
            (SELECTION.DONE, self.breed, 1000),
            (BREEDING.DONE, self.endGeneration, 1000),
        ])

        # Get all objects' methods
        listenersMethods = listeners.copy()
        for listener in listeners:
            if not (type(listener) is tuple or isfunction(listener) or ismethod(listener)):
                listenersMethods.remove(listener)
                for method in [method for method in dir(listener) if ismethod(getattr(listener, method))]:
                    if match('on([A-Z]\w+)', method):
                        listenersMethods.append(getattr(listener, method))

        # Inscribe all listeners
        for listener in listenersMethods:
```

```
            if type(listener) is tuple:
                self.dispatcher.listen(*listener)
            else:
                # Parse method names to get event names
                m = match('on([A-Z]\w+)', listener.__name__)
                if m:
                    event_name = ''
                    camel_event_name = m.group(1)
                    while True:
                        m = match('([A-Z][a-z0-9_]+)(\w*)', camel_event_name)
                        if not m:
                            break
                        if event_name:
                            event_name += '.'
                        event_name += m.group(1).lower()
                        camel_event_name = m.group(2)
                    self.dispatcher.listen(event_name, listener,
                        0 if not hasattr(listener, 'priority') else listener.priority)
                else:
                    raise ValueError('The given listener do not follow the format onEventName.')


    def dispatch(self, event_name):
        self.state.event_name = event_name
        self.dispatcher.dispatch(event_name, self.state)

    def dispatchGrading(self, individual, graduation):
        """Shorthand to dispatch grading events"""
        self.state.individual = individual
        self.state.graduation = graduation
        self.dispatch(GRADING.PROGRESS)


    def initProcessus(self):
        self.dispatch(PROCESSUS.START)

    def endProcessus(self):
        self.dispatch(PROCESSUS.DONE)

    def initGeneration(self, state):
        """Handle iteration"""
        self.iterating = True

        try:
            while True:
                state.generation_id += 1
                self.dispatch(GENERATION.START)
        except StopIteration:
            pass

        self.iterating = False

    def endGeneration(self, state):
        self.dispatch(GENERATION.DONE)
        if (
            state.generation_id >= state.generations
            or (state.generations == inf and self.end_statement(state))
        ):
            self.endProcessus()
            if self.iterating:
                raise StopIteration

        elif not self.iterating:
            self.initGeneration(state)


    def create(self, state):
        """Generate a whole initial population"""

        state.generation_id = 0
        self.dispatch(CREATION.START)

        state.population = set([self.factory.create() for i in range(state.pop_length)])

        self.dispatch(CREATION.DONE)


    def resumeGrading(self, state):
        """Grade non-graded individuals"""

        graded_individuals = set([individual for score, individual in state.grading])
        to_grade = state.population.difference(graded_individuals)

        state.grading.extend(
            self.graduator.gradeAll(to_grade, state.generation_id, self.dispatchGrading)
        )
        state.grading.sort(key=itemgetter(0), reverse=True)

        self.dispatch(GRADING.DONE)


    def grade(self, state):
        """Grade all individuals"""

        self.dispatch(GRADING.START)

        state.grading = []
        self.resumeGrading(state)


    def select(self, state):
```

```
        """Operate the selection

        This is a basic system to be overcome.
        The selection is a subset of the population.
        """

        self.dispatch(SELECTION.START)

        # Get a list of individuals
        ordered_individuals = [c[1] for c in state.grading]

        # The number of individuals to select
        selection_length = ceil(len(state.population) * state.proportion)
        # Among the [selection_length] best individuals select selection_length*(1-state.chance) ones
        selection = set(sample(
            ordered_individuals[:selection_length],
            int(selection_length * (1 - state.chance))
        ))
        # Complete selection with random individuals
        unused_individuals = state.population.difference(selection)
        while len(selection) < selection_length:
            choiced = choice(list(unused_individuals))
            selection.add(choiced)
            unused_individuals.remove(choiced)

        state.selection = selection
        self.dispatch(SELECTION.DONE)


    def breed(self, state):
        """Generate a new population based on selection

        This is a basic system to be overcome.
        """

        self.dispatch(BREEDING.START)

        new_pop = set()

        # Add artificially the best individual to the new pop : survival principle
        best = state.grading[0][1]
        new_pop.add(best)
        state.offspring = best
        state.parents = (best, best)
        self.dispatch(BREEDING.PROGRESS)

        while len(new_pop) < state.pop_length:
            parents = tuple([choice(list(state.selection)) for i in range(2)])
            offspring = self.factory.breed(*parents)
            new_pop.add(offspring)

            state.offspring = offspring
            state.parents = parents
            self.dispatch(BREEDING.PROGRESS)

        state.population = new_pop

        self.dispatch(BREEDING.DONE)


    def process(self, processus_id, generations, pop_length = 500, proportion = .5, chance = 0):
        """Process multiple generations

        If generations == inf then self.end_statement will be the stopping statement.

        Expects:
            generations to be an int or inf
            pop_length to be an int

            proportion to be a float between 0 and 1
            chance to be a float between 0 and 1

        Return the last generation
        """

        self.state.processus_id = processus_id
        self.state.generations = generations
        self.state.pop_length = pop_length
        self.state.proportion = proportion
        self.state.chance = chance

        self.initProcessus()

        return self.state.population


    def resume(self, state):
        """Resume a stopped processus """

        self.dispatcher.dispatch(PROCESSUS.RESUME, state)

        self.state = state
        if state.event_name in (
            PROCESSUS.START, CREATION.DONE, GENERATION.START, GRADING.DONE, SELECTION.DONE, BREEDING.DONE
        ):
            self.dispatch(self.state.event_name)
        elif state.event_name == CREATION.START:
            self.create(state)
        elif state.event_name == GRADING.START:
            self.grade(state)
        elif state.event_name == GRADING.PROGRESS:
```

```
302                 self.resumeGrading(state)
303             elif state.event_name == SELECTION.START:
304                 self.select(state)
305             elif state.event_name in (BREEDING.START, BREEDING.PROGRESS):
306                 self.breed(state)
307             elif state.event_name == GENERATION.DONE:
308                 self.endGeneration(state)
309             elif state.event_name == PROCESSUS.DONE:
310                 self.endProcessus()
311             else:
312                 raise ValueError(state.event_name + 'is not handled.')
313 
314             return self.state.population
```