Improved Implementation of Digital Organism Simulation Environment (DOSE Version 1.0.4)

BY

Clarence Fitzgerald Gumtang Castillo
Maurice Han Tong Ling

2015
Colossus Technologies LLP
Technical Report Number 001

Colossus Technologies LLP Technical Report No. 001

Improved Implementation of Digital Organism Simulation Environment (DOSE Version 1.0.4)

Clarence FG Castillo[1,2,5], Maurice HT Ling[3,4,6]
[1] Colossus Technologies LLP, Singapore
[2] School of Information Technology, Republic Polytechnic, Singapore
[3] School of BioSciences, The University of Melbourne, Australia
[4] clarence.castillo_33@yahoo.com; [5] mauriceling@acm.org

Abstract

Evolution is a fundamental aspect of biology but examining evolution is difficult and costly. Artificial life simulations via the use of digital organisms (DO) had been proposed as a feasible means of examining evolution *in silico* and had yield biologically relevant findings. Recently, original implementation of DOSE (Ling, 2012a) had been improved (Castillo and Ling, 2014a) for use as a Python library for simplified construction of simulation, enabling database logging and revival of simulations. This manuscript documents the implementation and improvement of DOSE, which is released as DOSE version 1.0.4 (http://github.com/mauriceling/dose/release/tag/v1.0.4) and licensed under GNU General Public License version 3. DOSE codebase is hosted and available for forking at http://github.com/mauriceling/dose.

Citation: Castillo, CFG, Ling, MHT. 2015. Improved Implementation of Digital Organism Simulation Environment (DOSE Version 1.0.4). Colossus Technologies LLP Technical Report Number 001.

Table of Contents

Introduction

Evolution is a fundamental aspect of biology. However, testing evolutionary hypotheses is a challenge (Batut et al., 2013) as it is highly time consuming and expensive, if not impossible. Christopher Langton (1986) had conceptualized that by casting chemical reactions, reactants, and products into computable operations, operands, and outputs respectively, it may be possible to simulate artificial life (organisms in the digital world, or digital organisms) as cellular automata “living” on artificial chemistries. Batut et al. (2013) argue that artificial life or digital organisms (DO) is a valuable tool to enable experimental evolution despite its drawbacks as repeated simulations can be carried out with recording of all events. Bersini (2009) argued that artificial life and theoretical biology shared many common grounds and presented genetic algorithm (GA) as an important model to bridge the two fields. GA was used to study microbial genetics and evolution (Watson, 2012).

A recent Python implementation of GA, grounded on the biological hierarchy from genome to organisms to population (Lim et al., 2010), had been used as a basis to develop into a simulation framework for digital organisms, known as Digital Organisms Simulation Environment (DOSE) (Ling, 2012a). The genetic material of DOSE organisms is an executable DNA based on a parameterless 3-character instruction code, known as Ragaraja (Ling, 2012b), which Ling (2012a) argued that such an instruction set mimics the naturally occurring DNA.

Recently, the original implementation of DOSE (Ling, 2012a) had been improved (Castillo and Ling, 2014a) for use as a Python library for simplified construction of simulation, enabling database logging and revival of simulations. A utility case study of DOSE demonstrated that adjacent migration, such as foraging or nomadic behavior, increases heterozygosity (local genetic distance) while long distance migration, such as flight covering the entire ecosystem, does not increase heterozygosity. These results are consistent with previous studies (Relethford, 1988, Raymond et al., 2013). In addition, DOSE had been used to simulate antibiotics resistance (Castillo and Ling, 2014b, Castillo et al., 2015). This manuscript documents the implementation of DOSE version 1.0.4 (http://github.com/ mauriceling/dose/release/tag/v1.0.4) which is licensed under GNU General Public License version 3, except for COPADS (Collection of Python Algorithms and Data Structures, which is released under Python Software Foundation License version 2). DOSE codebase is hosted and available for forking at http://github.com/ mauriceling/dose.

The rest of the manuscript is organized as follows. Section 2 provides a brief code description of DOSE version 1.0.4 (for detailed description, please see Lim et al., 2010; Ling, 2012a; Ling, 2012b; Castillo and Ling, 2014a). Section 3 presents all 5 new and improved codes of DOSE; namely, *dose.py*, *simulation_calls.py*, *database_calls.py*, *dose_world.py* (bug fixed; original version in Ling, 2012a) and *genetic.py* (original version in Lim et al., 2010). Two other code files (original version in Ling, 2012b); namely, *register_machine.py* and *ragaraja.py*; were unchanged from the original version and will not be presented. Section 4 presents the simulation implementations for the first case study described in Castillo and Ling (2014a), which examined the effects of different migration schemes on genetic heterozygosity (local genetic distance). Section 5 presents an example of reviving a simulation from Section 4 from the database file and continuing the simulation for another 200 generations. An appendix on how to write a simulation is provided.

Code Description

DOSE version 1.0.4 (Castillo and Ling, 2014a) is based on previous implementations of DOSE (Ling, 2012a; Ling, 2012b), which used a previous implementation of GA (*genetic.py*; Lim et al., 2010). The architecture of DOSE version 1.0.4 is given in Figure 1.

The most important enhancement to the original implementation of DOSE (Ling, 2012a) is the ability for the simulations to log all events into a SQLite database and subsequently, the ability to revive a simulation from the logged records and continuing the simulation. This is achieved in *database_calls.py* by logging all data attributes of each organism and the entire ecological system into the SQLite database, as well as re-creating all simulation objects (organisms and ecosystem) at a specific generation from the database. Other enhancements (in GA library; *genetic.py*; original version in Lim et al., 2010) are as follows: firstly, each organism is given a 32-character randomly generated name at initialization to aid in ancestry tracking. Secondly, chromosomes will inherit parental background mutation after crossover. Lastly, the statuses of each organism are increased from 6 in Lim et al. (2012) to 12. New statuses are

- parents (identity of parents),
- gender (gender of organism),
- blood (result of genomic interpretation or expression by Rajaraga interpreter),
- identity (32-character randomly generated name),
- deme (defined as a sub-population or local population), and
- location (location of the organism within the ecosystem).

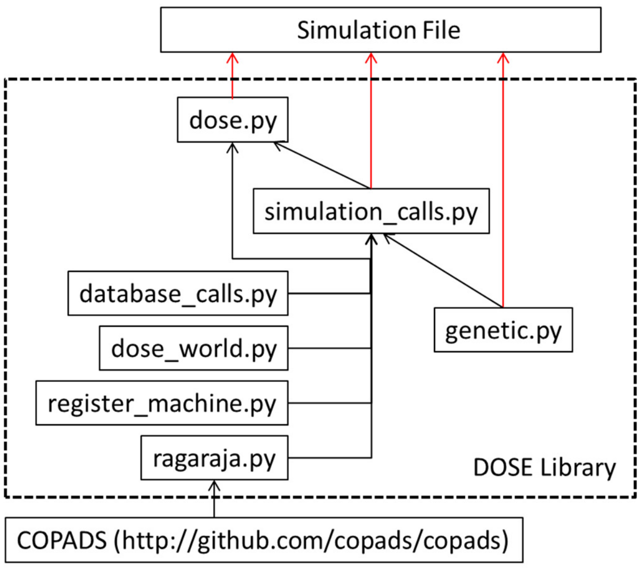

Figure 1. Architecture of DOSE Library. DOSE consists of 7 main files. Four of the files; *database_calls.py*, *dose_world.py*, *register_machine.py* and *ragaraja.py*; were used by *dose.py* and *simulation_calls.py*. *genetic.py* provides the GA for simulation. COPADS (released under Python Software Foundation License version 2) is required by Rajaraja interpreter (*ragaraja.py*; Ling, 2012b). Simulation File represents the simulation implementation, such as those in Sections 4 and 5 and requires *dose.py* and *simulation_calls.py*. *genetic.py*.

The main application programming interface (API) is provided by *dose.py*, which contains the main functions needed to write a simulation. Once all needed simulation parameters and functions are defined (for details, please see Castillo and Ling, 2014a), *dose.simulate* function is called (see examples in Section 4). This will in turn call functions in *simulation_calls.py* prepare the simulation (Figure 2) by constructing, initializing the DOs, and mapping each DO onto the world. Simulation preparation is dependent on whether the simulation is a new simulation (regular simulation) or revival and continuation of an existing simulation (simulation revival) from database or files. For regular simulation, if the initial chromosome/genome is given, it will be used for organism contruction; otherwise, a chromosome of single base will be used. The constructed organisms (population(s)) will be deployed accordingly before running the simulation (simulation core in Figure 2). On the other hand, a simulation can only be revived if either the simulation log files or the simulation results database with all required parameters and attributes logged are present. These will be used to reconstruct the state of each organism, the world, and mapping each organism to the ecosystem before running the simulation.

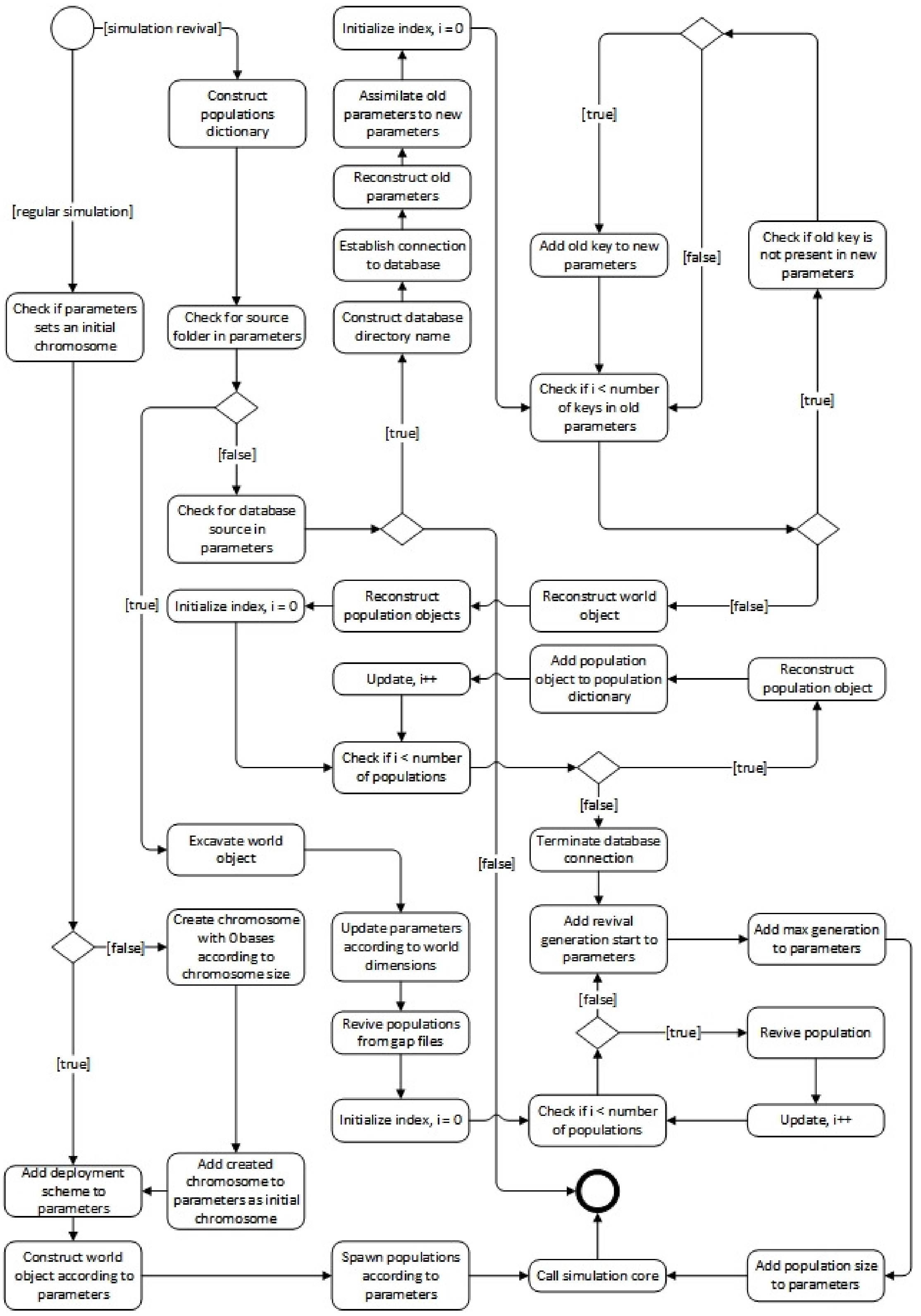
[simulation revival]
[regular simulation]
Construct populations dictionary
Check if parameters sets an initial chromosome
Check for source folder in parameters
[false]
[true]
Check for database source in parameters
Initialize index, i = 0
Assimilate old parameters to new parameters
Reconstruct old parameters
Establish connection to database
Construct database directory name
[true]
[false]
Add old key to new parameters
Check if old key is not present in new parameters
Check if i < number of keys in old parameters
Reconstruct world object
Reconstruct population objects
Initialize index, i = 0
Add population object to population dictionary
Update, i++
Reconstruct population object
Check if i < number of populations
Terminate database connection
Excavate world object
Create chromosome with 0 bases according to chromosome size
Update parameters according to world dimensions
Revive populations from gap files
Add revival generation start to parameters
Add max generation to parameters
Revive population
Initialize index, i = 0
Check if i < number of populations
Update, i++
Add deployment scheme to parameters
Add created chromosome to parameters as initial chromosome
Construct world object according to parameters
Spawn populations according to parameters
Call simulation core
Add population size to parameters

Figure 2. Activity Diagram for Simulation Preparation. Simulation preparation procedure is dependent on whether the simulation is a new simulation (regular simulation) or revival and continuation of an existing simulation (simulation revival) from database or files.

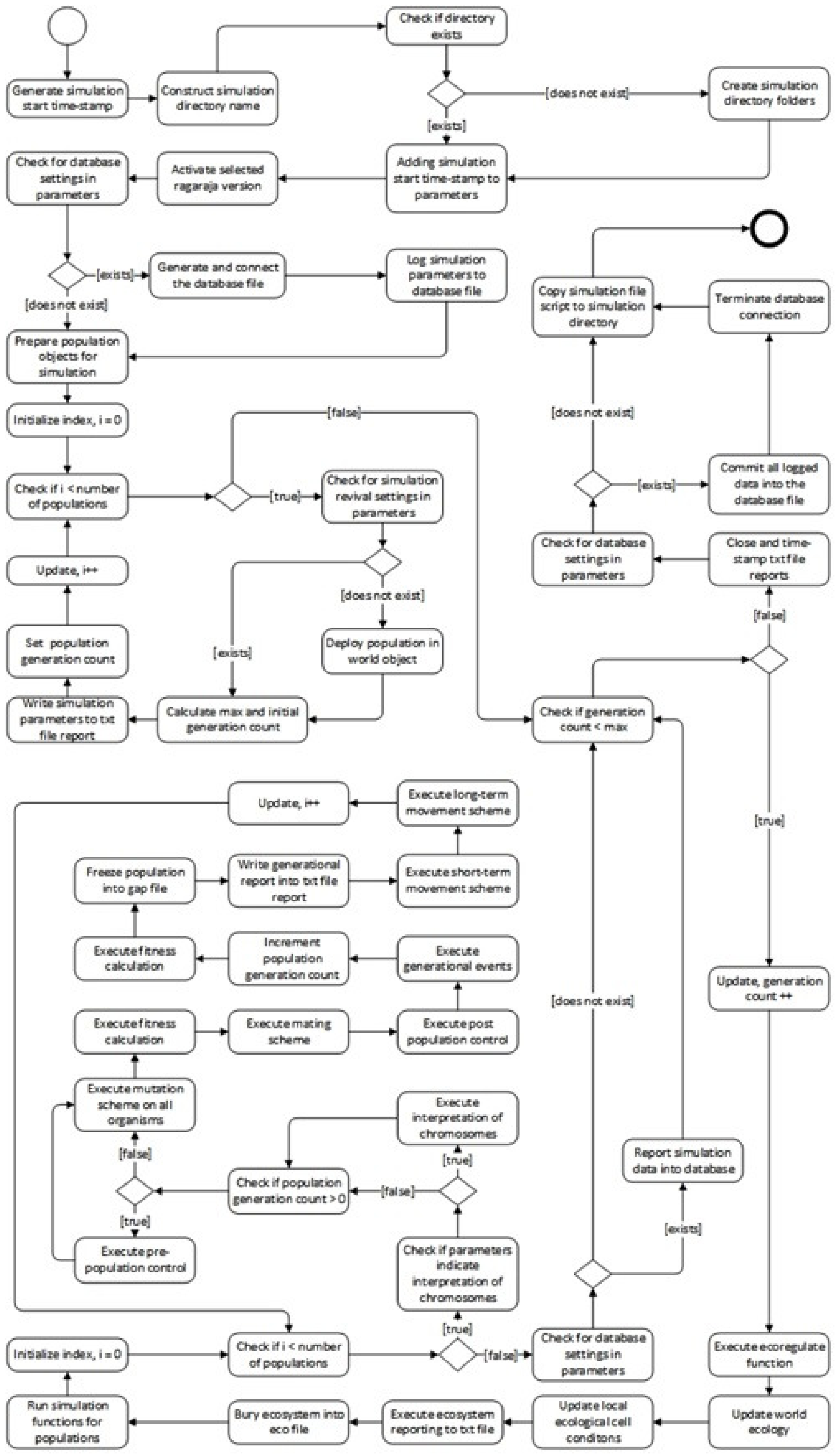
Generate simulation start time-stamp
Construct simulation directory name
Check if directory exists
[does not exist]
Create simulation directory folders
[exists]
Adding simulation start time-stamp to parameters
Activate selected ragaraja version
Check for database settings in parameters
[exists]
Generate and connect the database file
Log simulation parameters to database file
[does not exist]
Prepare population objects for simulation
Initialize index, i = 0
Check if i < number of populations
[false]
[true]
Check for simulation revival settings in parameters
[does not exist]
Deploy population in world object
[exists]
Calculate max and initial generation count
Write simulation parameters to txt file report
Set population generation count
Update, i++
Copy simulation file script to simulation directory
Terminate database connection
[does not exist]
[exists]
Commit all logged data into the database file
Check for database settings in parameters
Close and time-stamp txt file reports
[false]
Check if generation count < max
[true]
Update, generation count ++
Update, i++
Execute long-term movement scheme
Freeze population into gap file
Write generational report into txt file report
Execute short-term movement scheme
Execute fitness calculation
Increment population generation count
Execute generational events
Execute fitness calculation
Execute mating scheme
Execute post population control
Execute mutation scheme on all organisms
[false]
Execute interpretation of chromosomes
[true]
Check if population generation count > 0
[false]
[true]
Execute pre-population control
Check if parameters indicate interpretation of chromosomes
[true]
[does not exist]
Report simulation data into database
[exists]
Initialize index, i = 0
Check if i < number of populations
[false]
Check for database settings in parameters
Execute ecoregulate function
Run simulation functions for populations
Bury ecosystem into eco file
Execute ecosystem reporting to txt file
Update local ecological cell conditons
Update world ecology

Figure 3. Activity Diagram for Simulation Execution.

After simulation preparation is completed, a unique folder for storing text-based simulation results (results folder) is created using the datetime stamp of the simulation execution. Logging database will be created if database logging is requested and the database file is not present. The simulation is then executed from first generation to the maximum generation as defined in the parameter, and the events are reported into a text file or database as required (Figure 3; Castillo and Ling, 2014a). After DO initialization, the current simulation driver simulates each organism and ecological cell sequentially (Ling, 2012a). Hence, *simulation_calls.py* performs the task of the original command line simulation runner, *run_dose.py* (Ling, 2012a).

DOSE Code Files

File name: metadata.py

```
'''
Digital Organism Simulation Environment (DOSE) Metadata

Date created: 14th February 2014
'''

pypi_name = 'digital organism simulation environment'
description = 'Digital Organism Simulation Environment (DOSE)'
version = '1.0.4'
maintainer = 'Maurice HT Ling'
email = 'mauriceling@acm.org'
download_url = 'https://github.com/mauriceling/dose/releases'
project_start_year = '2010'
project_website = 'https://github.com/mauriceling/dose'
documentation_website = 'http://maurice.vodien.com/project-dose'
short_license = 'GNU General Public License version 3'
long_license = '''
Unless otherwise specified, dose.copads package will be licensed
under Python Software Foundation License version 2; all other files will
be licensed GNU General Public License version 3.'''
platforms = ['Operating System :: OS Independent']
trove_classifiers = \
    ['Development Status :: 3 - Alpha',
    'Intended Audience :: Developers',
    'Intended Audience :: Education',
    'Intended Audience :: Science/Research',
    'License :: OSI Approved :: GNU General Public License v3 (GPLv3)',
    'Natural Language :: English',
    'Operating System :: OS Independent',
    'Programming Language :: Python :: 2 :: Only',
    'Topic :: Education',
    'Topic :: Scientific/Engineering :: Artificial Life',
    'Topic :: Scientific/Engineering :: Bio-Informatics',
    'Topic :: Software Development :: Libraries :: Python Modules'
    ]

from datetime import datetime
copyright_year = '-'.join([project_start_year,
                           str(datetime.now().year)])

long_description = '''%s

Life is fascinating and deeply intriguing. Despite so, life forms on Earth
or carbon-based life forms as a group is just one form, one possible sample
of possibly a whole magnitude of life. Even then, there are many aspects of
life that cannot be deciphered even by examining current life forms; for
example, how did chemical reactions organize themselves into biochemical
```

pathways? How did life start? How is intelligence formed?

To answer such questions, we will have to restart our evolutionary time to the very beginning - clearly an impossibly gargantuan task. At the same time, studying biological/carbon-based life forms is expensive, time consuming and destructive. As a molecular biologist, there is no way I can examine the entire genome of even a bacteria in an inanimate state, then somehow allow it to continue living as if time had just stopped while I am examining it.

However, if I can simulate a bacteria or any life form in a computer, then I can make a digital copy of the bacterium, pull it apart to study it while the original bacterium continues "living" in my virtual world without even knowing that it had been duplicated. Many biologists thought of virtual life forms as a new way to learn about life itself. Studying of virtual life forms is known as Artificial Life and I term "virtual life forms" as "digital organisms". There are several advantages in experimenting using digital organisms. Firstly, generation time can be much faster compared to most biological life. Secondly, it is usually cheaper to examine computer simulations than working on actual biological life. Perhaps the most important advantage of looking at life from this perspective is that by recreating life in a different medium, we are not limited to our own system of carbon-based life; hence, studying life as what-it-could-be.

Digital Organisms Simulation Environment (DOSE) is essentially a virtual world simulator for studying digital organisms. I will argue that digital organisms are considered living organisms (Koh and Ling, 2013). Despite so, being a molecular biologist by training, I have a hard time mapping components of digital organisms into biological life whenever such components are too abstract.

Hence, I decided to design an artificial life / digital organism simulator that bears resemblance to biological life and ecology. These are the foundation papers:

1. Lim, JZR, Aw, ZQ, Goh, DJW, How, JA, Low, SXZ, Loo, BZL, Ling, MHT. 2010. A genetic algorithm framework grounded in biology. The Python Papers Source Codes 2: 6.

This manuscript describes the implementation of a GA framework that uses biological hierarchy - from chromosomes to organisms to population.

2. Ling, MHT. 2012. An Artificial Life Simulation Library Based on Genetic Algorithm, 3-Character Genetic Code and Biological Hierarchy. The Python Papers 7: 5.

Genetic algorithm (GA) is inspired by biological evolution of genetic organisms by optimizing the genotypic combinations encoded within each individual with the help of evolutionary operators, suggesting that GA may be a suitable model for studying real-life evolutionary processes. This paper describes the design of a Python library for artificial life simulation, Digital Organism Simulation Environment (DOSE), based on GA and biological hierarchy starting from genetic sequence to population. A 3-character instruction set that does not take any operand is introduced as genetic code for digital organism. This mimics the 3-nucleotide codon structure in naturally occurring DNA. In addition, the context of a 3-dimensional world composing of ecological cells is introduced to simulate a physical ecosystem.

- Ling, MHT. 2012. Ragaraja 1.0: The Genome Interpreter of Digital Organism Simulation Environment (DOSE). The Python Papers Source Codes 4: 2.

This manuscript describes the implementation and test of Ragaraja instruction set version 1.0, which is the core genomic interpreter of DOSE.

```
From this foundation, the complete suite of Digital Organisms Simulation
Environment (DOSE) can be build.

Project website: U{%s}

Documentation can be found at %s

License: %s

Copyright %s, Maurice HT Ling (on behalf of all authors).
''' % (description,
       project_website,
       documentation_website,
       long_license,
       copyright_year)
```

File name: __init__.py

```
'''
Digital Organism Simulation Environment (DOSE)

Date created: 27th September 2013
'''
import metadata

__version__ = metadata.version
__maintainer__ = metadata.maintainer
__email__ = metadata.email
__description__ = metadata.long_description

# Package imports (in ascending order of package names)
import copads

# Module imports (in ascending order of module names)
import database_calls
import dose
import genetic
import register_machine
import ragaraja

# COPADS Class imports (in ascending order of module names, then class names)
from copads.lindenmayer import lindenmayer

# DOSE Class imports (in ascending order of module names, then class names)
from dose import dose_functions
from dose_world import World
from genetic import Chromosome
from genetic import Organism
from genetic import Population

# Function imports (in ascending order of module names, then function names)
from database_calls import connect_database
from database_calls import db_list_datafields
from database_calls import db_list_generations
from database_calls import db_list_simulations
from database_calls import db_list_population_name
from database_calls import db_get_ecosystem
from database_calls import db_get_organisms_chromosome_sequences
from database_calls import db_get_organisms_genome
from database_calls import db_get_organisms_status
from dose import database_report_populations
from dose import database_report_world
from dose import filter_age
from dose import filter_deme
from dose import filter_gender
from dose import filter_location
from dose import filter_status
from dose import filter_vitality
from dose import revive_simulation
from dose import simulate
from genetic import crossover
from genetic import population_constructor
from genetic import population_simulate
```

File name: dose.py

```
'''
Application Programming Interface (API) for DOSE (digital organism
simulation environment). This contains the main functions and operations
needed to write a DOSE simulation. This file will be imported as top
level (from dose import *) when DOSE is imported; hence, all functions in
this file can be assessed at top level.

Date created: 27th September 2013
'''
import sys, os, random, inspect

import ragaraja, register_machine
import dose_world
import genetic, simulation_calls

from simulation_calls import spawn_populations, simulation_core
from simulation_calls import excavate_world, revive_population

from database_calls import connect_database, db_reconstruct_simulation_parameters
from database_calls import db_reconstruct_population, db_reconstruct_world,
db_list_simulations

class dose_functions():
    '''
    Abstract class to contain all of the simulation-specific functions
    (functions that vary with each simulation) that are to be defined /
    implemented by the user to be used in a simulation. This class should
    be inherited by every simulation to over-ride each function / method.

    This set of functions / methods is consolidated functions / methods to
    be over-ridden from genetic.Organism, genetic.Population, and
    dose_world.World classes. As a result, the functions / methods can be
    working at different levels - at the level of individual organisms,
    at the level of entire population(s), or at the level of the world.

    Please see the examples in examples directory on its use.
    '''
    def mutation_scheme(self, organism):
        '''
        Method / function to trigger mutational events in each chromosome
        of the genome within an organism. This function works at the
        level of individual organisms.

        @param organism: genetic.Organism object
        @return: None
        '''
        raise NotImplementedError
    def prepopulation_control(self, Populations, pop_name):
        '''
        Method / function to trigger population control events before
        mating event in each generation. For example, it can be used to
        simulate pre-puberty (childhood) death. This function works at
        the level of entire population(s).

        @param Populations: A dictionary containing one or more populations
        where the value is a genetic.Population object.
        @param pop_name: Name of the population which is used as key in
        the the dictionary (Populations parameter).
        @return: None
        '''
        raise NotImplementedError
    def fitness(self, Populations, pop_name):
        '''
        Method / function to calculate the fitness score of each organism
        within the population(s). This function works at the level of
        entire population(s) even though fitness calculation occurs at the
        organism level. The fitness of each organism may be stored in
        Organism.status['fitness'] and may be used by mating scheme.

        @param Populations: A dictionary containing one or more populations
        where the value is a genetic.Population object.
        @param pop_name: Name of the population which is used as key in
        the the dictionary (Populations parameter).
        @return: None
        '''
        raise NotImplementedError
    def mating(self, Populations, pop_name):
        '''
        Method / function to trigger mating events in each generation. For
        example, it can be used to simulates mate choices and progeny size.
        A support function provided is genetic.crossover() function which
        generates one random crossover operation between 2 chromosomes, to
        simulate meiosis crossover. This function may also use one or more
        of the dose.filter_XXX() functions to select or choose suitable
        mates. This function works at the level of entire population(s),
        which means that this function will have
            - to manage mating scheme and progeny (offspring) generation
            for the entire population
            - add or replace offsprings into the respective population(s)
            - (optional) store identity of parent(s) as list; for example,
            [parentA identity, parentB identity], in offspring's
            status['parents'] for ancestral tracing.

        @param Populations: A dictionary containing one or more populations
        where the value is a genetic.Population object.
        @param pop_name: Name of the population which is used as key in
        the the dictionary (Populations parameter).
        @return: None
        '''
        raise NotImplementedError
    def postpopulation_control(self, Populations, pop_name):
        '''
        Method / function to trigger population control events after
        mating event in each generation. For example, it can be used to
        simulate old-age death. This function works at the level of entire
        population(s).

        @param Populations: A dictionary containing one or more populations
        where the value is a genetic.Population object.
        @param pop_name: Name of the population which is used as key in
        the the dictionary (Populations parameter).
        @return: None
        '''
        raise NotImplementedError
    def generation_events(self, Populations, pop_name):
        '''
        Method / function to trigger other defined events in each
        generation. For example, it can be used to simulate catastrophe
        or epidemic that does not occur regularly, or simulates unusual
        occurrences of multiple mutation events. This function works at
        the level of entire population(s).
        @param Populations: A dictionary containing one or more populations
        where the value is a genetic.Population object.
        @param pop_name: Name of the population which is used as key in
        the the dictionary (Populations parameter).
        @return: None
        '''
        raise NotImplementedError
    def population_report(self, Populations, pop_name):
        '''
        Method / function to generate a text report of the population(s)
        and/or each organisms within the population at regular intervals,
        within the simulation, as determined by "print_frequency" in the
        simulation parameters. This function works at the level of entire
        population(s).

        @param Populations: A dictionary containing one or more populations
        where the value is a genetic.Population object.
        @param pop_name: Name of the population which is used as key in
        the the dictionary (Populations parameter).
        @return: Entire report of a population at a generation count in a
        string. To make it human-readable, usually the report string is
        both tab-delimited (for one organism) and newline-delimited (for
        entire population).
        '''
        raise NotImplementedError
    def organism_movement(self, Populations, pop_name, World):
        '''
        organism_movement and organism_location are both methods /
        functions to execute movement of organisms within the world. The
        semantic difference between organism_movement and organism_location
        is that organism_movement is generally used for short travels
        while organism_location is used for long travel. For example,
        organism_movement can be used to simulate foraging or nomadic
        behaviour. This function works at both the level of entire
        population(s) and world.

        For each organism to move, this function will have to
            - update the number of organisms in each
            World.ecosystem[x-axis][y-axis][z-axis]['organisms']
            - update the respective Organism's location in the status
            dictionary (Population[pop_name].agents[<index>].status['location'])

        @param Populations: A dictionary containing one or more populations
        where the value is a genetic.Population object.
        @param pop_name: Name of the population which is used as key in
        the the dictionary (Populations parameter).
        @param World: dose_world.World object.
        @return: None
        '''
        raise NotImplementedError
    def organism_location(self, Populations, pop_name, World):
        '''
        organism_movement and organism_location are both methods /
        functions to execute movement of organisms within the world. The
        semantic difference between organism_movement and organism_location
        is that organism_movement is generally used for short travels
        while organism_location is used for long travel. For example,
        organism_movement can be used to simulate long distance migration,
        such as air travel. This function works at both the level of entire
        population(s) and world.

        For each organism to move, this function will have to
            - update the number of organisms in each
            World.ecosystem[x-axis][y-axis][z-axis]['organisms']
            - update the respective Organism's location in the status
            dictionary (Population[pop_name].agents[<index>].status['location'])

        @param Populations: A dictionary containing one or more populations
        where the value is a genetic.Population object.
        @param pop_name: Name of the population which is used as key in
        the the dictionary (Populations parameter).
        @param World: dose_world.World object.
        @return: None
        '''
        raise NotImplementedError
    def ecoregulate(self, World):
        '''
        Method / function for broad spectrum management of the entire
        ecosystem defined as World.ecosystem[x-axis][y-axis][z-axis]
        ['local_input'] and World.ecosystem[x-axis][y-axis][z-axis]
        ['local_output']). For example, it can be used to simulate
        temperature, solar radiation, or resource gradients. This function
        works at the level of the world.

        @param World: dose_world.World object.
        @return: None
        '''
        raise NotImplementedError
    def update_ecology(self, World, x, y, z):
        '''
        Method / function to process the input and output from the
        activities of the organisms in the current ecological cell (defined
        as World.ecosystem[x-axis][y-axis][z-axis]['temporary_input'] and
        World.ecosystem[x-axis][y-axis][z-axis]['temporary_output']) into
        a local ecological cell condition (defined as World.ecosystem
        [x-axis][y-axis][z-axis]['local_input'] and World.ecosystem[x-axis]
        [y-axis][z-axis]['local_output']), and update the
        ecosystem (which is essentially the ecological cell adjacent to
        World.ecosystem[x-axis][y-axis][z-axis]). For example, it can be
        used to simulate secretion of chemicals or use of resources (such
        as food) by organisms, and diffusion of secretions to the
        neighbouring ecological cells.

        Essentially, this function simulates the "diffusion" of local
        situation outwards. This function works at the level of the world.

        @param World: dose_world.World object.
        @param x: x-axis of the World.ecosystem to identify the cell.
        @param y: y-axis of the World.ecosystem to identify the cell.
        @param z: z-axis of the World.ecosystem to identify the cell.
        @return: None
        '''
        raise NotImplementedError
    def update_local(self, World, x, y, z):
        '''
        Method / function to update local ecological cell condition (defined
        as World.ecosystem[x-axis][y-axis][z-axis]['local_input'] and
        World.ecosystem[x-axis][y-axis][z-axis]['local_output']) from the
        ecosystem (World.ecosystem [x-axis][y-axis][z-axis]['local_input']
        and World.ecosystem[x-axis][y-axis][z-axis]['local_output'] of
        adjacent ecological cells). For example, it can be used to
        simulates movement or diffusion of resources from the ecosystem to
        local.

        Essentially, this function is the reverse of update_ecology()
        function. In this case, the local ecological cell is affected by
        adjacent conditions. This function works at the level of the world.

        @param World: dose_world.World object.
        @param x: x-axis of the World.ecosystem to identify the cell.
        @param y: y-axis of the World.ecosystem to identify the cell.
        @param z: z-axis of the World.ecosystem to identify the cell.
        @return: None
        '''
        raise NotImplementedError
    def report(self, World):
        '''
        Method / function to generate a text report of the ecosystem status
        (World.ecosystem) at regular intervals, within the simulation, as
        determined by "print_frequency" in the simulation parameters. This
        function works at the level of world.

        @param World: dose_world.World object.
        @return: Entire report of a population at a generation count in a
        string. To make it human-readable, usually the report string is
        both tab-delimited (for one organism) and newline-delimited (for
        entire population).
        '''
        raise NotImplementedError
    def deployment_scheme(self, Populations, pop_name, World):
        '''
        Method / function to implement a user-specific / simulation-
        specific deployment scheme used to deploy organisms into the
        World. This function will only be used when "deployment_code" in
        simulation parameters dictionary equals to 0. This function works
        at all three levels - organism(s), population(s), and world.

        @param Populations: A dictionary containing one or more populations
        where the value is a genetic.Population object.
        @param pop_name: Name of the population which is used as key in
        the the dictionary (Populations parameter).
        @param World: dose_world.World object.
        @return: None
        '''
        raise NotImplementedError
    def database_report(self, con, cur, start_time,
                        Population, World, generation_count):
        '''
        Method / function to implement database logging of each organism
        in each population, and the ecosystem status. The frequency of
        logging is determined by "database_logging_frequency" in the
        simulation_parameters.

        Three database tables had been defined for use in this function:
            - organisms where the structure is (start_time text, pop_name
            text, org_name text, generation text, key text, value text)
            - world where the structure is (start_time text, x text,
            y text, z text, generation text, key text, value text)
            - miscellaneous where the structure is (start_time text,
            generation text, key text, value text)

        The logical purpose of these tables are:
            - organisms, to log status of each organism. Starting time of
            current simulation (start_time), population name (pop_name),
            organism name (org_name), and generation count (generation)
            are used as complex primary key to identify the organism
            within a specific simulation at a specific generation. Key and
            value pair makes up the actual data to be logged where key is
            the field and value is the datum or attribute.
            - world, to log the status of the ecosystem. Starting time of
            current simulation (start_time), location of ecological cell
            (x, y, z as coordinates), and generation count (generation)
            are used as complex primary key to identify the ecological cell
            within a specific simulation at a specific generation. Key and
            value pair makes up the actual data to be logged where key is
            the field and value is the datum or attribute.
            - miscellaneous, is used to log other undefined data. Starting
            time of current simulation (start_time), and generation count
            (generation) are used as complex primary key to identify a
            specific generation within a specific simulation. Key and
            value pair makes up the actual data to be logged where key is
            the field and value is the datum or attribute.

        @param con: Database connector. See Python DB-API for details.
        @param cur: Database cursor. See Python DB-API for details.
        @param start_time: Starting time of current simulation in the
        format of <date>-<seconds since epoch>; for example,
        2013-10-11-1381480985.77.
        @param Population: A dictionary containing one or more populations
        where the value is a genetic.Population object.
        @param World: dose_world.World object.
        @param generation_count: Current number of generations simulated.
        @return: None
        '''
        raise NotImplementedError

def database_report_populations(con, cur, start_time,
                                Populations, generation_count):
    '''
    Function to log organisms' status and genome into database. Organisms'
    status is implemented as a dictionary and each key-value pair in the
    status is logged as a separate record. Similarly, each chromosome is
    logged as a separately record. A combination of starting time of the
    simulation, population name, organism's name, and the generation can
    identify all the data specific to an organism within a population, at
    a specific generation within a simulation. This function is a complete
    logger - it logs everything there is about an organism. There is
    nothing to log for populations.

    The following transformations of data are made:
        - Organism.status['blood'] is a list of numbers resulting from
        interpreting the genome by Ragaraja interpreter. The numbers are
        concatenated and delimited by '|'. For example, [1, 2, 3] ==> 1|2|3
        - Organism.status['location'] is a tuple of 3 integers (x, y, z) for
        location of ecological cell. The numbers are concatenated and
        delimited by '|'. For example, (2, 3, 4) ==> 2|3|4
        - Each chromosome in Organism.genome is a list of bases. These bases
        are concatenated with no delimiter. For example, [1, 2, 3] ==> 123

    @param con: Database connector. See Python DB-API for details.
    @param cur: Database cursor. See Python DB-API for details.
    @param start_time: Starting time of current simulation in the
    format of <date>-<seconds since epoch>; for example,
    2013-10-11-1381480985.77.
    @param Populations: A dictionary containing one or more populations
    where the value is a genetic.Population object.
    @param generation_count: Current number of generations simulated.
    @return: None
    '''
    generation = str(generation_count)
    for pop_name in list(Populations.keys()):
        for org in Populations[pop_name].agents:
            org_name = str(org.status['identity'])
            # log each item in Organism.status dictionary
            for key in [key for key in list(org.status.keys())
                                   if key != 'identity']:
                if key in ('blood', 'location', 'parents'):
                    try:
                        # TypeError will occur when genome/chromosomes are
                        # not interpreted
                        value = '|'.join([str(x) for x in org.status[key]])
                    except TypeError: value = ''
                else:
                    value = str(org.status[key])
                cur.execute('insert into organisms values (?,?,?,?,?,?)',
                    (str(start_time), str(pop_name), org_name,
                     generation, key, value))
            # log each chromosome sequence
            for chromosome_count in range(len(org.genome)):
                key = 'chromosome_' + str(chromosome_count)
                sequence = ''.join(org.genome[chromosome_count].sequence)
                cur.execute('insert into organisms values (?,?,?,?,?,?)',
                    (str(start_time), str(pop_name), org_name,
                     generation, key, sequence))
    con.commit()

def database_report_world(con, cur, start_time, World, generation_count):
    '''
    Function to log World.ecosystem into database. The ecosystem is made
    up of a collection of ecological cells, identified by the (x, y, z)
    coordinates within the ecosystem. Each ecological cell is implemented
    as a dictionary of ecological status. This function logs the entire
    set of ecological cells; thus, a complete logger. Each ecosystem within
    a specific simulation, at a specific time/generation, can be identified
    by a combination of starting time of simulation and generation. Each
    ecological cell within the ecosystem can be further identified by the
    (x, y, z) coordinates.

    @param con: Database connector. See Python DB-API for details.
    @param cur: Database cursor. See Python DB-API for details.
    @param start_time: Starting time of current simulation in the
    format of <date>-<seconds since epoch>; for example,
    2013-10-11-1381480985.77.
    @param World: dose_world.World object.
    @param generation_count: Current number of generations simulated.
    @return: None
    '''
    generation = str(generation_count)
    ecosystem = World.ecosystem
    location = [(x, y, z)
                for x in range(len(ecosystem))
                    for y in range(len(ecosystem[x]))
                        for z in range(len(ecosystem[x][y]))]
    for cell in location:
        eco_cell = ecosystem[cell[0]][cell[1]][cell[2]]
        for key in list(eco_cell.keys()):
            value = str(eco_cell[key])
            cur.execute('insert into world values (?,?,?,?,?,?,?)',
                        (str(start_time),
                         str(cell[0]), str(cell[1]), str(cell[2]),
                         generation, key, value))
    con.commit()

def filter_deme(deme_name, agents):
    '''
    Function to identify organisms (agents) with a specific sub-population
    name (also known as deme) within a population. Demes can be considered
    as strains in bacteria, breed or sub-species in animals, races in
    humans, and cultivars in plants. This function is can be used to
    support the identification of suitable mates for mating schemes.

    @param deme_name: Name of deme (sub-population name)
    @type deme_name: string
    @param agents: A list of organisms, such as Population.agents.
    @return: List of Organism objects
    '''
    extract = [individual for individual in agents
               if individual.status['deme'].upper() == deme_name.upper()]
    return extract
```

```python
def filter_gender(gender, agents):
    '''
    Function to identify organisms (agents) with a specific gender within
    a population. This function is can be used to support the identification
    of suitable mates for mating schemes.

    @param gender: Gender
    @type gender: string
    @param agents: A list of organisms, such as Population.agents.
    @return: List of Organism objects
    '''
    extract = [individual for individual in agents
               if individual.status['gender'].upper() == gender.upper()]
    return extract

def filter_age(minimum, maximum, agents):
    '''
    Function to identify organisms (agents) within a certain age range in
    a population. This function is can be used to support the identification
    of suitable mates for mating schemes.

    @param minimum: Minimum age
    @type minimum: float
    @param maximum: Maximum age
    @type maximum: float
    @param agents: A list of organisms, such as Population.agents.
    @return: List of Organism objects
    '''
    extract = [individual for individual in agents
               if float(individual.status['age']) > (float(minimum) - 0.01) \
               and float(individual.status['age']) < float(maximum) + 0.01]
    return extract

def filter_location(location, agents):
    '''
    Function to identify organisms (agents) of a population within a
    specific ecological cell. This function is can be used to support the
    identification of suitable mates for mating schemes.

    @param location: (x, y, z) coordinates within the World.ecosystem
    @param location: tuple
    @param agents: A list of organisms, such as Population.agents.
    @return: List of Organism objects
    '''
    extract = [individual for individual in agents
               if individual.status['location'] == location]
    return extract

def filter_vitality(minimum, maximum, agents):
    '''
    Function to identify organisms (agents) within a certain vitality score
    in a population. This function is can be used to support the identification
    of suitable mates for mating schemes.

    @param minimum: Minimum vitality score
    @type minimum: float
    @param maximum: Maximum vitality score
    @type maximum: float
    @param agents: A list of organisms, such as Population.agents.
    @return: List of Organism objects
    '''
    extract = [individual for individual in agents
            if float(individual.status['vitality']) > (float(minimum) - 0.01) \
            and float(individual.status['vitality']) < float(maximum) + 0.01]
    return extract
```

```
def filter_status(status_key, condition, agents):
    '''
    Generic function to identity organisms (agents) within a population
    via the status of organisms (Organism.status dictionary). This function
    is can be used to support the identification of suitable mates for
    mating schemes.

    @param status_key: Status (key in Organism.status dictionary) to filter
    @type status_key: string
    @param condition: Condition of the status (value in Organism.status
    dictionary) to filter. This can be a unique condition, such as "True",
    or a range, such as (minimum, maximum) to define the minimum and
    maximum value of the condition.
    @param agents: A list of organisms, such as Population.agents.
    @return: List of Organism objects
    '''
    if type(condition) in (str, int, float, bool):
        extract = [individual for individual in agents
                   if individual.status[status_key] == condition]
    else:
        extract = [individual for individual in agents
        if float(individual.status[status_key]) > float(condition[0]) - 0.01 \
        and float(individual.status[status_key]) < float(condition[1]) + 0.01]
    return extract

def revive_simulation(rev_parameters, sim_functions):
    print('\n[' + rev_parameters["simulation_name"].upper() + ' REVIVAL SIMULATION]')
    Populations = {}
    if "sim_folder" in rev_parameters:
        print('Accessing simulation files directory...')
        print('Excavating World entity: ' + rev_parameters['eco_file'] + '...')
        World = excavate_world(rev_parameters['sim_folder'] + \
                               rev_parameters['eco_file'])
        print('Updating parameters with World dimensions...')
        rev_parameters["world_z"] = len(World.ecosystem[0][0][0])
        rev_parameters["world_y"] = len(World.ecosystem[0][0])
        rev_parameters["world_x"] = len(World.ecosystem[0])
        for i in range(len(rev_parameters["pop_files"])):
            print('\nReviving population file: ' + \
                rev_parameters["pop_files"][i] + '...')
            pop_file = rev_parameters["sim_folder"] + \
                rev_parameters["pop_files"][i]
            Populations[rev_parameters["population_names"][i]] = \
                revive_population(pop_file)
        print('\nUpdating revival generation start in simulation parameters...')
        rev_parameters["rev_start"] = [Populations[pop_name].generation
                                       for pop_name in Populations]
    elif "database_source" in rev_parameters:
        print('Constructing database directory...')
        dbpath = os.sep.join([os.getcwd(),
                              'Simulations',
                              rev_parameters["database_source"]])
        print('Connecting to database file: ' + \
            rev_parameters["database_source"] + '...')
        (con, cur) = connect_database(dbpath, None)
        if rev_parameters["simulation_time"] == 'default':
            print('Acquiring simulation starting time...')
            rev_parameters["simulation_time"] = db_list_simulations(cur)[0][0]
        print('Reconstructing old simulation parameters...')
        temp_parameters = db_reconstruct_simulation_parameters(cur,
                                   rev_parameters["simulation_time"])
        print('Assimilating old simulation parameters with new simulation
parameters...')
        for key in temp_parameters:
            if key not in rev_parameters:
                rev_parameters[key] = temp_parameters[key]
        print('Reconstructing World entity...')
        World = db_reconstruct_world(cur, rev_parameters["simulation_time"],
                                     rev_parameters["rev_start"][0])
        print('\nUpdating population names parameter...')
        for pop_name in rev_parameters["population_names"]:
            print('Reconstructing population: ' + pop_name + '...')
            Populations[pop_name] = db_reconstruct_population(cur,
                              rev_parameters["simulation_time"], pop_name,

rev_parameters["rev_start"][rev_parameters["population_names"].index(pop_name)])
        print('Terminating database connection...')
        con.close()
    print('Updating last generation revival and population size simulation
parameters...')
    rev_parameters["rev_finish"] = [(Populations[pop_name].generation + \
                                     rev_parameters["extend_gen"])
                                    for pop_name in Populations]
    rev_parameters["rev_pop_size"] = [len(Populations[pop_name].agents)
                                      for pop_name in Populations]
    print('\nStarting simulation core...')
    simulation_core(sim_functions, rev_parameters, Populations, World)

def simulate(sim_parameters, sim_functions):
    '''
    Function called by simulation to run the actual simulation based on a
    set of parameters and functions.

    Simulation parameters dictionary contains the following keys:
        - simulation_name: Short name of the simulation.
        - population_names: Population name(s) in a list.
        - population_locations: A list of lists containing (x, y, z)
        coordinates in the world to deploy the populations. There must be
        equal numbers of items in this list as population_names. However,
        one population can be deployed into one or more ecological cell(s).
        - deployment_code: Defines the type of deployment. Allowable values
        are 0 (custom deployment scheme which users can implement by over-
        riding dose.dose_functions.deployment_scheme() function), 1 (all
        organisms are in one location), 2 (organisms are randomly deployed
        across a list of population locations given for the population), 3
        (oganisms are randomly deployed across a list of population locations
        given for the population), 4 (organisms are dispersed from a specific
        location where the defined location will have the maximum allocation
        before dispersal happens). (see simulation_calls.deployment for more
        details)
        - chromosome_bases: List containing allowable bases in the
        chromosome.
        - background_mutation: Defines background mutation rate where 0.01
        represents 1% mutation and 0.5 represents 50% mutation.
        - additional_mutation: Defines mutation rate on top of background
        mutation rate where 0.01 represents 1% mutation and 0.5 represents
        50% mutation.
        - mutation_type: Type of mutation for background mutation rate.
        Allowable values are 'point' (point mutation), 'insert' (insert a
        base), 'delete' (delete a base), 'invert' (invert a stretch of the
        chromosome), 'duplicate' (duplicate a stretch of the chromosome),
        'translocate' (translocate  a stretch of chromosome to another
        random position).
        - chromosome_size: Number of bases for a chromosome.
        - genome_size: Number of chromosome(s) in a genome.
        - max_tape_length: Maximum number of cells for array used in
        Ragaraja interpreter, which will be stored as
        Organism.status['blood'].
        - clean_cell: Toggle to define if a new tape will be used for
        Ragaraja interpreter. If True, at every Ragaraja interpretation of
        the source (genome), a new tape ([0] * max_tape_length) will be
        provided. If False, the tape stored as Organism.status['blood']
        (results from previous interpretation of genome) will be used.
        - interpret_chromosome: Toggle to define whether genomes will be
        interpreted by Ragaraja. If True, chromosomes of each organism will
        be interpreted by Ragaraja.
        - max_codon: Maximum number of Ragaraja instructions to execute per
        organism, which can be used as force-break from endless loop.
        - population_size: Number of organisms per population for initial
        deployment.
        - eco_cell_capacity: Maximum number of organisms per ecological
        cell at the time of deployment prior to the start of simulation.
        - world_x: Number of ecological cells in the x-axis of the
        World.ecosystem.
        - world_y: Number of ecological cells in the y-axis of the
        World.ecosystem.
        - world_z: Number of ecological cells in the z-axis of the
        World.ecosystem.
        - goal: Goal for population to reach. This provides a goal for use
        in fitness functions.
        - maximum_generations: Number of generations to simulate.
        - fossilized_ratio: Proportion (less than 1) of the population to
        fossilize or freeze.
        - fossilized_frequency: Number of generations (intervals) for each
        population fossilization or freezing event.
        - print_frequency: Number of generations (intervals) for each
        reporting event into files.
        - ragaraja_version: Ragaraga instruction version to activate. (see
        ragaraja.activate_version() for more details)
        - ragaraja_instructions: A list defining a set of Ragaraga
        instructions to activate. This is only useful when ragaraja_version
        is 0.
        - eco_buried_frequency: Number of generations (intervals) for each
        ecological freezing or burial event.
        - database_file: Logging database file name. This file will be
        located in <current working directory>/Simulations folder
        - database_logging_frequency: Number of generations (intervals) for
        each database logging event.

    Methods / Functions from dose.dose_functions class to be over-ridden
    as simulation_functions (for more details, please look at
    dose.dose_functions class):
        - organism_movement: organism_movement and organism_location are
        both methods / functions to execute movement of organisms within
        the world. The semantic difference between organism_movement and
        organism_location is that organism_movement is generally used for
        short travels while organism_location is used for long travel.
        - organism_location: organism_movement and organism_location are
        both methods / functions to execute movement of organisms within
        the world. The semantic difference between organism_movement and
        organism_location is that organism_movement is generally used for
        short travels while organism_location is used for long travel.
        - ecoregulate: Broad spectrum management of the entire
        ecosystem.
        - update_ecology: Process the input and output from the activities
        of the organisms in the current ecological cell into a local
        ecological cell condition, and update the ecosystem.
        - update_local: Update local ecological cell condition from the
        ecosystem.
        - report: Generate a text report of ecosystem (World.ecosystem) at
        regular intervals, within the simulation, as determined by
        "print_frequency" in the simulation parameters.
        - fitness: Calculate the fitness score of each organism within the
        population(s).
        - mutation_scheme: Trigger mutational events in each chromosome
        of the genome within an organism.
        - prepopulation_control: Trigger population control events before
        mating event in each generation.
        - mating: Trigger mating events in each generation.
        - postpopulation_control: Trigger population control events after
        mating event in each generation.
        - generation_events: Trigger other defined events in each
        generation.
        - population_report: Generate a text report of the population(s)
        and/or each organisms within the population at regular intervals,
        within the simulation, as determined by "print_frequency" in the
        simulation parameters
        - database_report: Implement database logging of each organism
        in each population, and the ecosystem status. The frequency of
        logging is determined by "database_logging_frequency" in the
        simulation_parameters.
        - deployment_scheme: Implement a user-specific / simulation-
        specific deployment scheme used to deploy organisms into the
        World. This function will only be used when "deployment_code" in
        simulation parameters dictionary equals to 0.

    @param sim_parameters: Dictionary of simulation parameters
    @param sim_functions: A class inherited from dose.dose_functions
    class to implement all the needed simulation functions.
    '''
    print('\n[' + sim_parameters["simulation_name"].upper() + ' SIMULATION]')
    if "initial_chromosome" not in sim_parameters:
        print('Adding initial chromosome to simulation parameters...')
        sim_parameters["initial_chromosome"] = ['0'] * \
                                    sim_parameters["chromosome_size"]
    print('Adding deployment scheme to simulation parameters...')
    sim_parameters["deployment_scheme"] = sim_functions.deployment_scheme
    print('Constructing World entity...')
    World = dose_world.World(sim_parameters["world_x"],
                             sim_parameters["world_y"],
                             sim_parameters["world_z"])
    print('Spawning populations...')
    Populations = spawn_populations(sim_parameters)
    print('\nStarting simulation core...')
    simulation_core(sim_functions, sim_parameters, Populations, World)
```

File name: dose_world.py

```
'''
World structure for DOSE (digital organism simulation environment)
Date created: 13th September 2012

Reference: Ling, MHT. 2012. An Artificial Life Simulation Library Based on
Genetic Algorithm, 3-Character Genetic Code and Biological Hierarchy. The
Python Papers 7: 5.
'''
import copy

class World(object):
    '''
    Representation of a 3-dimensional ecological world.

    The ecosystem is made up of ecological cells. Each ecological cell is
    modelled as a dictionary of
        - local_input: A list containing processed input, representing
          the partial local ecological condition, to be used as input to
          the organisms in the current ecological cell. This is updated
          by World.update_local function.
        - local_output: A list containing processed output, representing
          the partial local ecological condition. This is updated by
          World.update_local function.
        - temporary_input: A list acting as temporary holding for input
```

```
          after being fed to the organisms in the current ecological
          cell, which is to be used to update local_input and local_output
          lists by World.update_local and World.update_ecology functions.
        - temporary_output: A list acting as temporary holding for output
          from the organisms in the current ecological cell, which is to
          be used to update local_input and local_output lists by
          World.update_local and World.update_ecology functions.
        - organisms: The number of organisms in the current ecological
          cell which is updated by World.organism_movement and
          World.organism_location functions.

    @see: Ling, MHT. 2012. An Artificial Life Simulation Library Based on
    Genetic Algorithm, 3-Character Genetic Code and Biological Hierarchy.
    The Python Papers 7: 5.
    '''


    def __init__(self, world_x, world_y, world_z):
        '''
        Setting up the world and ecosystem

        @param world_x: number of ecological cells on the x-axis
        @type world_x: integer
        @param world_y: number of ecological cells on the y-axis
        @type world_y: integer
        @param world_z: number of ecological cells on the z-axis
        @type world_z: integer
        '''
        self.ecosystem = {}

        eco_cell = {'local_input': [], 'local_output': [],
                    'temporary_input': [], 'temporary_output': [],
                    'organisms': 0}
        self.world_x = int(world_x)
        self.world_y = int(world_y)
        self.world_z = int(world_z)
        for x in range(self.world_x):
            eco_x = {}
            for y in range(self.world_y):
                eco_y = {}
                for z in range(self.world_z):
                    eco_y[z] = copy.deepcopy(eco_cell)
                eco_x[y] = copy.deepcopy(eco_y)
            self.ecosystem[x] = copy.deepcopy(eco_x)

    def eco_burial(self, filename):
        '''
        Function to preserve the entire ecosystem.

        @param filename: file name of preserved ecosystem.
        '''

        # In Python 3, cPickle is no longer needed: Py3 looks for
        # an optimized version, and if it founds none, will load the
        # pure python implementation of pickle.
        try:
            import cPickle as pickle
        except ImportError:
            import pickle

        f = open(filename, 'wb')
        pickle.dump(self.ecosystem, f)
        f.close()

    def eco_excavate(self, filename):
        '''
        Function to excavate entire ecosystem.

        @param filename: file name of preserved ecosystem.
        '''
        # In Python 3, cPickle is no longer needed: Py3 looks for
        # an optimized version, and if it founds none, will load the
        # pure python implementation of pickle.
        try:
            import cPickle as pickle
        except ImportError:
            import pickle

        self.ecosystem = pickle.load(open(filename, 'rb'))

    def ecoregulate(self):
        '''
        Function to simulate events to the entire ecosystem. B{This
        function may be over-ridden by the inherited class or substituted
        to cater for ecological schemes but not an absolute requirement
        to do so.}
        '''
        pass

    def organism_movement(self, x, y, z):
        '''
        Function to trigger organism movement from current ecological cell
        to an adjacent ecological cell. B{This function may be over-ridden
        by the inherited class or substituted to cater for mobility
        schemes but not an absolute requirement to do so.}

        @param x: location of current ecological cell on the x-axis
        @type x: integer
        @param y: location of current ecological cell on the y-axis
        @type y: integer
        @param z: location of current ecological cell on the z-axis
        @type z: integer
        '''
        pass
    def organism_location(self, x, y, z):
        '''
        Function to trigger organism movement from current ecological cell
        to a distant ecological cell. B{This function may be over-ridden
        by the inherited class or substituted to cater for mobility
        schemes but not an absolute requirement to do so.}

        @param x: location of current ecological cell on the x-axis
        @type x: integer
        @param y: location of current ecological cell on the y-axis
        @type y: integer
        @param z: location of current ecological cell on the z-axis
        @type z: integer
        '''
        pass

    def update_ecology(self, x, y, z):
        '''
        Function to process temporary_input and temporary_output from the
        activities of the organisms in the current ecological cell into a
        local ecological cell condition, and update the ecosystem.
        B{This function may be over-ridden by the inherited class or
        substituted to cater for ecological schemes but not an absolute
        requirement to do so.}

        @param x: location of current ecological cell on the x-axis
        @type x: integer
        @param y: location of current ecological cell on the y-axis
        @type y: integer
        @param z: location of current ecological cell on the z-axis
        @type z: integer
        '''
        pass

    def update_local(self, x, y, z):
        '''
        Function to update local ecological cell condition from the
        ecosystem.
        B{This function may be over-ridden by the inherited class or
        substituted to cater for ecological schemes but not an absolute
        requirement to do so.}

        @param x: location of current ecological cell on the x-axis
        @type x: integer
        @param y: location of current ecological cell on the y-axis
        @type y: integer
        @param z: location of current ecological cell on the z-axis
        @type z: integer
        '''
        pass

    def report(self):
        '''
        Function to report the status of the world and ecosystem. B{This
        function may be over-ridden by the inherited class or substituted
        to cater for specific reporting schemes but not an absolute
        requirement to do so.}

        @return: dictionary of status describing the current generation
        '''
        pass
```

File name: database_calls.py

```
'''
File containing support functions for database logging of simulations.

Date created: 10th October 2013
'''
import os, copy
import sqlite3 as s

def connect_database(dbpath, sim_parameters=None):
    '''
    Connects to logging database and prepares database for use, if
    database does not exist. This function can be used to connect to the
    logging database using a file path or using simulation parameters
    dictionary.

    Simulation parameters dictionary takes precedence - if simulation
    parameters dictionary (sim_parameters) is not None, this function will
    look for database file in <simulation execution directory>/Simulations/
    <database file>.

    In both cases, if the database is not present, it will create a SQLite3
    database file and create the required database tables.

    @param dbpath: File path of logging database. This parameter will only
    be used when sim_parameters == None.
    @param sim_parameters: Dictionary of simulation parameters. Default is
    None.
    @return: (con, cur) where
        - con = connector
        - cur = cursor
    '''
    if sim_parameters:
        dbpath = os.sep.join([os.getcwd(),
                              'Simulations',
                              sim_parameters["database_file"]])
    con = s.connect(dbpath)
    cur = con.cursor()
    cur.execute('''
        create table if not exists parameters
            (start_time text, simulation_name text,
             key text, value text)''')
    cur.execute('''
        create table if not exists organisms
            (start_time text, pop_name text,
             org_name text, generation text,
             key text, value text)''')
    cur.execute('''
        create table if not exists world
            (start_time text, x text, y text,
             z text, generation text,
             key text, value text)''')
    cur.execute('''
        create table if not exists miscellaneous
            (start_time text, generation text,
             key text, value text)''')
    cur.execute('''
        create index if not exists organisms_index1 on organisms
            (pop_name, generation)''')
    cur.execute('''
        create index if not exists organisms_index2 on organisms
            (generation, org_name)''')
    con.commit()
    return (con, cur)

def db_log_simulation_parameters(con, cur, sim_parameters):
    '''
    Function to log the simulation parameters. A database table, parameters,
    had been defined for simulation parameters logging where the structure
    is (start_time text, simulation_name text, key text, value text). The
    starting time of current simulation (start_time) and name of the
    current simulation (simulation_name) are used as complex primary key
    to identify the current simulation. All parameters will be logged,
    except for "starting_time" and "simulation_name".

    The following transformations of data are made:
        - Population name (key = "population_names") is a list of names.
        These are concatenated and delimited by '|'. For example,
        ['pop_01', 'pop_02'] ==> pop_01|pop_02
        - Genetic bases (key = "chromosome_bases") is a list of bases to
        make up the genetic code. These are concatenated and delimited by
        '|'. For example, ['1', '2'] ==> 1|2
        - Ragaraja instructions to be used (key = "ragaraja_instructions")
        is a list of 3-character Ragaraja instructions in numbers. These
        are concatenated and delimited by '|'. For example, ['000', '004',
        '008']  ==> 000|004|008
        - Initial (ancestral) chromosome is a list of bases. These bases
        are concatenated with no delimiter. For example, [1, 2, 3] ==> 123

    @param con: Database connector from connect_database() function.
    @param cur: Database cursor from connect_database() function.
    @param sim_parameters: Dictionary of parameters used in simulation.
    @return: (con, cur) where
        - con = connector
        - cur = cursor
    '''
    start_time = sim_parameters["starting_time"]
    simulation_name = sim_parameters["simulation_name"]
    for key in [k for k in list(sim_parameters.keys())
                if k not in ("simulation_name", "starting_time")]:
        value = sim_parameters[key]
        if key in ("population_names", "chromosome_bases",
                   "ragaraja_instructions"):
            value = '|'.join([str(x) for x in value])
            cur.execute('''insert into parameters values (?,?,?,?)''',
                         (str(start_time), str(simulation_name),
                          str(key), value))
        elif key in ("initial_chromosome"):
            value = ''.join(value)
            cur.execute('''insert into parameters values (?,?,?,?)''',
                        (str(start_time), str(simulation_name),
                         str(key), value))
        else:
            cur.execute('''insert into parameters values (?,?,?,?)''',
                        (str(start_time), str(simulation_name),
                         str(key), str(value)))
    con.commit()
    return (con, cur)

def db_report(con, cur, sim_functions, start_time,
              Populations, World, generation_count):
    '''
    Wrapper around user-implemented database logging which over-rides
    dose.dose_functions.database_report() function.

    @param con: Database connector from connect_database() function.
    @param cur: Database cursor from connect_database() function.
    @param sim_functions: Object of simulation-specific functions, which
    inherits dose.dose_functions class.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @param Populations: A dictionary containing one or more populations
    where the value is a genetic.Population object.
    @param World: dose_world.World object.
    @param generation_count: Current number of generations simulated.
    @return: (con, cur) where
        - con = connector
        - cur = cursor
    '''
    sim_functions.database_report(con, cur, start_time,
                                  Populations, World, generation_count)
    con.commit()
    return (con, cur)

def db_list_simulations(cur, table='parameters'):
    '''
    Function to list simulations, identified by starting time of the
    simulation (and simulation name, if available), from logging database.

    @param cur: Database cursor from connect_database() function.
    @param table: Database table name to list simulations. Allowable values
    are 'parameters', 'organisms', 'world', and 'miscellaneous'. Default
    value is 'parameters'.
    @return: A list containing the results.
        - If table = 'parameters', the returned list will be a list of list
        (consisting of [starting time of simulation, simulation name]).
        - If table is not 'parameters', the returned list will be a list of
        starting time of simulation.
    '''
    if table not in ('parameters', 'organisms',
                     'world', 'miscellaneous'):
        table = 'parameters'
    if table == 'parameters':
        cur.execute("""select distinct start_time, simulation_name
                    from parameters""")
        return [[str(x[0]), str(x[1])] for x in cur.fetchall()]
    else:
        cur.execute("select distinct start_time from %s", table)
        return [str(x[0]) for x in cur.fetchall()]

def db_list_generations(cur, start_time, table='organisms'):
    '''
    Function to list all logged generations within a simulation, identified
    by starting time of the simulation.

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @param table: Database table name to list simulations. Allowable values
    are 'parameters', 'organisms', 'world', and 'miscellaneous'. Default
    value is 'organisms'.
    @return: A list containing the results (generation counts).
    '''
    # query plan: SCAN TABLE organisms USING COVERING INDEX organisms_index2
    cur.execute("select distinct generation from %s where start_time='%s'"
                % (str(table), str(start_time)))
    generations = sorted([int(str(x[0])) for x in cur.fetchall()])
    return [str(gen) for gen in generations]

def db_list_datafields(cur, start_time, table='organisms'):
    '''
    Function to list all logged data fields (types of data logged) within
    a simulation, identified by starting time of the simulation.

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @param table: Database table name to list simulations. Allowable values
    are 'parameters', 'organisms', 'world', and 'miscellaneous'. Default
    value is 'organisms'.
    @return: A list containing the results (types of data logged).
    '''
    cur.execute("select distinct key from %s where start_time='%s'"
                % (str(table), str(start_time)))
    return [str(x[0]) for x in cur.fetchall()]

def db_list_population_name(cur, start_time):
    '''
    Function to list all logged populations (as population names) within a
    simulation, identified by starting time of the simulation.

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @return: A list containing the results (types of data logged).
    '''
    # query plan: SCAN TABLE organisms USING COVERING INDEX organisms_index1
    cur.execute("select distinct pop_name from organisms where \
                start_time='%s'" % str(start_time))
    return [str(x[0]) for x in cur.fetchall()]

def db_get_ecosystem(cur, start_time, datafield='all', generation='all'):
    '''
    Analysis helper function to get a specific field of the ecosystem
    (World.ecosystem) or the entire ecosystem for one or more generations
    within a simulation (as identified by the starting time of the
    simulation).

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @param datafield: The specific datafield in World.ecosystem to extract.
    Predefined datafields for ecosystem are 'local_input', 'local_output',
    'temporary_input', 'temporary_output', and 'organisms'. Default = 'all',
    which returns the entire World.ecosystem(s) of the generation(s). Only
    one or all datafield(s) in World.ecosystem can be extracted at a time.
    @type datafield: string
    @param generation: Generation(s) to extract World.ecosystem. Default =
    'all', which extracts World.ecosystems for all logged generations within
    the specific simulation.
    @type generation: list
    @return: A dictionary of results - {<generation count>: <value>}. The
    value is the value of the specific datafield if a specific datafield
    is required (the data type of this value is dependent on that of the
    data type of World.ecosystem[datafield]) or the entire ecosystem as a
    dictionary.
    '''
    if generation == 'all':
        generation = db_list_generations(cur, start_time, 'world')
    else:
        generation = [str(x) for x in generation]
    results = {}
    for gen in generation:
        results[gen] = {}
        ecosystem = db_reconstruct_world(cur, start_time, gen).ecosystem
        if datafield == 'all':
            results[gen] = ecosystem
        else:
            for x in list(ecosystem.keys()):
                results[gen][x] = {}
                for y in list(ecosystem[x].keys()):
                    results[gen][x][y] = {}
                    for z in list(ecosystem[x][y].keys()):
                        results[gen][x][y][z] = ecosystem[x][y][z][datafield]
    return results

def db_get_organisms_status(cur, start_time, population_name,
                            datafield='all', generation='all'):
    '''
    Analysis helper function to get a specific field of the Organism.status
    dictionary or the entire Organism.status dictionary for one or more
    generations within a simulation (as identified by the starting time of
    the simulation).

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @param population_name: Name of the population.
    @param datafield: The specific key in Organism.status dictionary to
    extract. Predefined keys for Organism.status dictionary are'age',
    'alive', 'blood', 'death', 'deme', 'fitness', 'gender', 'identity',
    'lifespan', 'location', 'parents', 'vitality'. Default = 'all', which
    returns the entire Organism.status dictionaries of the generation(s).
    Only one or all datafield(s) in Organism.status dictionary can be
    extracted at a time.
    @type datafield: string
    @param generation: Generation(s) to extract Organism.status dictionary.
    Default = 'all', which extracts Organism.status dictionary for all
    logged generations within the specific simulation.
    @type generation: list
    @return: A dictionary of dictionary of results -
    {<generation count>: {<Organism identity>: <value>}}. The value is the
    value of the specific datafield if a specific datafield is required
    (the data type of this value is dependent on that of the data type of
    Organism.status[datafield]) or the entire Organism.status as a dictionary.
    '''
    if generation == 'all':
        generation = db_list_generations(cur, start_time, 'organisms')
    else:
        generation = [str(x) for x in generation]
    results = {}
    for gen in generation:
        results[gen] = {}
        agents = db_reconstruct_organisms(cur, start_time,
                                          population_name, gen)
        for org in agents:
            identity = org.status['identity']
            if datafield == 'all':
                results[gen][identity] = org.status
            else:
                results[gen][identity] = org.status[datafield]
    return results

def db_get_organisms_genome(cur, start_time,
                            population_name, generation='all'):
    '''
    Analysis helper function to get entire genome of organisms for one or
    more generations within a simulation (as identified by the starting
    time of the simulation).

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @param population_name: Name of the population.
    @param generation: Generation(s) to extract Organism.genome.
    Default = 'all', which extracts Organism.genome for all logged
    generations within the specific simulation.
    @type generation: list
    @return: A dictionary of dictionary of genome -
    {<generation count>: {<Organism identity>: <genome as list of chromosomes>}}.
    '''
    if generation == 'all':
        generation = db_list_generations(cur, start_time, 'organisms')
    else:
        generation = [str(x) for x in generation]
    results = {}
    for gen in generation:
        results[gen] = {}
        agents = db_reconstruct_organisms(cur, start_time,
                                          population_name, gen)
        for org in agents:
            identity = org.status['identity']
            results[gen][identity] = org.genome
    return results

def db_get_organisms_chromosome_sequences(cur, start_time,
                                          population_name,
                                          generation='all'):
    '''
    Analysis helper function to get chromosomal sequences of entire genome
    of organisms for one or more generations within a simulation (as
    identified by the starting time of the simulation).

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @param population_name: Name of the population.
    @param generation: Generation(s) to extract Organism.genome.
    Default = 'all', which extracts Organism.genome for all logged
    generations within the specific simulation.
    @type generation: list
    @return: A dictionary of dictionary of chromosomal sequences -
    {<generation count>: {<Organism identity>: <list (all chromosomes) of
    list (individual chromosome) of chromosomal sequence>}}.
    '''
    genome_dict = db_get_organisms_genome(cur, start_time,
                                          population_name, generation)
    results = {}
    for gen in list(genome_dict.keys()):
        results[gen] = {}
        for identity in list(genome_dict[gen].keys()):
            results[gen][identity] = [chromosome.sequence
                                for chromosome in genome_dict[gen][identity]]
    return results

def db_reconstruct_simulation_parameters(cur, start_time):
    '''
    Function to reconstruct simulation parameters dictionary of a
    simulation (as identified by the starting time of the simulation).

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @return: Simulation parameters dictionary containing the following keys:
        - simulation_name
        - population_names
        - population_locations
        - deployment_code
        - chromosome_bases
        - background_mutation
        - additional_mutation
        - mutation_type
        - chromosome_size
        - genome_size
        - max_tape_length
        - clean_cell
        - interpret_chromosome
        - max_codon
        - population_size
        - eco_cell_capacity
        - world_x
        - world_y
        - world_z
        - goal
        - maximum_generations
        - fossilized_ratio
        - fossilized_frequency
        - print_frequency
        - ragaraja_version
        - ragaraja_instructions
        - eco_buried_frequency
        - database_file
        - database_logging_frequency
    '''
    parameters = {}
    for key in ('simulation_name', 'population_names',
                'population_locations', 'deployment_code',
                'chromosome_bases', 'background_mutation',
                'additional_mutation', 'mutation_type',
                'chromosome_size', 'genome_size', 'max_tape_length',
                'clean_cell', 'interpret_chromosome', 'max_codon',
                'population_size', 'eco_cell_capacity',
                'world_x', 'world_y', 'world_z', 'goal',
                'maximum_generations', 'fossilized_ratio',
                'fossilized_frequency', 'print_frequency',
                'ragaraja_version', 'ragaraja_instructions',
                'eco_buried_frequency', 'database_file',
                'database_logging_frequency'):
            parameters[key] = None
    start_time = str(start_time)
    cur.execute("select distinct simulation_name from parameters where \
    start_time = '%s'" % start_time)
    parameters["simulation_name"] = str(cur.fetchone()[0])
    cur.execute("select key, value from parameters where \
                start_time = '%s'" % start_time)
    for r in cur.fetchall():
        if str(r[0]) == 'population_names':
            value = str(r[1]).split('|')
            exec("parameters['population_names'] = %s" % str(value))
        elif str(r[0]) == 'population_locations':
            exec("parameters['population_locations'] = %s" % str(r[1]))
        elif str(r[0]) == 'deployment_code':
            parameters['deployment_code'] = int(r[1])
        elif str(r[0]) == 'chromosome_bases':
            value = str(r[1]).split('|')
            exec("parameters['chromosome_bases'] = %s" % str(value))
        elif str(r[0]) == 'background_mutation':
            parameters['background_mutation'] = float(r[1])
        elif str(r[0]) == 'additional_mutation':
            parameters['additional_mutation'] = float(r[1])
        elif str(r[0]) == 'mutation_type':
            parameters['mutation_type'] = str(r[1])
        elif str(r[0]) == 'chromosome_size':
            parameters['chromosome_size'] = int(r[1])
        elif str(r[0]) == 'genome_size':
            parameters['genome_size'] = int(r[1])
        elif str(r[0]) == 'max_tape_length':
            parameters['max_tape_length'] = int(r[1])
        elif str(r[0]) == 'clean_cell':
            exec("parameters['clean_cell'] = %s" % str(r[1]))
        elif str(r[0]) == 'interpret_chromosome':
            exec("parameters['interpret_chromosome'] = %s" % str(r[1]))
        elif str(r[0]) == 'max_codon':
            parameters['max_codon'] = int(r[1])
        elif str(r[0]) == 'population_size':
            parameters['population_size'] = int(r[1])
        elif str(r[0]) == 'eco_cell_capacity':
            parameters['eco_cell_capacity'] = int(r[1])
        elif str(r[0]) == 'world_x':
            parameters['world_x'] = int(r[1])
        elif str(r[0]) == 'world_y':
            parameters['world_y'] = int(r[1])
        elif str(r[0]) == 'world_z':
            parameters['world_z'] = int(r[1])
        elif str(r[0]) == 'goal':
            try: parameters['goal'] = float(r[1])
            except ValueError: exec("parameters['goal'] = %s" % str(r[1]))
        elif str(r[0]) == 'maximum_generations':
            parameters['maximum_generations'] = int(r[1])
        elif str(r[0]) == 'fossilized_ratio':
            parameters['fossilized_ratio'] = float(r[1])
        elif str(r[0]) == 'fossilized_frequency':
            parameters['fossilized_frequency'] = int(r[1])
        elif str(r[0]) == 'print_frequency':
            parameters['print_frequency'] = int(r[1])
        elif str(r[0]) == 'ragaraja_version':
            version = str(r[1])
            if version == '0.1': parameters['ragaraja_version'] = 0.1
            else: parameters['ragaraja_version'] = int(r[1])
        elif str(r[0]) == 'ragaraja_instructions':
            value = str(r[1]).split('|')
            exec("parameters['ragaraja_instructions'] = %s" % str(value))
        elif str(r[0]) == 'eco_buried_frequency':
            parameters['eco_buried_frequency'] = int(r[1])
        elif str(r[0]) == 'database_file':
            parameters['database_file'] = str(r[1])
        elif str(r[0]) == 'database_logging_frequency':
            parameters['database_logging_frequency'] = int(r[1])
    return parameters

def db_reconstruct_world(cur, start_time, generation):
    '''
    Function to reconstruct the world object of a simulation (as identified
    by the starting time of the simulation) at a specific generation.

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @param generation: Current number of generations simulated.
    @return: dose_world.World object
    '''
    import dose_world
    eco_cell = {'local_input': [], 'local_output': [],
                'temporary_input': [], 'temporary_output': [],
                'organisms': 0}
    cur.execute("select max(x), max(y), max(z) from world where \
    start_time = '%s'" % start_time)
    coordinates = cur.fetchone()
    world_x = int(coordinates[0]) + 1
    world_y = int(coordinates[1]) + 1
    world_z = int(coordinates[2]) + 1
    ecosystem = {}
    for x in range(world_x):
        ecosystem[x] = {}
        for y in range(world_y):
            ecosystem[x][y] = {}
            for z in range(world_z):
                ecosystem[x][y][z] = {}
                for key in ('local_input', 'local_output',
                            'temporary_input', 'temporary_output',
                            'organisms'):
                    ecosystem[x][y][z][key] = None
    start_time = str(start_time)
    generation = str(generation)
    # query plan: SCAN TABLE world
    cur.execute("select x, y, z, key, value from world where \
    start_time =  '%s' and generation = '%s'" % (start_time, generation))
    for r in cur.fetchall():
        x = int(r[0])
        y = int(r[1])
        z = int(r[2])
        key = str(r[3])
        value = str(r[4])
        exec("ecosystem[%i][%i][%i]['%s'] = %s" % (x, y, z, key, value))
    World = dose_world.World(1, 1, 1)
    World.ecosystem = ecosystem
    return World

def db_reconstruct_organisms(cur, start_time, population_name, generation):
    '''
    Function to reconstruct a list of organisms (genetic.Organism objects)
    of a population within a simulation (as identified by the starting time
    of the simulation) at a specific generation.

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @param population_name: Name of the population
    @param generation: Current number of generations simulated.
    @return: A list of Organisms (genetic.Organism objects)
    '''
    import genetic as g
    start_time = str(start_time)
    population_name = str(population_name)
    generation = str(generation)
    # query plan: SEARCH TABLE organisms USING INDEX
    #              organisms_index2 (generation=?)
    cur.execute("select distinct org_name from organisms where \
    start_time = '%s' and pop_name = '%s' and generation = '%s'" %
    (start_time, population_name, generation))
    names = [x[0] for x in cur.fetchall()]
    agents = [0] * len(names)
    for i in range(len(names)):
        org_name = str(names[i])
        org = g.Organism()
        org.genome = []
        org.status['identity'] = org_name
        # query plan: SEARCH TABLE organisms USING INDEX
        #              organisms_index2 (generation=? AND org_name=?)
        cur.execute("select key, value from organisms where \
        generation = '%s' and org_name = '%s' and \
        start_time = '%s' and pop_name = '%s'" %
        (generation, org_name, start_time, population_name))
        for r in cur.fetchall():
            key = str(r[0])
            value = str(r[1])
            if key == 'alive':
                exec("org.status['alive'] = %s" % str(value))
            elif key == 'vitality':
                org.status['vitality'] = float(value)
            elif key == 'parents':
                if value == '': value = '[]'
                else: value = value.split('|')
                exec("org.status['parents'] = %s" % str(value))
            elif key == 'age':
                org.status['age'] = float(value)
            elif key == 'gender':
                exec("org.status['gender'] = %s" % str(value))
            elif key == 'lifespan':
                org.status['lifespan'] = float(value)
            elif key == 'fitness':
                exec("f = '%s'" % str(value))
                if type(f) == type(1) or type(f) == type(1.0):
                    org.status['fitness'] = float(value)
                else:
                    exec("org.status['fitness'] = %s" % str(value))
            elif key == 'blood':
                if value == '': value = '[]'
                else: value = value.split('|')
                exec("org.status['blood'] = %s" % str(value))
            elif key == 'deme':
                org.status['deme'] = str(value)
            elif key == 'location':
                value = tuple([int(x) for x in value.split('|')])
                exec("org.status['location'] = %s" % str(value))
            elif key == 'death':
                exec("org.status['death'] = %s" % str(value))
            elif key.startswith('chromosome'):
                chr_position = key.split('_')[1]
                sequence = [str(x) for x in str(value)]
                cur.execute("select value from parameters where \
                    key='background_mutation' and start_time='%s'" %
                    start_time)
                background_mutation = float(cur.fetchone()[0])
                cur.execute("select value from parameters where \
                    key='chromosome_bases' and start_time='%s'" %
                    start_time)
                bases = str(cur.fetchone()[0]).split('|')
                exec("chromosome_bases = %s" % bases)
                chromosome = g.Chromosome(sequence, chromosome_bases,
                                          background_mutation)
                org.genome.append(chromosome)
            else:
                exec("org.status['%s'] = %s" % (str(key), str(value)))
        agents[i] = org
    return agents

def db_reconstruct_population(cur, start_time,
                              population_name, generation):
    '''
    Function to reconstruct a population within a simulation (as identified
    by the starting time of the simulation) at a specific generation.

    @param cur: Database cursor from connect_database() function.
    @param start_time: Starting time of current simulation in the format
    of <date>-<seconds since epoch>; for example, 2013-10-11-1381480985.77.
    @param population_name: Name of the population
    @param generation: Current number of generations simulated.
    @return: genetic.Population object
    '''
    import genetic
    agents = db_reconstruct_organisms(cur, start_time,
                                      population_name, generation)
    cur.execute("select value from parameters where \
                key='goal' and start_time='%s'" % start_time)
    g = cur.fetchone()[0]
    try: goal = float(g)
    except ValueError: exec("goal = %s" % str(g))
    return genetic.Population(goal, 1e24, agents)
```

File name: simulation_calls.py

```
'''
File containing support functions for running a simulation. The functions
in this file is not for public use; all functions in this file are private
functions.

Date created: 10th October 2013
'''
import random, inspect, os
import os.path
from datetime import datetime
from time import time
from copy import deepcopy
from shutil import copyfile

# In Python 3, cPickle is no longer needed: Py3 looks for
# an optimized version, and if it founds none, will load the
# pure python implementation of pickle.
try:
    import cPickle as pickle
except ImportError:
    import pickle

import dose_world
import genetic
import ragaraja, register_machine

from database_calls import connect_database, db_log_simulation_parameters
from database_calls import db_report
```

```
def simulation_core(sim_functions, sim_parameters, Populations, World):
    '''
    Sequential ecological cell DOSE simulator.

    Performs the following operations:
        - Creating simulation file directory for results text file, population
        freeze, and world burial storage
        - Generate a simulation start time to identify the current simulation
        - Define active Ragaraja instructions
        - Connecting to logging database (if needed)
        - Writing simulation parameters into results text file
        - Initialize World and Population
        - Deploy population(s) onto the world
        - Run the simulation and recording the results
        - Writing final results into results text file
        - Close logging database (if used)
        - Copy simulation script into simulation file directory

    @param sim_functions: implemented simulation functions (see
    dose.dose_functions)
    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param Populations: dictionary of population objects
    @param World: dose_world.World object
    '''
    time_start = '-'.join([str(datetime.utcnow()).split(' ')[0],
                           str(time())])
    print('Creating simulation file directories...')
    directory ='_'.join([sim_parameters["simulation_name"],time_start])
    directory = os.sep.join([os.getcwd(), 'Simulations', directory])
    directory = directory + os.sep
    if not os.path.exists(directory): os.makedirs(directory)
    print('Adding simulation directory to simulation parameters...')
    sim_parameters["directory"] = directory
    print('Adding starting time to simulation parameters...')
    sim_parameters["starting_time"] = time_start
    sim_functions = sim_functions()
    if sim_parameters["ragaraja_version"] == 0:
        print('Activating ragaraja version: 0...')
        ragaraja.activate_version(sim_parameters["ragaraja_version"],
                                  sim_parameters["ragaraja_instructions"])
    else:
        print('Activating ragaraja version: ' + \
            str(sim_parameters["ragaraja_version"]) + '...')
        ragaraja.activate_version(sim_parameters["ragaraja_version"])
    if "database_file" in sim_parameters and \
        "database_logging_frequency" in sim_parameters:
        print('Connecting to database file: ' + \
            sim_parameters["database_file"] + '...')
        (con, cur) = connect_database(None, sim_parameters)
        print('Logging simulation parameters to database file...')
        (con, cur) = db_log_simulation_parameters(con, cur, sim_parameters)
    for pop_name in Populations:
        print('\nPreparing population: ' + pop_name + ' for simulation...')
        if 'sim_folder' in sim_parameters or \
            'database_source' in sim_parameters:
            print('Calculating final generation count...')
            max = sim_parameters["rev_start"] \
                [sim_parameters["population_names"].index(pop_name)] + \
                sim_parameters["extend_gen"]
            print('Updating generation count from previous simulation...')
            generation_count = sim_parameters["rev_start"][0]
        else:
            print('Deploying population in World entity...')
            if sim_parameters["deployment_code"] == 0:
                print('Executing user defined deployment scheme...')
                deploy_0(sim_parameters, Populations, pop_name, World)
            elif sim_parameters["deployment_code"] == 1:
                print('Executing deployment code 1: Single eco-cell deployment...')
                deploy_1(sim_parameters, Populations, pop_name, World)
            elif sim_parameters["deployment_code"] == 2:
                print('Executing deployment code 2: Random eco-cell deployment...')
                deploy_2(sim_parameters, Populations, pop_name, World)
            elif sim_parameters["deployment_code"] == 3:
                print('Executing deployment code 3: Even eco-cell deployment...')
                deploy_3(sim_parameters, Populations, pop_name, World)
            elif sim_parameters["deployment_code"] == 4:
                print('Executing deployment code 4: Centralized eco-cell 
deployment...')
                deploy_4(sim_parameters, Populations, pop_name, World)
            print('Adding maximum generations to simulation parameters...')
            max = sim_parameters["maximum_generations"]
            generation_count = 0
        print('Writing simulation parameters into txt file report...')
        write_parameters(sim_parameters, pop_name)
        print('Updating generation count...')
        Populations[pop_name].generation = generation_count
    print('\nSimulation preparation complete...')
    while generation_count < max:
        generation_count = generation_count + 1
        sim_functions.ecoregulate(World)
        eco_cell_iterator(World, sim_parameters,
                          sim_functions.update_ecology)
        eco_cell_iterator(World, sim_parameters,
                          sim_functions.update_local)
        eco_cell_iterator(World, sim_parameters, sim_functions.report)
        bury_world(sim_parameters, World, generation_count)
        for pop_name in Populations:
            if sim_parameters["interpret_chromosome"]:
                interpret_chromosome(sim_parameters, Populations,
                                     pop_name, World)
            report_generation(sim_parameters, Populations, pop_name,
                              sim_functions, generation_count)
            sim_functions.organism_movement(Populations, pop_name, World)
            sim_functions.organism_location(Populations, pop_name, World)
        if "database_file" in sim_parameters and \
            "database_logging_frequency" in sim_parameters and \
            generation_count % \
            int(sim_parameters["database_logging_frequency"]) == 0:
                (con, cur) = db_report(con, cur, sim_functions,
                                       sim_parameters["starting_time"],
                                       Populations, World, generation_count)
        print('Generation ' + str(generation_count) + ' complete...')
    print('\nClosing simulation results...')
    for pop_name in Populations: close_results(sim_parameters, pop_name)
    if "database_file" in sim_parameters and \
        "database_logging_frequency" in sim_parameters:
        print('Committing logged data into database file...')
        con.commit()
        print('Terminating database connection...')
        con.close()
    print('Copying simulation file script to simulation results directory...')

    #DV on my system, the inspect.stack()[2][1] value returns the full
    #   path to the file ('/home/douwe/scr/.../scriptname.py').
    #   The end result is an error for the original code, below, as the
    #   copyfile command is pointed to a wrong location: the directory
    #   is added twice. This might be Py3, or Windows functionality.
    #   With using the os.path module, this should give an OS-independent
    #   way of extracting the basename and linking to the directory.
    sim_script_basename = os.path.basename(inspect.stack()[2][1])
    copyfile(inspect.stack()[2][1],
#DV_original#             sim_parameters['directory'] + inspect.stack()[2][1])
             os.path.join(sim_parameters['directory'], sim_script_basename))
    print('\nSimulation ended...')

def coordinates(location):
    '''
    Helper function to transpose ecological cell into a tuple.

    @param location: location of ecological cell as 3-element iterable
    data type
    @return: location of ecological cell as (x,y,z)
    '''
    x = location[0]
    y = location[1]
    z = location[2]
    return (x,y,z)

def adjacent_cells(sim_parameters, location):
    '''
    Function to get a list of adjacent ecological cells from a given
    location.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param location: location of ecological cell as (x,y,z)
    @return: list of locations of adjacent cells.
    '''
    trashbin = []
    temp_cells = []
    world_size = [sim_parameters["world_x"],
                  sim_parameters["world_y"],
                  sim_parameters["world_z"]]
    for i in range(3):
        new_location = [spot for spot in location]
        new_location[0] += 1
        temp_cells.append(new_location)
        new_location = [spot for spot in location]
        new_location[0] -= 1
        temp_cells.append(new_location)
    for i in range(2):
        new_location = [spot for spot in location]
        new_location[1] -= 1
        temp_cells.append(new_location)
    for i in range(0,4,3):
        temp_cells[i][1] += 1
        temp_cells[i+1][1] -= 1
    temp_cells[-1][1] += 2
    for i in range(8):
        for x in range(2):
            if temp_cells[i][x] >= world_size[x] or temp_cells[i][x] < 0:
                if temp_cells[i] not in trashbin:
                    trashbin.append(temp_cells[i])
    for location in trashbin:
        temp_cells.remove(location)
    return [tuple(location) for location in temp_cells]

def spawn_populations(sim_parameters):
    '''
    Initializing starting population(s) for a simulation. Each organism
    in each population at this stage will be genetic clones of each
    other.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @return: dictionary of population objects with population name as key
    '''
    temp_Populations = {}
    print(' - Accessing population names...')
    for pop_name in sim_parameters["population_names"]:
        print(' - Constructing population: ' + pop_name + '...')
        temp_Populations[pop_name] = \
            genetic.population_constructor(sim_parameters)
        print(' - Updating organism identity and deme status...')
        for individual in temp_Populations[pop_name].agents:
            individual.generate_name()
            individual.status['deme'] = pop_name
    return temp_Populations

def eco_cell_iterator(World, sim_parameters, function):
    '''
    Generic caller to call any function to be executed in each ecological
    cell in sequence.

    @param World: dose_world.World object
    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param function: function to be executed
    @return: none
    '''
    for x in range(sim_parameters["world_x"]):
        for y in range(sim_parameters["world_y"]):
            for z in range(sim_parameters["world_z"]):
                if len(inspect.getargspec(function)[0]) == 5:
                    function(World, x, y, z)
                else:
                    function(World)

def deploy_0(sim_parameters, Populations, pop_name, World):
    '''
    Organism deployment scheme 0 - User defined deployment scheme. This is
    called when "deployment_code" in simulation parameters = 0. User will
    have to provide a deployment scheme/function as "deployment_scheme" in
    simulation parameters.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param Populations: dictionary of population objects
    @param pop_name: population name
    @param World: dose_world.World object
    @return: none.
    '''
    sim_parameters["deployment_scheme"](Populations, pop_name, World)

def deploy_1(sim_parameters, Populations, pop_name, World):
    '''
    Organism deployment scheme 1 - Single ecological cell deployment
    scheme. This is called when "deployment_code" in simulation parameters
    = 1. In this scheme, all organisms in the specified population
    (specified by population name) will be deployed in the ecological cell
    specified in "population_locations" of simulation parameters.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param Populations: dictionary of population objects
    @param pop_name: population name
    @param World: dose_world.World object
    @return: none.
    '''
    position = sim_parameters["population_names"].index(pop_name)
    locations = [location
        for location in sim_parameters["population_locations"][position]]
    (x,y,z) = coordinates(locations[0])
    World.ecosystem[x][y][z]['organisms'] = sim_parameters["population_size"]
    for individual in Populations[pop_name].agents:
        individual.status['location'] = locations[0]

def deploy_2(sim_parameters, Populations, pop_name, World):
    '''
    Organism deployment scheme 2 - Random deployment scheme. This is called
    when "deployment_code" in simulation parameters = 2. In this scheme,
    all organisms in the specified population (specified by population
    name) will be randomly deployed across the list of ecological cells
    specified as "population_locations" of simulation parameters.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param Populations: dictionary of population objects
    @param pop_name: population name
    @param World: dose_world.World object
    @return: none
    '''
    position = sim_parameters["population_names"].index(pop_name)
    locations = [location
        for location in sim_parameters["population_locations"][position]]
    for individual in Populations[pop_name].agents:
            location = random.choice(locations)
            (x,y,z) = coordinates(location)
            while World.ecosystem[x][y][z]['organisms'] >= \
                sim_parameters["eco_cell_capacity"]:
                location = random.choice(locations)
                (x,y,z) = coordinates(location)
            World.ecosystem[x][y][z]['organisms'] = \
                World.ecosystem[x][y][z]['organisms'] + 1
            individual.status['location'] = location

def deploy_3(sim_parameters, Populations, pop_name, World):
    '''
    Organism deployment scheme 3 - Even deployment scheme. This is called
    when "deployment_code" in simulation parameters = 3. In this scheme,
    all organisms in the specified population (specified by population
    name) will be evenly deployed (as much as possible) across the list of
    ecological cells specified as "population_locations" of simulation
    parameters.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param Populations: dictionary of population objects
    @param pop_name: population name
    @param World: dose_world.World object
    @return: none
    '''
    position = sim_parameters["population_names"].index(pop_name)
    locations = [location
        for location in sim_parameters["population_locations"][position]]
    iterator = 0
    for i in range(sim_parameters["population_size"]):
        individual = Populations[pop_name].agents[i]
        location = locations[iterator]
        (x,y,z) = coordinates(location)
        World.ecosystem[x][y][z]['organisms'] = \
            World.ecosystem[x][y][z]['organisms'] + 1
        individual.status['location'] = location
        iterator += 1
        if iterator == len(locations):
            iterator = 0

def deploy_4(sim_parameters, Populations, pop_name, World):
    '''
    Organism deployment scheme 4 - Centralized deployment scheme. This is
    called when "deployment_code" in simulation parameters = 4. In this
    scheme, all organisms in the specified population (specified by
    population name) will be deployed onto the ecological cell specified
    in "population_locations" of simulation parameters, up to the organism
    capacity of the ecological cell (specified in "eco_cell_capacity" of
    simulation parameters). In event that the specified deployment
    locations is filled, undeployed organisms will be randomly deployed
    onto adjacent cells.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param Populations: dictionary of population objects
    @param pop_name: population name
    @param World: dose_world.World object
    @return: none
    '''
    position = sim_parameters["population_names"].index(pop_name)
    locations = [location
        for location in sim_parameters["population_locations"][position]]
    adj_cells = adjacent_cells(sim_parameters, locations[0])
    for group in range((sim_parameters["population_size"] / \
                         sim_parameters["eco_cell_capacity"]) + 1):
        start = sim_parameters["eco_cell_capacity"] * group
        end = start + sim_parameters["eco_cell_capacity"]
        for x in range(start,end):
            if x == sim_parameters["population_size"]: break
            individual = Populations[pop_name].agents[x]
            if x > (sim_parameters["eco_cell_capacity"] - 1):
                location = random.choice(adj_cells)
                (x,y,z) = coordinates(location)
                while World.ecosystem[x][y][z]['organisms'] > \
                    sim_parameters["eco_cell_capacity"]:
                    location = random.choice(adj_cells)
                    (x,y,z) = coordinates(random.choice(adj_cells))
            (x,y,z) = coordinates(location)
            World.ecosystem[x][y][z]['organisms'] += 1
            individual.status['location'] = location

def interpret_chromosome(sim_parameters, Populations, pop_name, World):
    '''
    Function to call Ragaraja interpreter to express / execute the genome
    for each organism in a population. The Turing tape (array) after
    execution will be logged as "blood" of each organism. Ragaraja
    interpreter will use the temporary_input and temporary_output lists
    from each ecological cell as the input data and output data respectively
    for genome execution - this will resemble consumption of environmental
    resources and replenishing of environmental resources or dumping of
    wastes respectively.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param Populations: dictionary of population objects
    @param pop_name: population name
    @param World: dose_world.World object
    @return: none
    '''
    array = [0] * sim_parameters["max_tape_length"]
    for i in range(len(Populations[pop_name].agents)):
        individual = Populations[pop_name].agents[i]
        location = individual.status['location']
        (x,y,z) = coordinates(location)
        if sim_parameters["clean_cell"]:
            array = [0] * sim_parameters["max_tape_length"]
        else:
            array = Populations[pop_name].agents[i].status['blood']
            if array == None:
                array = [0] * sim_parameters["max_tape_length"]
        for chromosome_count in range(len(individual.genome)):
            inputdata = World.ecosystem[x][y][z]['local_input']
            output = World.ecosystem[x][y][z]['local_output']
            source = ''.join(individual.genome[chromosome_count].sequence)
            array = Populations[pop_name].agents[i].status['blood']
            try: (array, apointer, inputdata, output, source, spointer) = \
                register_machine.interpret(source, ragaraja.ragaraja, 3,
                                                 inputdata, array,
                                                 sim_parameters["max_tape_length"],
                                                 sim_parameters["max_codon"])
            except Exception as e:
                error_msg = '|'.join(['Error at Chromosome_' + \
                    str(chromosome_count), str(e)])
                Populations[pop_name].agents[i]. \
                    status['chromosome_error'] = error_msg
                Populations[pop_name].agents[i].status['blood'] = array
            Populations[pop_name].agents[i].status['blood'] = array
            World.ecosystem[x][y][z]['temporary_input'] = inputdata
            World.ecosystem[x][y][z]['temporary_output'] = output

def step(Populations, pop_name, sim_functions):
    '''
    Performs a generational step for a population
        - Prepopulation control
        - Mutations
        - Before mating fitness measurement
        - Mating
        - Postpopulation control
        - Generational events
        - After mating fitness measurement
        - Generate a textual report for the current generation

    @param Populations: dictionary of population objects
    @param pop_name: population name
    @param sim_functions: implemented simulation functions
    (see dose.dose_functions)
    @return: report as a string
    '''
    if Populations[pop_name].generation > 0:
        sim_functions.prepopulation_control(Populations, pop_name)
    for organism in Populations[pop_name].agents:
        sim_functions.mutation_scheme(organism)
    sim_functions.fitness(Populations, pop_name)
    sim_functions.mating(Populations, pop_name)
    sim_functions.postpopulation_control(Populations, pop_name)
    sim_functions.generation_events(Populations, pop_name)
    Populations[pop_name].generation = Populations[pop_name].generation + 1
    sim_functions.fitness(Populations, pop_name)
    return sim_functions.population_report(Populations, pop_name)

def report_generation(sim_parameters, Populations, pop_name,
                      sim_functions, generation_count):
    '''
    Performs a generational step (using step function) for a population
    and writes out the resulting report into results text file.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param Populations: dictionary of population objects
    @param pop_name: population name
    @param sim_functions: implemented simulation functions (see
    dose.dose_functions)
    @param generation_count: current generation count for reporting
    @return: none
    '''
    report = step(Populations, pop_name, sim_functions)
    if generation_count % int(sim_parameters["fossilized_frequency"]) == 0:
        file = '%s%s_%s_' % (sim_parameters["directory"],
                             sim_parameters["simulation_name"], pop_name)
        freeze_population(file, sim_parameters["fossilized_ratio"],
                          Populations, pop_name)
    if generation_count % int(sim_parameters["print_frequency"]) == 0:
        f = open(('%s%s_%s.result.txt' % (sim_parameters["directory"],
                                          sim_parameters["simulation_name"],
                                                        pop_name)), 'a')
        dtstamp = str(datetime.utcnow())
        f.write('\n'.join(['\n' + dtstamp, 'GENERATION: ' + \
                                   str(generation_count), str(report)]))
        f.write('\n')
        f.close

def bury_world(sim_parameters, World, generation_count):
    '''
    Function to bury entire world into a file.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param World: dose_world.World object
    @param generation_count: current generation count for file name generation
    '''
    if generation_count % int (sim_parameters["eco_buried_frequency"]) == 0:
       filename = '%s%s_gen%s.eco' % (sim_parameters["directory"],
                                      sim_parameters["simulation_name"],
                                      str(generation_count))
       f = open(filename, 'wb')
       pickle.dump(World, f)
       f.close()

def excavate_world(eco_file):
    '''
    Excavate buried world from file.

    @param eco_file: buried world file generated by bury_world function.
    @return: excavated dose_world.World object
    '''
    f = open(eco_file, 'rb')
    return pickle.load(f)

def freeze_population(file, proportion, Populations, pop_name):
    '''
    Function to freeze part or whole of the population into a file. If
    the number of organisms is less than 101, the entire population will
    be frozen.

    @param file: file name prefix
    @param proportion: proportion of the population to freeze
    @param Populations: dictionary of population objects
    @param pop_name: population name to freeze
    '''
    if proportion > 1.0: proportion = 1.0
    agents = Populations[pop_name].agents
    if len(agents) < 101 or len(agents) * proportion < 101:
        sample = deepcopy(Populations[pop_name])
    else:
        new_agents = [agents[random.randint(0, len(agents) - 1)]
                      for x in range(int(len(agents) * proportion))]
        sample = deepcopy(Populations[pop_name])
        sample.agents = new_agents
    name = ''.join([file, 'pop', str(Populations[pop_name].generation),
                    '_', str(len(sample.agents)), '.gap'])
    f = open(name, 'wb')
    pickle.dump(sample, f)
    f.close()

def revive_population(gap_file):
    '''
    Revive population from a frozen population file.

    @param gap_file: frozen population file generated by freeze_population
    function.
    @return: revived population object
    '''
    f = open(gap_file, 'rb')
    return pickle.load(f)

def write_parameters(sim_parameters, pop_name):
    '''
    Function to write simulation parameters into results text file as
    header.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param pop_name: population name
    @return: none
    '''
    f = open(('%s%s_%s.result.txt' % (sim_parameters["directory"],
                                       sim_parameters["simulation_name"],
                                      pop_name)), 'a')
    f.write("""SIMULATION: %s

-----------------------------------------------------------------------
""" % sim_parameters["simulation_name"])
    if ('database_source' in sim_parameters) or \
        ('sim_folder' in sim_parameters):
        f.write("SIMULATION REVIVAL STARTED: %s\n\n" % \
                sim_parameters["starting_time"])
    else:
        f.write("SIMULATION STARTED: %s\n\n" % \
                sim_parameters["starting_time"])
    for key in sim_parameters:
        if key not in ('deployment_scheme', 'directory', 'sim_folder'):
            f.write("%s : %s\n" % (key, sim_parameters[key]))
    f.write("""\n\nREPORT
-----------------------------------------------------------------------
""")

def close_results(sim_parameters, pop_name):
    '''
    Function to write footer of results text file.

    @param sim_parameters: simulation parameters dictionary (see Examples)
    @param pop_name: population name
    @return: none
    '''
    f = open(('%s%s_%s.result.txt' % (sim_parameters["directory"],
                                       sim_parameters["simulation_name"],
                                      pop_name)), 'a')
    f.write('''
-----------------------------------------------------------------------
SIMULATION ENDED: ''' + str(datetime.utcnow()))
    f.close()
```

File name: genetic.py

```
"""
Framework for Genetic Algorithm Applications.
Date created: 23rd February 2010

@see: Lim, JZR, Aw, ZQ, Goh, DJW, How, JA, Low, SXZ, Loo, BZL,
Ling, MHT. 2010. A genetic algorithm framework grounded in biology.
The Python Papers Source Codes 2: 6.
"""
import random, os, string
from copy import deepcopy

class Chromosome(object):
    """
    Representation of a linear chromosome.
    @see: Lim, JZR, Aw, ZQ, Goh, DJW, How, JA, Low, SXZ, Loo, BZL,
    Ling, MHT. 2010. A genetic algorithm framework grounded in biology.
    The Python Papers Source Codes 2: 6.

    @since: version 0.4
    """
    def __init__(self, sequence, base,
                 background_mutation=0.0001):
        """
        Sets up a chromosome.

        @param sequence: a subscriptable object (list or string) representing
            the sequence of the chromosome.
        @param base: a subscriptable object (list or string) representing
            allowable entities in the sequence.
        @param background_mutation: background mutation rate represented as the
            probability of number of mutations per base. Default = 0.0001
            (0.01%).

        @since: version 0.4
        """
        self.sequence = sequence
        self.base = base
        self.background_mutation = background_mutation

    def rmutate(self, type='point', rate=0.01, start=0, end=-1):
        """
        Random Mutation operator - to simulate random point, insertion,
        deletion, inversion, gene translocation and gene duplication events.

        The start and end parameters are useful for simulating mutational
        hotspots in the genome.

        @param type: type of mutation. Accepts 'point' (point mutation),
            'insert' (insert a base), 'delete' (delete a base), 'invert'
            (invert a stretch of the chromosome), 'duplicate' (duplicate a
            stretch of the chromosome), 'translocate' (translocate a
            stretch of chromosome to another random position).
            Default = point.
        @param rate: probability of mutation per base above background
            mutation rate. Default = 0.01 (1%). No mutation event will ever
            happen if (rate + background_mutation) is less than zero.
        @param start: starting base on the sequence for mutation.
            Default = 0, start of the genome.
        @param end: last base on the sequence for mutation. Default = -1,
            end of the genome.

        @since: version 0.4
        """
        if end > len(self.sequence) - 1: end = len(self.sequence) - 1
        if end == -1: end = len(self.sequence) - 1
        if start == end: start = 0
        length = int(end - start)
        mutation = int((self.background_mutation + rate) * length)
        while mutation > 0:
            position = int(start) + random.randrange(length - 1)
            new_base = self.base[random.randrange(len(self.base))]
            if type == 'point':
                self.sequence[position] = new_base
            if type == 'delete':
                self.sequence.pop(position)
            if type == 'insert':
                self.sequence.insert(position, new_base)
            if type == 'duplicate':
                end_pos = random.randrange(position + 1, end)
                fragment = self.sequence[position:end_pos]
                for i in range(len(fragment)):
                    self.sequence.insert(end_pos + i, fragment[i])
            if type == 'invert':
                end_pos = random.randrange(position + 1, end)
                fragment = [self.sequence.pop(position)
                            for i in range(end_pos - position)]
                fragment.reverse()
                for base in fragment:
                    self.sequence.insert(position, base)
            if type == 'translocate':
                end_pos = random.randrange(position + 1, end)
                fragment = [self.sequence.pop(position)
                            for i in range(end_pos - position)]
                insertion_point = random.randint(0, len(self.sequence))
                for i in range(len(fragment)):
                    self.sequence.insert(insertion_point + i, fragment[i])
            mutation = mutation - 1

    def kmutate(self, type='point', start=0, end=0, sequence=None, tpos=0):
        """
        Known Mutation operator - to simulate a known point, insertion,
        deletion, inversion, gene translocation or gene duplication event.

        The required parameters will be determined by the type of mutation:
            - point requires
                - start (the position of the base to mutate)
                - sequence (the new base)
            - delete requires
                - start (the position of the base to delete)
            - insert requires
                - start (the position of the base to begin insertion)
                - sequence (list of sequence or a base to insert)
            - invert requires
                - start (the position of the base to start inversion)
                - end (the position of the base to end inversion)
            - duplicate requires
                - start (the position of the starting base to duplicate)
                - end (the position of the last base to duplicate)
            - translocate requires
                - start (the position of the base to start translocation)
                - end (the position of the base to end translocation)
                - tpos (the position of the base insert the translocated
                    sequence)

        @param type: type of mutation. Accepts 'point' (point mutation),
            'insert' (insert a base), 'delete' (delete a base), 'invert'
            (invert a stretch of the chromosome), 'duplicate' (duplicate a
            stretch of the chromosome), 'translocate' (translocate a stretch
            of chromosome to another random position). Default = point.
        @param start: starting base on the chromosome for mutation.
            Default = 0, start of the genome.
        @param end: last base on the chromosome for mutation. Default = 0.
        @param sequence: list of bases to change to (point mutation) or to
            insert (insertion mutation)
        @param tpos: position of the chromosome to insert the translocated
            sequence.

        @since: version 0.4
        """
        if type == 'point':
            self.sequence[start] = sequence
        if type == 'delete':
            self.sequence.pop(start)
        if type == 'insert':
            for base in list(sequence):
                self.sequence.insert(start, base)
        if type == 'duplicate':
            fragment = self.sequence[start:end + 1]
            for i in range(len(fragment)):
                self.sequence.insert(end + i, fragment[i])
        if type == 'invert':
            for base in self.sequence[start:end]:
                self.sequence.insert(start, base)
        if type == 'translocate':
            fragment = [self.sequence.pop(start)
                        for i in range(end - start)]
            for i in range(len(fragment)):
                self.sequence.insert(tpos + i, fragment[i])

    def replicate(self):
        """
        Replicates (deep copy) the chromosome.

        @return: a copy of current chromosome.

        @since: version 0.4
        """
        return deepcopy(self)


class Organism(object):
    """
    An organism represented by a list of chromosomes and a status table.
    This class should almost never be instantiated on its own but acts as
    an ancestor class as some functions need to be over-ridden in the
    inherited class for the desired use.

    Each organism is identifiable by a randomly generated 32-character
    'name' as Organism.identity.

    Methods to be over-ridden in the inherited class or substituted are
        - fitness
        - mutation_scheme

    Pre-defined status are
        1. alive - is the organism alive? True or False.
        2. vitality - percentage of maximum vitality.
        3. age - the current age of the organism.
        4. lifespan - pre-defined maximum lifespan to be set based on scenario.
        5. fitness - how fit the organism is? Set to maximum fitness.
        6. death - reason of death (as death code).
        7. parents - identity of parents.
        8. gender - gender of organism.
        9. blood - result of genomic interpretation or expression by
        Rajaraga interpreter.
        10. identity - 32-character randomly generated name.
        11. deme - defined as a sub-population or local population.
        12. location - location of the organism within the ecosystem.


    List of defined death codes
        1. death01 - zero vitality
        2. death02 - maximum age reached
        3. death03 - unknown death cause

    @see: Lim, JZR, Aw, ZQ, Goh, DJW, How, JA, Low, SXZ, Loo, BZL,
    Ling, MHT. 2010. A genetic algorithm framework grounded in biology.
    The Python Papers Source Codes 2: 6.

    @since: version 0.4
    """
    def __init__(self, genome='dummy', mutation_type='point',
                 additional_mutation_rate=0.01, gender=None):
        """
        Sets up a new organism with default status (age = 0, vitality = 100,
        lifespan = 100, fitness = 100, alive = True)

        @param genome: list of chromosomes to inherit. Default = 'dummy'.
            This creates a 'dummy' chromosome which is basically a one-base
            chromosome - this is for applications which does not utilize
            the chromosome.
        @param mutation_type: type of mutation. Accepts 'point' (point
            mutation), 'insert' (insert a base), 'delete' (delete a base),
            'invert' (invert a stretch of the chromosome), 'duplicate'
            (duplicate a stretch of the chromosome), 'translocate'
            (translocate a stretch of chromosome to another random position).
            Default = point.
        @param additional_mutation_rate: probability of mutation per base
            above background mutation rate. Default = 0.01 (1%). No mutation
            event will ever happen if (rate + background_mutation) is less
            than zero.
        @param gender: establishes the gender of the organism which may be
            used for mating routines.

        @since: version 0.4
        """
        self.status = {'alive': True,           # is the organism alive?
                       'vitality': 100.0,       # % of vitality
                       'parents': None,         # identity of parent(s)
                       'age': 0.0,              # age of the organism
                       'gender': None,          # gender of organism
                       'lifespan': 100.0,       # maximum lifespan
                       'fitness': 100.0,        # % of fitness
                       'blood': None,           # interpreted chromosome
                       'identity': None,        # name of the organism
                       'deme': None,            # sub-population or race
                       'location': None,        # location of the organism
                       'death': None}
        if genome == 'dummy':
            self.genome = [Chromosome([0], [0])]
        else:
            self.genome = genome
        self.mutation_type = mutation_type
        self.additional_mutation_rate = additional_mutation_rate
        self.status['gender'] = gender

    def generate_name(self):
        name = ''.join([random.choice(('1', '2', '3', '4', '5', '6', '7',
                                       '8', '9', 'A', 'B', 'C', 'D', 'E',
                                       'F', 'G', 'H', 'I', 'J', 'K', 'L',
                                       'M', 'N', 'O', 'P', 'Q', 'R', 'S',
                                       'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
                                       'a', 'b', 'c', 'd', 'e', 'f', 'g',
                                       'h', 'i', 'j', 'k', 'l', 'm', 'n',
                                       'o', 'p', 'q', 'r', 's', 't', 'u',
                                       'v', 'w', 'x', 'y', 'z', '=', '#',
                                       '$', '%', '&', '@', '<', '>', '?'
                                       )) for x in range(32)])

        self.status['identity'] = name

    def fitness(self):
        """
        Function to calculate the fitness of the current organism.
        B{This function MUST be over-ridden by the inherited class or
        substituted as fitness function is highly dependent on utility.}
        The only requirement is that a fitness score must be returned.

        Here, the sample implementation calculates fitness as proportion of
        the genome with '1's.

        @return: fitness score or fitness list

        @since: version 0.4
        """
        one_count = sum([sum(chromosome.sequence)
                         for chromosome in self.genome])
        length_of_genome = sum([len(chromosome.sequence)
                                for chromosome in self.genome])
        return float(one_count) / float(length_of_genome)

    def mutation_scheme(self, type=None, rate=None):
        """
        Function to trigger mutation events in each chromosome. B{This
        function may be over-ridden by the inherited class or substituted
        to cater for specific mutation schemes but not an absolute
        requirement to do so.}

        Both type and rate must be defined at the same time, otherwise the
        initiated mutation_type and additional_mutation_rate will be used.

        @param type: type of mutation. Accepts 'point' (point mutation),
            'insert' (insert a base), 'delete' (delete a base), 'invert'
            (invert a stretch of the chromosome), 'duplicate' (duplicate a
            stretch of the chromosome), 'translocate' (translocate a stretch
            of chromosome to another random position). Default = None.
        @param rate: probability of mutation per base above background mutation
            rate. Default = None. No mutation event will ever happen if
            (rate + background_mutation) is less than zero.

        @since: version 0.4
        """
        for chromosome in self.genome:
            if not type and not rate:
                chromosome.rmutate(self.mutation_type,
                                   self.additional_mutation_rate)
            if type and rate:
                chromosome.rmutate(type, rate)

    def setStatus(self, variable, value):
        """
        Sets new status or change status of the organism. However, the
        following status change will result in death of the organism
            1. 'alive' to False
            2. 'vitality' to or below zero
            3. 'age' to or above lifespan

        @param variable: name of status to change
        @param value: new value of the status

        @since: version 0.4
        """
        if variable == 'alive' and value == True:
            self.status['alive'] = value
        if variable == 'alive' and value == False:
            self.status['alive'] = value
            self.status['death'] = 'death03'
        if variable == 'vitality':
            if float(value) > 100.1:
                self.status['vitality'] = 100.0
            if float(value) > 0.0 and float(value) < 100.1:
                self.status['vitality'] = float(value)
            if float(value) == 0 or float(value) < 0:
                self.status['vitality'] = 0
                self.status['alive'] = False
                self.status['death'] = 'death01'
        elif variable == 'age':
            if float(value) < self.status['lifespan']:
                self.status['age'] = float(value)
            else:
                self.status['age'] = self.status['lifespan']
                self.status['alive'] = False
                self.status['death'] = 'death02'
        else:
            self.status[variable] = value

    def getStatus(self, variable):
        """
        Returns a status variable of the current organism

        @param variable: name of the status variable
        @return: status or a KeyError if status is not found

        @since: version 0.4
        """
        try:
            return self.status[variable]
        except:
            raise KeyError('%s not found in organism status' % str(variable))

    def __str__(self):
        """Returns the genome of the organism"""
        return str([chromosome.sequence for chromosome in self.genome])

    def clone(self):
        """
        Cloness (deep copy) the organism.

        @return: a copy of current organism.

        @since: version 0.4
        """
        org = deepcopy(self)
        return org

class Population(object):
    """
    Representation of a population as a list of organisms. The entire
    population is in memory.

    Methods to be over-ridden in the inherited class or substituted are
        - prepopulation_control
        - mating
        - postpopulation_control
        - generation_events
        - report

    @see: Lim, JZR, Aw, ZQ, Goh, DJW, How, JA, Low, SXZ, Loo, BZL,
    Ling, MHT. 2010. A genetic algorithm framework grounded in biology.
    The Python Papers Source Codes 2: 6.

    @since: version 0.4
    """

    def __init__(self, goal, maxgenerations='infinite', agents=[]):
        """
        Establishes a population of organisms.
        @param goal: the goal to be reached by the population.
        @type goal: the return type of Organism.fitness()
        @param maxgenerations: maximum number of generations to evolve.
            Default = 'infinite'.
        @param agents: organisms making up the initial population.
        @type agents: list of Organism objects

        @since: version 0.4
        """
        self.agents = agents
        self.goal = goal
        self.maxgenerations = maxgenerations
        self.generation = 0

    def prepopulation_control(self):
        """
        Function to trigger population control events before mating event in
        each generation (For example, to simulate pre-puberty death). B{This
        function may be over-ridden by the inherited class or substituted to
        cater for specific events.} Although this is not an absolute
        requirement, it is extremely encouraged to prevent exhaustion of
        memory space. Without population control, it will seems like a
        reproducing immortal population.

        Here, the sample implementation eliminates the bottom half of the
        population based on fitness score unless the number of organisms is
        less than 20. This is to prevent extinction. However, if the number
        of organisms is more than 2000 (more than 100x initial population
        size, a random selection of 2000 will be used for the next generation.

        @since: version 0.4
        """
        size = len(self.agents)
        sfitness = [self.agents[x].fitness() for x in range(size)]
        threshold = sum(sfitness) / len(sfitness)
        temp = [self.agents[x]
                for x in range(size)
                    if sfitness[x] > threshold]
        if len(temp) > 2001:
            size = len(temp)
            self.agents = [temp[random.randint(0, size - 1)]
                           for x in range(2000)]
        if len(temp) > 21:
            self.agents = temp

    def mating(self):
        """
        Function to trigger mating events in each generation. B{This function
        may be over-ridden by the inherited class or substituted to cater for
        specific mating schemes but not an absolute requirement to do so.}
        Mating schemes should include
            - selection of mating partners using Organism.fitness() function
                and/or other status
            - processes and actions of mating

        Here, the sample implementation randomly selects any 2 organisms from
        the culled list and generate a new genome for the progeny organism by
        single crossover.

        @since: version 0.4
        """
        size = len(self.agents)
        temp = []
        for x in range(size):
            organism1 = self.agents[random.randint(0, size - 1)]
            organism2 = self.agents[random.randint(0, size - 1)]
            crossover_pt = random.randint(0, len(organism1.genome[0].sequence))
            (g1, g2) = crossover(organism1.genome[0], organism2.genome[0],
                                 crossover_pt)
            temp = temp + [Organism([g1])]
        self.add_organism(temp)

    def postpopulation_control(self):
        """
        Function to trigger population control events after mating event in
        each generation(For example, to simulate old-age death). B{This
        function may be over-ridden by the inherited class or substituted to
        cater for specific events.} Although this is not an absolute
        requirement, it is extremely encouraged to prevent exhaustion of
        memory space. Without population control, it will seems like a
        reproducing immortal population.

        @since: version 0.4
        """
        pass

    def generation_events(self):
        """
        Function to trigger other defined events in each generation. B{This
        function may be over-ridden by the inherited class or substituted to
        cater for specific events but not an absolute requirement to do so.}
        Events and controls may include
            - processes simulating disaster or other catastrophic events
            - changes in mutations

        @since: version 0.4
        """
        pass

    def report(self):
        """
        Function to report the status of each generation. B{This function may
        be over-ridden by the inherited class or substituted to cater for
        specific reporting schemes but not an absolute requirement to do so.}
        At the very least, this function should report whether the goal is
        reached.

        @return: dictionary of status describing the current generation

        @since: version 0.4
        """
        sfitness = [self.agents[x].fitness()
                    for x in range(len(self.agents))]
        afitness = sum(sfitness) / float(len(self.agents))
        return {'generation': self.generation,
                'average fitness': afitness,
                '% to goal': float(afitness - self.goal) / self.goal * 100}

    def generation_step(self):
        """
        Function to simulate events for one generation. These includes
            - mating (according to the mating scheme or function)
            - mutating each organism in the population
            - population size control
            - other events defined under generation_events function
            - increment of generation count
            - reporting the population status

        @return: information returned from report function.

        @since: version 0.4
        """
        if self.generation > 0:
            self.prepopulation_control()
        self.mating()
        self.postpopulation_control()
        for organism in self.agents:
            organism.mutation_scheme()
        self.generation_events()
        self.generation = self.generation + 1
        return self.report()

    def add_organism(self, organism):
        """Add a new organism(s) to the population.

        @param organism: list of new Organism object(s)

        @since: version 0.4"""
        self.agents = self.agents + organism

    def freeze(self, prefix='pop', proportion=0.01):
        """
        Preserves part or the entire population. If the population size or
        the preserved proportion is below 100, the entire population will be
        preserved. The preserved sample will be written into a file with name
        in the following format - <prefix><generation count>_<sample size>.gap

        @param prefix: prefix of file name. Default = 'pop'.
        @param proportion: proportion of population to be preserved.
            Default = 0.01, preserves 1% of the population.

        @since: version 0.4
        """
        # In Python 3, cPickle is no longer needed: Py3 looks for
        # an optimized version, and if it founds none, will load the
        # pure python implementation of pickle.
        try:
            import cPickle as pickle
        except ImportError:
            import pickle

        if proportion > 1.0: proportion = 1.0
        if len(self.agents) < 101 or len(self.agents) * proportion < 101:
            sample = self.agents
        else:
            size = len(self.agents)
            sample = [self.agents[random.randint(0, size - 1)]
                      for x in range(int(len(self.agents) * proportion))]
        name = ''.join([prefix, str(self.generation), '_',
                        str(len(sample)), '.gap'])
        f = open(name, 'wb')
        pickle.dump(sample, f)
        f.close()

    def revive(self, filename, type='replace'):
        """
        Revives a frozen population.

        @param filename: file name of frozen population (generated by
            Population.freeze() function)
        @param type: type of revival. Allows 'replace' (replace the current
            population with the revived population) or 'add' (add the revived
            population to the current population). Default = 'replace'.

        @since: version 0.4
        """
        # In Python 3, cPickle is no longer needed: Py3 looks for
        # an optimized version, and if it founds none, will load the
        # pure python implementation of pickle.
        try:
            import cPickle as pickle
        except ImportError:
            import pickle
        if type == 'replace':
            self.agents = pickle.load(open(filename, 'rb'))
        if type == 'add':
            self.agents = self.agents + pickle.load(open(filename, 'rb'))

def crossover(chromosome1, chromosome2, position):
    """
    Cross-over operator - swaps the data on the 2 given chromosomes after
    a given position.

    @param chromosome1: Chromosome object
    @param chromosome2: Chromosome object
    @param position: base position of the swap over
    @type position: integer
    @return: (resulting chromosome1, resulting chromosome2)

    New chromosomes inherit their parent's crossed sequences, bases and
    background mutation rate.

    @since: version 0.4
    """
    seq1 = chromosome1.sequence
    seq2 = chromosome2.sequence
    position = int(position)
    if len(seq1) > position and len(seq2) > position:
        new1 = Chromosome(seq1[:position] + seq2[position:],
                          chromosome1.base, chromosome1.background_mutation)
        new2 = Chromosome(seq2[:position] + seq1[position:],
                          chromosome2.base, chromosome2.background_mutation)
        return (new1, new2)
    elif len(seq1) > position:
        new1 = Chromosome(seq1[:position], chromosome1.base,
                          chromosome1.background_mutation)
        new2 = Chromosome(chromosome2.sequence + seq1[position:],
                          chromosome2.base, chromosome2.background_mutation)
        return (new1, new2)
    elif len(seq2) > position:
        new1 = Chromosome(chromosome1.sequence + seq2[position:],
                          chromosome1.base, chromosome1.background_mutation)
        new2= Chromosome(seq2[:position], chromosome2.base,
                         chromosome2.background_mutation)
        return (new1, new2)
    else:
        return (chromosome1, chromosome2)

population_data = \
{
    'chromosome_bases' : [1, 2, 3, 4],
    'chromosome_length' : 200,
    'chromosome_type' : 'defined',
    'initial_chromosome' : [1] * 200,
    'background_mutation' : 0.0001,
    'genome_size' : 1,
    'population_size' : 200,
    'fitness_function' : 'default',
    'mutation_scheme' : 'default',
    'additional_mutation' : 0.01,
    'mutation_type' : 'point',
    'goal' : 4,
    'maximum_generations' : 'infinite',
    'prepopulation_control' : 'default',
    'mating' : 'default',
    'postpopulation_control' : 'default',
    'generation_events' : 'default',
    'population_report' : 'default'
}

def population_constructor(data=population_data):
    """
    Function to construct a population based on a dictionary of population
    data.

    Population data contains the following keys:
        - 'chromosome_bases' = List of allowable bases.
            Default = [1, 2, 3, 4].
        - 'chromosome_length' = Length of a chromosome. Default = 200.
        - 'chromosome_type' = Type of chromosome. Default = 'defined'.
        - 'initial_chromosome' = Initial chromosome. Default = [1] * 200.
        - 'background_mutation' = Background mutation rate.
            Default = 0.0001 (0.01%).
        - 'genome_size' = Number of chromosomes per organism. Default = 1.
        - 'population_size' = Size of initial population (number of organisms).
            Default = 200.
        - 'fitness_function' = Fitness evaluation function. Accepts 'default'
            or a function. Please refer to Organism. Default = 'default'.
        - 'mutation_scheme' = Function simulating the mutation scheme of an
            organism. Accepts 'default' or a function. Please refer to
            Organism. Default = 'default'.
        - 'additional_mutation' = Mutation rate on top of background
            mutation rate. Default = 0.01 (1%).
        - 'mutation_type' = Type of default mutation. Default = 'point'.
        - 'goal' = Goal of the population, as evaluated by fitness function.
            Default = 4.
        - 'maximum_generations' = Number of generations to simulate. Accepts
            an integer or 'infinite'. Default = 'infinite'.
        - 'prepopulation_control' = Function simulating pre-mating population
            control. Please refer to Population. Default = 'default'.
        - 'mating' = Function simulating mating procedure (mate selection
            and act of mating control. Please refer to Population.
            Default = 'default'.
        - 'postpopulation_control' = Function simulating post-mating population
            control. Please refer to Population. Default = 'default'.
        - 'generation_events' = Function simulating other possible (usually
            rare or random) events in the generation. Please refer to
            Population. Default = 'default'.
        - 'population_report' = Function to generate the status report of
            the generation. Please refer to Population. Default = 'default'.

    @param data: population data
    @type data: dictionary

    @return: Population object

    @see: Lim, JZR, Aw, ZQ, Goh, DJW, How, JA, Low, SXZ, Loo, BZL,
    Ling, MHT. 2010. A genetic algorithm framework grounded in biology.
    The Python Papers Source Codes 2: 6.

    @since: version 0.4
    """
    chr = Chromosome(data['initial_chromosome'],
                     data['chromosome_bases'],
                     data['background_mutation'])
    org = Organism([chr]*data['genome_size'],
                   data['mutation_type'],
                   data['additional_mutation'])
    org_set = [org.clone() for x in range(data['population_size'])]
    pop = Population(data['goal'],
                         int(data['maximum_generations']),
                         org_set)
    return pop

def population_simulate(population,
                            printfreq=100,
                            freezefreq='never',
                            freezefile='pop',
                            freezeproportion=0.01,
                            resultfile='result.txt'):
    """
    Function to simulate the population - start the GA.

    @param population: Population to run.
    @type population: Population object
    @param printfreq: Reporting intervals on screen. Default = 100.
    @param freezefreq: Generation intervals to freeze population into a file.
        See Population.freeze method. Accepts an integer or 'never'.
        Default = 'never'.
    @param freezefile: Prefix of file name of frozen population.
        See Population.freeze method. Default = 'pop'.
    @param freezeproportion: Proportion of population to be preserved.
        See Population.freeze method. Default = 0.01, preserves 1% of the
        population.
    @param resultfile: Name of file to print out results of each generation.
        Format of output is dependent on reporting method of the population
        (Population.report). If 'None', it will print out the results into
        a file. Default = 'result.txt'.

    @see: Lim, JZR, Aw, ZQ, Goh, DJW, How, JA, Low, SXZ, Loo, BZL,
    Ling, MHT. 2010. A genetic algorithm framework grounded in biology.
    The Python Papers Source Codes 2: 6.

    @since: version 0.4
    """
    if freezefreq == 'never': freezefreq = int(12e14)
    if resultfile != 'None': result = open(resultfile, 'w')
    report = population.generation_step()
    reportitems = ['|'.join([str(key), str(report[key])])
                    for key in list(report.keys())]
    result.writelines('|'.join(reportitems))
    result.writelines(os.linesep)
    while population.generation < population.maxgenerations:
        if resultfile != 'None':
            report = population.generation_step()
            reportitems = ['|'.join([str(key), str(report[key])])
                            for key in list(report.keys())]
            result.writelines('|'.join(reportitems))
        if population.generation % int(printfreq) == 0:
            print('|'.join(reportitems))
        result.writelines(os.linesep)
        if population.generation % int(freezefreq) == 0:
            population.freeze(freezefile, freezeproportion)
    population.freeze(freezefile, 1.0)
    result.close()
```

Examples of Simulation File from Castillo and Ling (2014a)

File name: run_examples_without_installation.py

```
'''
This file contains inserts the path for DOSE into system path to
allow for any examples importing this file to execute without
needing DOSE to be installed into into Python site-packages.
```

```
'''
import sys
import os

cwd = os.getcwd().split(os.sep)
#cwd[-1] = 'dose'
cwd = os.sep.join(cwd[:-1])

sys.path.append(cwd)

import dose
```

File name: 09_no_migration_isolated_mating.py

```
'''
Example 09a: Examining the effects of no migration on genetic distance
differences from an initially identical population (development of sub-
populations or demes which may lead to speciation)

This example is identical to Example 03, except background mutation rate is
changed from 10% in Example 03 to 0.1% in this example.

In this simulation,
    - 1 populations of 1250 organisms
    - each organism will have 1 chromosome of only 2 bases (1 and 0)
    - Evenly deployed across 25 eco-cells (50 organism per eco-cell)
    - 0.1% background point mutation on chromosome of 50 bases
    - no organism movement throughout the simulation
    - no Ragaraja interpretation of genome
    - 1000 generations to be simulated
'''
# needed to run this example without prior
# installation of DOSE into Python site-packages
try:
    import run_examples_without_installation
except ImportError: pass

# Example codes starts from here
import dose, random

parameters = {
              "simulation_name": "09_no_migration_isolated_mating",
              "population_names": ['pop_01'],
              "population_locations": [[(x,y,z)
                                          for x in range(5)
                                              for y in range(5)
                                                  for z in range(1)]],
              "deployment_code": 3,
              "chromosome_bases": ['0','1'],
              "background_mutation": 0.001,
              "additional_mutation": 0,
              "mutation_type": 'point',
              "chromosome_size": 5000,
              "genome_size": 1,
              "max_tape_length": 50,
              "clean_cell": True,
              "interpret_chromosome": False,
              "max_codon": 2000,
              "population_size": 1250,
              "eco_cell_capacity": 50,
              "world_x": 5,
              "world_y": 5,
              "world_z": 1,
              "goal": 0,
              "maximum_generations": 1000,
              "fossilized_ratio": 0.01,
              "fossilized_frequency": 100,
              "print_frequency": 10,
              "ragaraja_version": 0,
              "ragaraja_instructions": ['000', '001', '010',
                                        '011', '100', '101'],
              "eco_buried_frequency": 1250,
              "database_file": "sim09_no_migration.db",
              "database_logging_frequency": 1
             }

class simulation_functions(dose.dose_functions):

    def organism_movement(self, Populations, pop_name, World): pass

    def organism_location(self, Populations, pop_name, World): pass

    def ecoregulate(self, World): pass

    def update_ecology(self, World, x, y, z): pass

    def update_local(self, World, x, y, z): pass

    def report(self, World): pass

    def fitness(self, Populations, pop_name): pass

    def mutation_scheme(self, organism):
        organism.genome[0].rmutate(parameters["mutation_type"],
                                   parameters["additional_mutation"])

    def prepopulation_control(self, Populations, pop_name): pass

    def mating(self, Populations, pop_name):
        for location in parameters["population_locations"][0]:
            group = dose.filter_location(location, Populations[pop_name].agents)
            for x in range(len(group)//2):
                parents = []
                for i in range(2):
                    parents.append(random.choice(Populations[pop_name].agents))
                    while parents[i] not in group:
                        parents[i] = random.choice(Populations[pop_name].agents)
                    Populations[pop_name].agents.remove(parents[i])
                crossover_pt = random.randint(0,
                               len(parents[0].genome[0].sequence))
                (new_chromo1, new_chromo2) = \
                    dose.genetic.crossover(parents[0].genome[0],
                    parents[1].genome[0], crossover_pt)
                children = [dose.genetic.Organism([new_chromo1],
                                              parameters["mutation_type"],
                                              parameters["additional_mutation"]),
                            dose.genetic.Organism([new_chromo2],
                                              parameters["mutation_type"],
                                              parameters["additional_mutation"])]
                for child in children:
                    child.status['parents'] = [parents[0].status['identity'],
                                               parents[1].status['identity']]
                    child.status['location'] = location
                    child.generate_name()
                    child.status['deme'] = pop_name
                    Populations[pop_name].agents.append(child)

    def postpopulation_control(self, Populations, pop_name): pass

    def generation_events(self, Populations, pop_name): pass
    def population_report(self, Populations, pop_name):
        report_list = []
        for organism in Populations[pop_name].agents:
            identity = str(organism.status['identity'])
            report_list.append(identity)
        return '\n'.join(report_list)

    def database_report(self, con, cur, start_time,
                        Populations, World, generation_count):
        try:
            dose.database_report_populations(con, cur, start_time,
                                             Populations, generation_count)
        except: pass
        try:
            dose.database_report_world(con, cur, start_time,
                                       World, generation_count)
        except: pass

    def deployment_scheme(self, Populations, pop_name, World): pass

dose.simulate(parameters, simulation_functions)
```

File name: 09_adjacent_migration_isolated_mating.py

```
'''
Example 09b: Extending from Example 09a (as baseline) to examine the effects
of short migration (adjacent cell migration)  on genetic distance
differences from an initially identical population (development of sub-
populations or demes which may lead to speciation)

This example is identical to Example 04, except background mutation rate is
changed from 10% in Example 03 to 0.1% in this example.

In this simulation,
    - 1 populations of 1250 organisms
    - each organism will have 1 chromosome of only 2 bases (1 and 0)
    - Evenly deployed across 25 eco-cells (50 organism per eco-cell)
    - 0.1% background point mutation on chromosome of 50 bases
    - 10% organism movement per eco-cell per generation throughout the
    simulation
    - the same organism may move twice due to sequential evaluation of the
    eco-cells but the probability of such event will be 10% x 10% = 1%;
    similarly, 3 or more movement by the same organism may happen with
    reducing probabilities
    - no Ragaraja interpretation of genome
    - 1000 generations to be simulated
'''
# needed to run this example without prior
# installation of DOSE into Python site-packages
try:
    import run_examples_without_installation
except ImportError: pass

# Example codes starts from here
import dose, random

parameters = {
              "simulation_name": "09_adjacent_migration_isolated_mating",
              "population_names": ['pop_01'],
              "population_locations": [[(x,y,z)
                                        for x in range(5)
                                            for y in range(5)
                                                for z in range(1)]],
              "deployment_code": 3,
              "chromosome_bases": ['0','1'],
              "background_mutation": 0.001,
              "additional_mutation": 0,
              "mutation_type": 'point',
              "chromosome_size": 5000,
              "genome_size": 1,
              "max_tape_length": 50,
              "clean_cell": True,
              "interpret_chromosome": False,
              "max_codon": 2000,
              "population_size": 1250,
              "eco_cell_capacity": 50,
              "world_x": 5,
              "world_y": 5,
              "world_z": 1,
              "goal": 0,
              "maximum_generations": 1000,
              "fossilized_ratio": 0.01,
              "fossilized_frequency": 100,
              "print_frequency": 10,
              "ragaraja_version": 0,
              "ragaraja_instructions": ['000', '001', '010',
                                        '011', '100', '101'],
              "eco_buried_frequency": 1250,
              "database_file": "sim09_adjacent.db",
              "database_logging_frequency": 1
             }

class simulation_functions(dose.dose_functions):

    def organism_movement(self, Populations, pop_name, World):
        for location in parameters["population_locations"][0]:
            group = dose.filter_location(location, Populations[pop_name].agents)
            adj_cells = dose.simulation_calls.adjacent_cells(parameters, location)
            for i in range(int(round((len(group) * 0.1)))):
                (x,y,z) = dose.simulation_calls.coordinates(location)
                World.ecosystem[x][y][z]['organisms'] -= 1
                immigrant = random.choice(Populations[pop_name].agents)
                while immigrant not in group:
                    immigrant = random.choice(Populations[pop_name].agents)
                new_location = random.choice(adj_cells)
                immigrant.status['location'] = new_location
                (x,y,z) = dose.simulation_calls.coordinates(new_location)
                World.ecosystem[x][y][z]['organisms'] += 1

    def organism_location(self, Populations, pop_name, World): pass

    def ecoregulate(self, World): pass

    def update_ecology(self, World, x, y, z): pass

    def update_local(self, World, x, y, z): pass

    def report(self, World): pass

    def fitness(self, Populations, pop_name): pass

    def mutation_scheme(self, organism):
        organism.genome[0].rmutate(parameters["mutation_type"],
                                   parameters["additional_mutation"])

    def prepopulation_control(self, Populations, pop_name): pass

    def mating(self, Populations, pop_name):
        for location in parameters["population_locations"][0]:
            group = dose.filter_location(location, Populations[pop_name].agents)
            for x in range(len(group)//2):
                parents = []
                for i in range(2):
                    parents.append(random.choice(Populations[pop_name].agents))
                    while parents[i] not in group:
                        parents[i] = random.choice(Populations[pop_name].agents)
                    Populations[pop_name].agents.remove(parents[i])
                crossover_pt = random.randint(0,
                           len(parents[0].genome[0].sequence))
                (new_chromo1, new_chromo2) = \
                    dose.genetic.crossover(parents[0].genome[0],
                    parents[1].genome[0], crossover_pt)
                children = [dose.genetic.Organism([new_chromo1],
                                              parameters["mutation_type"],
                                              parameters["additional_mutation"]),
                        dose.genetic.Organism([new_chromo2],
                                              parameters["mutation_type"],
                                              parameters["additional_mutation"])]
                for child in children:
                    child.status['parents'] = [parents[0].status['identity'],
                                               parents[1].status['identity']]
                    child.status['location'] = location
                    child.generate_name()
                    child.status['deme'] = pop_name
                    Populations[pop_name].agents.append(child)

    def postpopulation_control(self, Populations, pop_name): pass

    def generation_events(self, Populations, pop_name): pass

    def population_report(self, Populations, pop_name):
        report_list = []
        for organism in Populations[pop_name].agents:
            identity = str(organism.status['identity'])
            report_list.append(identity)
        return '\n'.join(report_list)

    def database_report(self, con, cur, start_time,
                        Populations, World, generation_count):
        try:
            dose.database_report_populations(con, cur, start_time,
                                             Populations, generation_count)
        except: pass
        try:
            dose.database_report_world(con, cur, start_time,
                                       World, generation_count)
        except: pass

    def deployment_scheme(self, Populations, pop_name, World): pass

dose.simulate(parameters, simulation_functions)
```

File name: 09_long_migration_isolated_mating.py

```
'''
Example 09c: Extending from Example 09a (as baseline) and Example 09b
(short distance migration) to examine the effects of long distance
migration (across one or more eco-cells)  on genetic distance differences
from an initially identical population (development of sub-populations or
demes which may lead to speciation)

This example is identical to Example 05, except background mutation rate is
changed from 10% in Example 03 to 0.1% in this example.

In this simulation,
    - 1 populations of 1250 organisms
    - each organism will have 1 chromosome of only 2 bases (1 and 0)
    - Evenly deployed across 25 eco-cells (50 organism per eco-cell)
    - 20% background point mutation on chromosome of 50 bases
    - 10% organism movement per eco-cell per generation throughout the
    simulation
    - the same organism may move twice due to sequential evaluation of the
    eco-cells but the probability of such event will be 10% x 10% = 1%;
    similarly, 3 or more movement by the same organism may happen with
    reducing probabilities
    - the destination eco-cell for movement is random; thus, from 25
    possible eco-cells, there is a probability of 4% chance of relocating
    back to the same (original) eco-cell, 16% chance of relocating to one
    of the 4-neighbour eco-cells and 32% chance of relocating to one of the
    8-neighbour eco-cells
    - no Ragaraja interpretation of genome
    - 1000 generations to be simulated
'''
# needed to run this example without prior
# installation of DOSE into Python site-packages
try:
    import run_examples_without_installation
except ImportError: pass

# Example codes starts from here
import dose, random

parameters = {
              "simulation_name": "09_long_migration_isolated_mating",
              "population_names": ['pop_01'],
              "population_locations": [[(x,y,z)
                                         for x in range(5)
                                             for y in range(5)
                                                 for z in range(1)]],
              "deployment_code": 3,
              "chromosome_bases": ['0','1'],
              "background_mutation": 0.001,
              "additional_mutation": 0,
              "mutation_type": 'point',
              "chromosome_size": 5000,
              "genome_size": 1,
              "max_tape_length": 50,
              "clean_cell": True,
              "interpret_chromosome": False,
              "max_codon": 2000,
              "population_size": 1250,
              "eco_cell_capacity": 50,
              "world_x": 5,
              "world_y": 5,
              "world_z": 1,
              "goal": 0,
              "maximum_generations": 1000,
              "fossilized_ratio": 0.01,
              "fossilized_frequency": 100,
              "print_frequency": 10,
              "ragaraja_version": 0,
              "ragaraja_instructions": ['000', '001', '010',
                                        '011', '100', '101'],
              "eco_buried_frequency": 1250,
              "database_file": "sim09_long_migration.db",
              "database_logging_frequency": 1
             }

class simulation_functions(dose.dose_functions):

    def organism_movement(self, Populations, pop_name, World): pass
    def organism_location(self, Populations, pop_name, World):
        for location in parameters["population_locations"][0]:
            group = dose.filter_location(location,
                                         Populations[pop_name].agents)
            for i in range(int(round((len(group) * 0.1)))):
                (x,y,z) = dose.simulation_calls.coordinates(location)
                World.ecosystem[x][y][z]['organisms'] -= 1
                immigrant = random.choice(Populations[pop_name].agents)
                while immigrant not in group:
                    immigrant = random.choice(Populations[pop_name].agents)
                new_location = \
                    random.choice(parameters["population_locations"][0])
                while new_location == location:
                    new_location = \
                        random.choice(parameters["population_locations"][0])
                immigrant.status['location'] = new_location
                (x,y,z) = dose.simulation_calls.coordinates(new_location)
                World.ecosystem[x][y][z]['organisms'] += 1

    def ecoregulate(self, World): pass

    def update_ecology(self, World, x, y, z): pass

    def update_local(self, World, x, y, z): pass

    def report(self, World): pass

    def fitness(self, Populations, pop_name): pass

    def mutation_scheme(self, organism):
        organism.genome[0].rmutate(parameters["mutation_type"],
                                   parameters["additional_mutation"])

    def prepopulation_control(self, Populations, pop_name): pass

    def mating(self, Populations, pop_name):
        for location in parameters["population_locations"][0]:
            group = dose.filter_location(location,
                                         Populations[pop_name].agents)
            for x in range(len(group)//2):
                parents = []
                for i in range(2):
                    parents.append(random.choice(
                    Populations[pop_name].agents))
                    while parents[i] not in group:
                        parents[i] = \
                            random.choice(Populations[pop_name].agents)
                    Populations[pop_name].agents.remove(parents[i])
                crossover_pt = random.randint(0,
                                   len(parents[0].genome[0].sequence))
                (new_chromo1, new_chromo2) = \
                    dose.genetic.crossover(parents[0].genome[0],
                                           parents[1].genome[0],
                                           crossover_pt)
                children = [dose.genetic.Organism([new_chromo1],
                                             parameters["mutation_type"],
                                             parameters["additional_mutation"]),
                            dose.genetic.Organism([new_chromo2],
                                             parameters["mutation_type"],
                                             parameters["additional_mutation"])]
                for child in children:
                    child.status['parents'] = [parents[0].status['identity'],
                                               parents[1].status['identity']]
                    child.status['location'] = location
                    child.generate_name()
                    child.status['deme'] = pop_name
```

```
                        Populations[pop_name].agents.append(child)

    def postpopulation_control(self, Populations, pop_name): pass

    def generation_events(self, Populations, pop_name): pass

    def population_report(self, Populations, pop_name):
        report_list = []
        for organism in Populations[pop_name].agents:
            identity = str(organism.status['identity'])
            report_list.append(identity)
        return '\n'.join(report_list)

    def database_report(self, con, cur, start_time,
                        Populations, World, generation_count):
        try:
            dose.database_report_populations(con, cur, start_time,
                                             Populations, generation_count)
        except: pass
        try:
            dose.database_report_world(con, cur, start_time,
                                       World, generation_count)
        except: pass

    def deployment_scheme(self, Populations, pop_name, World): pass

dose.simulate(parameters, simulation_functions)
```

Example for Demonstrating Revival of Simulation from Database

```
'''
Reviving a simulation from simulation logging database and run the
simulation for another 200 generations.

Example 09 ran the simulation for 1000 generations and logged the events
into case_study_01.db file. This example revived the logged simulation
from case_study_01.db file using the simulation start time as identifier
(2013-10-19-1382200534.1) and run the simulation from generation 1001 to
1200. The events of generation 1001 to 1200 are logged in case_study_01.db
file.
'''
try:
    import run_examples_without_installation
except ImportError: pass

import dose, genetic
import os, random

'''
Needs pre-existing 09_no_migration_isolated_mating.py.
Change simulation_time below to the corresponding starting time
of the said simulation.
'''
rev_parameters = {"database_source" : "case_study_01.db",
                  "simulation_time": "2013-10-19-1382200534.1",
                  "population_locations": [[(x,y,z)
                                            for x in xrange(5)
                                                for y in xrange(5)
                                                    for z in xrange(1)]],
                  "rev_start" : [1000],
                  "extend_gen" : 200,
                  "simulation_name": "08_revive_simulation_03",
                  "chromosome_bases": ['0','1'],
                  "background_mutation": 0.1,
                  "additional_mutation": 0,
                  "mutation_type": 'point',
                  "max_tape_length": 50,
                  "clean_cell": True,
                  "interpret_chromosome": True,
                  "max_codon": 2000,
                  "goal": 0,
                  "eco_cell_capacity": 100,
                  "fossilized_ratio": 0.01,
                  "fossilized_frequency": 20,
                  "print_frequency": 10,
                  "ragaraja_version": 0,
                  "ragaraja_instructions": ['000', '001', '010',
                                            '011', '100', '101'],
                  "eco_buried_frequency": 100,
                  "database_file": "case_study_01.db",
                  "database_logging_frequency": 1
                  }

class simulation_functions(dose.dose_functions):

    def organism_movement(self, Populations, pop_name, World): pass

    def organism_location(self, Populations, pop_name, World): pass

    def ecoregulate(self, World): pass

    def update_ecology(self, World, x, y, z): pass

    def update_local(self, World, x, y, z): pass

    def report(self, World): pass

    def fitness(self, Populations, pop_name): pass

    def mutation_scheme(self, organism):
        organism.genome[0].rmutate(rev_parameters["mutation_type"],
                                   rev_parameters["additional_mutation"])

    def prepopulation_control(self, Populations, pop_name): pass

    def mating(self, Populations, pop_name):
        for location in rev_parameters["population_locations"][0]:
            group = dose.filter_location(location,
                                         Populations[pop_name].agents)
            for x in xrange(len(group)/2):
                parents = []
                for i in xrange(2):
                    parents.append(random.choice(Populations[pop_name].agents))
                    while parents[i] not in group:
                        parents[i] = \
                            random.choice(Populations[pop_name].agents)
                    Populations[pop_name].agents.remove(parents[i])
                crossover_pt = random.randint(0,
                               len(parents[0].genome[0].sequence))
                (new_chromo1, new_chromo2) = \
                     genetic.crossover(parents[0].genome[0],
                                       parents[1].genome[0],
                                       crossover_pt)
                children = [genetic.Organism([new_chromo1],
                                         rev_parameters["mutation_type"],
                                         rev_parameters["additional_mutation"]),
                            genetic.Organism([new_chromo2],
                                        rev_parameters["mutation_type"],
                                        rev_parameters["additional_mutation"])]
                for child in children:
                    child.status['location'] = location
                    child.generate_name()
                    child.status['deme'] = pop_name
                    Populations[pop_name].agents.append(child)

    def postpopulation_control(self, Populations, pop_name): pass

    def generation_events(self, Populations, pop_name): pass

    def population_report(self, Populations, pop_name):
        report_list = []
        for organism in Populations[pop_name].agents:
            chromosome = ''.join(organism.genome[0].sequence)
            location = str(organism.status['location'])
            report_list.append(chromosome + '  ' + location)
        return '\n'.join(report_list)

    def database_report(self, con, cur, start_time,
                        Populations, World, generation_count):
        try:
            dose.database_report_populations(con, cur, start_time,
                                             Populations, generation_count)
        except: pass
        try:
            dose.database_report_world(con, cur, start_time,
                                       World, generation_count)
        except: pass

    def deployment_scheme(self, Populations, pop_name, World): pass

dose.revive_simulation(rev_parameters, simulation_functions)
```

References

Batut B, Parsons DP, Fischer S, Beslon G, Knibbe, C. 2013. *In silico* experimental evolution: a tool to test evolutionary scenarios. BMC Bioinformatics 14(Suppl 15): S11.

Bersini, H. 2009. How artificial life relates to theoretical biology. Origins of Life: Self-Organization and/or Biological Evolution?, 61-78.

Castillo, CFG, Ling, MHT. 2014a. Digital Organism Simulation Environment (DOSE): A Library for Ecologically-Based In Silico Experimental Evolution. Advances in Computer Science: an International Journal 3(1): 44-50.

Castillo, CFG, Ling, MHT. 2014b. Resistant Traits in Digital Organisms Do Not Revert Preselection Status despite Extended Deselection: Implications to Microbial Antibiotics Resistance. BioMed Research International 2014, Article ID 648389.

Castillo, CFG, Chay ZE, Ling, MHT. 2015. Resistance Maintained in Digital Organisms Despite Guanine/Cytosine-Based Fitness Cost and Extended De-Selection: Implications to Microbial Antibiotics Resistance. MOJ Proteomics & Bioinformatics 2(2): 00039.

Langton, CG. 1986. Studying artificial life with cellular automata. Physica D: Nonlinear Phenomena 22, 120-149.

Lim, JZR, Aw, ZQ, Goh, DJW, How, JA, Low, SXZ, Loo, BZL, Ling, MHT. 2010. A genetic algorithm framework grounded in biology. The Python Papers Source Codes 2: 6.

Ling, MHT. 2012a. An artificial life simulation library based on genetic algorithm, 3-character genetic code and biological hierarchy. The Python Papers 7: 5.

Ling, MHT. 2012b. Ragaraja 1.0: The Genome Interpreter of Digital Organism Simulation Environment (DOSE). The Python Papers Source Codes 4: 2.

Raymond L, Plantegenest M, Vialatte, A. 2013. Migration and dispersal may drive to high genetic variation and significant genetic mixing: the case of two agriculturally important, continental hoverflies (*Episyrphus balteatus* and *Sphaerophoria scripta*). Molecular Ecology 22(21): 5329-5339.

Relethford, JH. 1988. Heterogeneity of long-distance migration in studies of genetic structure. Annals of Human Biology 15(1): 55-63.

Watson, R. 2012. Is evolution by natural selection the algorithm of biological evolution? In C. Adami, DM. Bryson, C. Ofria, and RT. Pennock (eds.) Artificial Life XIII: Proceedings of the Thirteenth International Conference on the Synthesis and Simulation of Living Systems Artificial Life XIII: 13th International Conference on the Simulation and Synthesis of Living Systems, pp. 121-128.

Appendix: How to Write a Simulation

A simulation script is a Python code with 4 sections:

1. Import dose library by "import dose"
2. A dictionary of simulation parameters
3. A class containing simulation methods/functions
4. Run simulation by calling DOSE simulation function with the simulation parameters and simulation methods as parameters: dose.simulate(parameters, simulation_functions)

Since (1) and (4) are standard, only (2) and (3) are elaborated here.

A dictionary of simulation parameters provides for the set of parameters needed to construct the simulation and store the simulation results.

```
parameters = {"simulation_name": "01_basic_functions_one_cell_deployment",
              "population_names": ['pop_01'],
              "population_locations": [[(0,0,0)]],
              "deployment_code": 1,
              "chromosome_bases": ['0','1'],
              "background_mutation": 0.1,
              "additional_mutation": 0,
              "mutation_type": 'point',
              "chromosome_size": 30,
              "genome_size": 1,
              "max_tape_length": 50,
              "clean_cell": True,
              "interpret_chromosome": True,
              "max_codon": 2000,
              "population_size": 100,
              "eco_cell_capacity": 100,
              "world_x": 5,
              "world_y": 5,
              "world_z": 5,
              "goal": 0,
              "maximum_generations": 100,
              "fossilized_ratio": 0.01,
              "fossilized_frequency": 20,
              "print_frequency": 10,
              "ragaraja_version": 0,
              "ragaraja_instructions": ['000', '001', '010',
                                        '011', '100', '101'],
              "eco_buried_frequency": 100,
              "database_file": "simulation.db",
              "database_logging_frequency": 1
             }
```

A class containing simulation methods/functions provides a set of simulation-specific functions.

```
class simulation_functions(dose.dose_functions):

    def organism_movement(self, Populations, pop_name, World): pass

    def organism_location(self, Populations, pop_name, World): pass

    def ecoregulate(self, World): pass

    def update_ecology(self, World, x, y, z): pass

    def update_local(self, World, x, y, z): pass

    def report(self, World): pass

    def fitness(self, Populations, pop_name): pass

    def mutation_scheme(self, organism):
        organism.genome[0].rmutate(parameters["mutation_type"],
                                   parameters["additional_mutation"])

    def prepopulation_control(self, Populations, pop_name): pass

    def mating(self, Populations, pop_name): pass

    def postpopulation_control(self, Populations, pop_name): pass

    def generation_events(self, Populations, pop_name): pass

    def population_report(self, Populations, pop_name):
        sequences = [''.join(org.genome[0].sequence) for org in
Populations[pop_name].agents]
        identities = [org.status['identity'] for org in Populations[pop_name].agents]
        locations = [str(org.status['location']) for org in
Populations[pop_name].agents]
        demes = [org.status['deme'] for org in Populations[pop_name].agents]
        return '\n'.join(sequences)

    def database_report(self, con, cur, start_time,
                        Populations, World, generation_count):
        try:
            dose.database_report_populations(con, cur, start_time,
                                             Populations, generation_count)
        except: pass
        try:
            dose.database_report_world(con, cur, start_time,
                                       World, generation_count)
        except: pass

    def deployment_scheme(self, Populations, pop_name, World): pass
```