Xentica Documentation

Release 0.1.0

Andrey Zamaraev

Oct 31, 2022

Contents

1 User Guide **3**
- 1.1 Installation Instructions . . . 3
- 1.2 Testing . . . 5
- 1.3 Tutorial . . . 6

2 API Reference **13**
- 2.1 The Core (xentica.core) . . . 13
- 2.2 The Topology (xentica.core.topology) . . . 26
- 2.3 Tools (xentica.tools) . . . 32
- 2.4 The Seeds (xentica.seeds) . . . 34
- 2.5 The Bridge (xentica.bridge) . . . 37
- 2.6 Utilities (xentica.utils) . . . 38

3 Indices and tables **39**

Python Module Index **41**

Index **43**

Xentica is the framework to build GPU-accelerated models for multi-dimensional cellular automata. Given pure Python definitions, it generates kernels in CUDA C and runs them on NVIDIA hardware.

Warning: Current version is a work-in-progress, it works to some degree, but please do not expect something beneficial from it. As planned, really useful stuff would be available only starting from version 0.3.

CHAPTER 1

User Guide

If you brave enough to ignore the warning above, dive right into this guide. Hopefully, you will manage to install Xentica on your system and at least run some examples. Otherwise, just read Tutorial and watch some videos to decide is it worth waiting for future versions.

1.1 Installation Instructions

Xentica is planned to run with several GPU backends in the future, like *CUDA*, *OpenCL* and *GLSL*. However, right now, only *CUDA* is supported.

> **Warning:** If your GPU is not CUDA-enabled, this guide is **not** for you. The framework will just not run, no matter how hard you try. You may check the list of CUDA-enabled cards if you have any doubts.

Note: This page is currently containing instructions only for Debian-like systems. If you are on another system, you still can use links to pre-requisites in order to install them. If so, please contact us by opening an issue on GitHub. We could help you if you'll meet some troubles during installation, and also your experience could be used to improve this document.

1.1.1 Core Prerequisites

In order to run CA models without any visualization, you have to correctly install the following software.

- NVIDIA CUDA Toolkit

  Generally, you can install it just from your distribution's repository:

```
sudo apt-get install nvidia-cuda-toolkit
```

Nevertheless, default packages are often out of date, so in case you have one of those latest cool GPUs, you may want to upgrade to the latest CUDA version from official NVIDIA source. We'll hint you with a good article explaining how to do it. But you are really on your own with this stuff.

- Python 3.5+

  Your distribution should already have all you need:

```
sudo apt-get install python3 python3-dev python3-pip wheel
```

- NumPy

  Once Python3 is correctly installed, you can install NumPy by:

```
pip3 install numpy
```

- PyCUDA

  If CUDA is correctly installed, you again can simply install PyCUDA with `pip`:

```
pip3 install pycuda
```

Other pre-requisites should be transparently installed with the main Xentica package.

1.1.2 GUI Prerequisites

If you are planning to run visual examples with Moire GUI, you have to install Kivy framework.

Its pre-requisites could be installed by:

```
sudo apt-get install \
  build-essential \
  git \
  ffmpeg \
  libsdl2-dev \
  libsdl2-image-dev \
  libsdl2-mixer-dev \
  libsdl2-ttf-dev \
  libportmidi-dev \
  libswscale-dev \
  libavformat-dev \
  libavcodec-dev \
  zlib1g-dev
```

Then, to install stable Kivy:

```
pip3 install Cython==0.25 Kivy==1.10.0
```

On the latest Debian distributions you can meet conflicts with `libsdl-mixer`. Then, try to install the latest developer version, like:

```
pip3 install Cython==0.27.3 git\+https://github.com/kivy/kivy.git
```

If you have any other troubles with that, please refer to the official Kivy installation instructions.

1.1.3 Main package

Xentica package could be installed with:

```
pip3 install xentica
```

Note, it does not depend on pre-requisites described above, but you still need to install them properly, or Xentica will not run.

1.1.4 Run Xentica examples

In order to run the Game of Life model built with Xentica:

```
pip3 install moire
wget https://raw.githubusercontent.com/a5kin/xentica/master/examples/game_of_life.py
python3 game_of_life.py
```

Or, if you are using `optirun`:

```
optirun python3 game_of_life.py
```

1.2 Testing

1.2.1 Prerequisites

Our test suite is based on tox. It allows us to run tests in all supported Python environments with a single `tox` command and also automates checks for package build, docs, coverage, code style, and performance.

So, in order to run Xentica tests you at minimum have to set up the CUDA environment properly (as described above), and install `tox`:

```
pip3 install tox
```

If you are planning to run the full test suite, you also need to install all necessary Python interpreters: 3.5-3.7 and pypy3, along with dev headers to build NumPy and PyCUDA.

On Ubuntu, regular Python interpreters are available with the amazing deadsnakes repo:

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt-get update
sudo apt-get install python3.5 python3.5-dev
sudo apt-get install python3.6 python3.6-dev
sudo apt-get install python3.7 python3.7-dev
```

Pypy3, however, comes in binaries (make sure you downloaded the latest):

```
wget -q -P /tmp https://bitbucket.org/pypy/pypy/downloads/pypy3.5-v7.0.0-linux64.tar.
↪bz2
sudo tar -x -C /opt -f /tmp/pypy3.5-v7.0.0-linux64.tar.bz2
rm /tmp/pypy3.5-v7.0.0-linux64.tar.bz2
sudo mv /opt/pypy3.5-v7.0.0-linux64 /opt/pypy3
sudo ln -s /opt/pypy3/bin/pypy3 /usr/local/bin/pypy3
```

1.2.2 Run tests

To run full tests:

```
git clone https://github.com/a5kin/xentica.git
cd xentica
tox
```

Or, if you are using `optirun`:

```
optirun tox
```

For the first time, it would take an amount of time to download/install environments and all its dependencies. Tox will automatically set up all necessary stuff for you, including NumPy and PyCUDA. Subsequent runs should be much quicker, as everything is already set up. In the developer's environment (Ubuntu 18.04) it takes ~42 sec to finish the full test suite.

If you run tests often, it would also be helpful to get less verbose output. For that, you could execute a strict version of tox:

```
tox -q
```

Or if you'd like to skip all uninstalled interpreters:

```
tox -s
```

Or even quicker, for immediate test purposes, you could run the default Python3 interpreter tests only with code style and coverage:

```
tox -q -e flake8,coverage
```

You could also check the full list of available environments with:

```
tox -l -v
```

If you don't mind, please update us with the metrics under the "Benchmark" section, along with the info about your GPU, environment and Xentica version. It would help us analyze performance and plan for future optimizations.

Note: When planning for a pull request in core Xentica repo, it is a good practice to run a full test suite with `tox -q`. It will be accepted at least if everything is green =)

1.3 Tutorial

The tutorial is coming soon. Meanwhile, you may check the official Game Of Life example.

```
"""
A collection of models derived from Conway's Game Of Life.

Experiment classes included.

"""
import importlib

from xentica import core
from xentica import seeds
from xentica.core import color_effects
from xentica.tools.rules import LifeLike
```

(continues on next page)

(continued from previous page)

```
class GameOfLife(core.CellularAutomaton):
    """
    The classic CA built with Xentica framework.

    It has only one property called ``state``, which is positive
    integer with max value of 1.

    """

    state = core.IntegerProperty(max_val=1)

    class Topology:
        """
        Mandatory class for all ``CellularAutomaton`` instances.

        All class variables below are also mandatory.

        Here, we declare the topology as a 2-dimensional orthogonal
        lattice with Moore neighborhood, wrapped to a 3-torus.

        """

        dimensions = 2
        lattice = core.OrthogonalLattice()
        neighborhood = core.MooreNeighborhood()
        border = core.TorusBorder()

    def emit(self):
        """
        Implement the logic of emit phase.

        Statements below will be translated into C code as emit kernel
        at the moment of class creation.

        Here, we just copy main state to surrounding buffers.

        """
        for buf in self.buffers:
            buf.state = self.main.state

    def absorb(self):
        """
        Implement the logic of absorb phase.

        Statements below will be translated into C code as well.

        Here, we sum all neigbors buffered states and apply Conway
        rule to modify cell's own state.

        """
        neighbors_alive = core.IntegerVariable()
        for i in range(len(self.buffers)):
            neighbors_alive += self.neighbors[i].buffer.state
        is_born = (8 >> neighbors_alive) & 1
        is_sustain = (12 >> neighbors_alive) & 1
```

(continues on next page)

(continued from previous page)

```
        self.main.state = is_born | is_sustain & self.main.state

    @color_effects.MovingAverage
    def color(self):
        """
        Implement the logic of cell's color calculation.

        Must return a tuple of RGB values computed from ``self.main``
        properties.

        Also, must be decorated by a class from ``color_effects``
        module.

        Here, we simply define 0 state as pure black, and 1 state as
        pure white.

        """
        red = self.main.state * 255
        green = self.main.state * 255
        blue = self.main.state * 255
        return (red, green, blue)


class GameOfLifeStatic(GameOfLife):
    """
    Game of Life variant with static border made of live cells.

    This is an example of how easy you can inherit other models.

    """

    class Topology(GameOfLife.Topology):
        """
        You can inherit parent class ``Topology``.

        Then, override only necessary variables.

        """

        border = core.StaticBorder(1)


class GameOfLifeColor(GameOfLife):
    """
    Game Of Life variant with RGB color.

    This is an example of how to use multiple properties per cell.

    """

    state = core.IntegerProperty(max_val=1)
    red = core.IntegerProperty(max_val=255)
    green = core.IntegerProperty(max_val=255)
    blue = core.IntegerProperty(max_val=255)

    def emit(self):
        """Copy all properties to surrounding buffers."""
```

(continues on next page)

(continued from previous page)

```
        for buf in self.buffers:
            buf.state = self.main.state
            buf.red = self.main.red
            buf.green = self.main.green
            buf.blue = self.main.blue

    def absorb(self):
        """
        Calculate RGB as neighbors sum for living cell only.

        Note, parent ``absorb`` method should be called using direct
        class access, not via ``super``.

        """
        GameOfLife.absorb(self)
        red_sum = core.IntegerVariable()
        green_sum = core.IntegerVariable()
        blue_sum = core.IntegerVariable()
        for i in range(len(self.buffers)):
            red_sum += self.neighbors[i].buffer.red + 1
            green_sum += self.neighbors[i].buffer.green + 1
            blue_sum += self.neighbors[i].buffer.blue + 1
        self.main.red = red_sum * self.main.state
        self.main.green = green_sum * self.main.state
        self.main.blue = blue_sum * self.main.state

    @color_effects.MovingAverage
    def color(self):
        """Calculate color as usual."""
        red = self.main.state * self.main.red
        green = self.main.state * self.main.green
        blue = self.main.state * self.main.blue
        return (red, green, blue)


class GameOfLife6D(GameOfLife):
    """
    Game of Life variant in 6D.

    Nothing interesting, just to prove you can do it with ease.

    """

    class Topology(GameOfLife.Topology):
        """
        Hyper-spacewalk, is as easy as increase ``dimensions`` value.

        However, we are also changing neighborhood to Von Neumann
        here, to prevent neighbors number exponential grow.

        """

        dimensions = 6
        neighborhood = core.VonNeumannNeighborhood()


class LifelikeCA(GameOfLife):
```

(continues on next page)

(continued from previous page)

```
    """Lifelike CA with a flexible rule that could be changed at runtime."""
    rule = core.Parameter(
        default=LifeLike.golly2int("B3/S23"),
        interactive=True,
    )

    def absorb(self):
        """Implement parent's clone with a rule as a parameter."""
        neighbors_alive = core.IntegerVariable()
        for i in range(len(self.buffers)):
            neighbors_alive += self.neighbors[i].buffer.state
        is_born = (self.rule >> neighbors_alive) & 1
        is_sustain = (self.rule >> 9 >> neighbors_alive) & 1
        self.main.state = is_born | is_sustain & self.main.state

    def step(self):
        """Change the rule interactively after some time passed."""
        if self.timestep == 23:
            self.rule = LifeLike.golly2int("B3/S23")
        super().step()


class GOLExperiment(core.Experiment):
    """
    Particular experiment for the vanilla Game of Life.

    Here, we define constants and initial conditions from which the
    world's seed will be generated.

    The ``word`` is an RNG seed string. The ``size``, ``zoom`` and
    ``pos`` are board contstants. The ``seed`` is a pattern used in
    the initial board state generation.

    ``BigBang`` is a pattern when small area initialized with a
    high-density random values.

    """

    word = "OBEY XENTICA"
    size = (640, 360, )
    zoom = 3
    pos = [0, 0]
    seed = seeds.patterns.BigBang(
        pos=(320, 180),
        size=(100, 100),
        vals={
            "state": seeds.random.RandInt(0, 1),
        }
    )


class GOLExperiment2(GOLExperiment):
    """
    Another experiment for the vanilla GoL.

    Since it is inherited from ``GOLExperiment``, we can redefine only
    values we need.
```

(continues on next page)

(continued from previous page)

```
    ``PrimordialSoup`` is a pattern when the whole board is
    initialized with low-density random values.

    """

    word = "XENTICA IS YOUR GODDESS"
    seed = seeds.patterns.PrimordialSoup(
        vals={
            "state": seeds.random.RandInt(0, 1),
        }
    )


class GOLExperimentColor(GOLExperiment):
    """
    The experiment for ``GameOfLifeColor``.

    Here, we introduce ``fade_out`` constant, which is used in
    rendering and slowly fading out the color of cells.

    Note, it is only an aestetic effect, and does not affect the real
    cell state.

    """

    fade_in = 255
    fade_out = 10
    smooth_factor = 1
    seed = seeds.patterns.PrimordialSoup(
        vals={
            "state": seeds.random.RandInt(0, 1),
            "red": seeds.random.RandInt(0, 255),
            "green": seeds.random.RandInt(0, 255),
            "blue": seeds.random.RandInt(0, 255),
        }
    )


class GOLExperiment6D(GOLExperiment2):
    """
    Special experiment for 6D Life.

    Here, we define the world with 2 spatial and 4 looped
    micro-dimensions, 3 cells per micro-dimension.

    As a result, we get large quasi-stable oscillators, looping over
    micro-dimensions. Strangely formed, but nothing interesting,
    really.

    """

    size = (640, 360, 3, 3, 3, 3)


class DiamoebaExperiment(GOLExperiment):
    """Experiment with the interactive rule."""
```

(continues on next page)

(continued from previous page)

```
    rule = LifeLike.golly2int("B35678/S5678")


def main():
    """Run model/experiment interactively."""
    moire = importlib.import_module("moire")
    model = GameOfLifeColor(GOLExperimentColor)
    gui = moire.GUI(runnable=model)
    gui.run()


if __name__ == "__main__":
    main()
```

CHAPTER 2

API Reference

2.1 The Core (xentica.core)

Xentica core functionality is available via modules from this package.

In addition, you may use `core` package as a shortcut to the main classes of the framework.

- **Base classes**
  - `core.CellularAutomaton` → *xentica.core.base.CellularAutomaton*
  - `core.Experiment` → *xentica.core.experiment.Experiment*
- **Lattices**
  - `core.OrthogonalLattice` → *xentica.core.topology.lattice.OrthogonalLattice*
- **Neighborhoods**
  - `core.MooreNeighborhood` → *xentica.core.topology.neighborhood.MooreNeighborhood*
  - `core.VonNeumannNeighborhood` → *xentica.core.topology.neighborhood.VonNeumannNeighborhood*
- **Borders**
  - `core.TorusBorder` → *xentica.core.topology.border.TorusBorder*
  - `core.StaticBorder` → *xentica.core.topology.border.StaticBorder*
- **Properties**
  - `core.IntegerProperty` → *xentica.core.properties.IntegerProperty*
  - `core.FloatProperty` → `xentica.core.properties.FloatProperty`
  - `core.TotalisticRuleProperty` → `xentica.core.properties.TotalisticRuleProperty`

  - `core.RandomProperty` → `xentica.core.properties.RandomProperty`

- **Parameters**
  - `core.Parameter` → *xentica.core.parameters.Parameter*

- **Variables**
  - `core.IntegerVariable` → *xentica.core.variables.IntegerVariable*
  - `core.FloatVariable` → *xentica.core.variables.FloatVariable*

The classes listed above are all you need to build CA models and experiments with Xentica, unless you are planning to implement custom core features like new lattices, borders, etc.

2.1.1 Base Classes (xentica.core.base)

The module with the base class for all CA models.

All Xentica models should be inherited from *CellularAutomaton* base class. Inside the model, you should correctly define the `Topology` class and describe the CA logic in `emit()`, `absorb()` and `color()` methods.

`Topology` is the place where you define the dimensionality, lattice, neighborhood and border effects for your CA. See *xentica.core.topology* package for details.

The logic of the model will follow Buffered State Cellular Automaton (BSCA) principle. In general, every cell mirrors its state in buffers by the number of neighbors, each buffer intended for one of the neighbors. Then, at each step, the interaction between cells is performed via buffers in 2-phase emit/absorb process. A more detailed description of the BSCA principle is available in The Core section of The Concept document.

`emit()` describes the logic of the first phase of BSCA. At this phase, you should fill the cell's buffers with corresponding values, depending on the cell's main state and (optionally) on neighbors' main states. The easiest logic is to just copy the main state to buffers. It is especially useful when you're intending to emulate classic CA (like Conway's Life) with BSCA. Write access to the main state is prohibited there.

`absorb()` describes the logic of the second phase of BSCA. At this phase, you should set the cell's main state, depending on neighbors' buffered states. Write access to buffers is prohibited there.

`color()` describes how to calculate cell's color from its raw state. See detailed instructions on it in *xentica.core.color_effects*.

The logic of the functions from above will be translated into C code at the moment of class instance creation. For the further instructions on how to use the cell's main and buffered states, see *xentica.core.properties*, for the instructions on variables and expressions with them, see *xentica.core.variables*. Another helpful thing is meta-parameters, which are described in *xentica.core.parameters*.

A minimal example, the CA where each cell is taking the mean value of its neighbors each step:

```
from xentica import core
from xentica.core import color_effects

class MeanCA(core.CellularAutomaton):

    state = core.IntegerProperty(max_val=255)

    class Topology:
        dimensions = 2
        lattice = core.OrthogonalLattice()
        neighborhood = core.MooreNeighborhood()
        border = core.TorusBorder()
```

(continues on next page)

(continued from previous page)

```
    def emit(self):
        for i in range(len(self.buffers)):
            self.buffers[i].state = self.main.state

    def absorb(self):
        s = core.IntegerVariable()
        for i in range(len(self.buffers)):
            s += self.neighbors[i].buffer.state
        self.main.state = s / len(self.buffers)

    @color_effects.MovingAverage
    def color(self):
        v = self.main.state
        return (v, v, v)
```

class xentica.core.base.**BSCA**

Bases: type

The meta-class for *CellularAutomaton*.

Prepares `main`, `buffers` and `neighbors` class variables being used in `emit()`, `absorb()` and `color()` methods.

mandatory_fields = ('dimensions', 'lattice', 'neighborhood', 'border')

class xentica.core.base.**CellularAutomaton**(*experiment_class*)

Bases: xentica.core.base.Translator

The base class for all Xentica models.

Generates GPU kernels source code, compiles them, initializes necessary GPU arrays and populates them with the seed.

After initialization, you can run a step-by-step simulation and render the field at any moment:

```
from xentica import core
import moire

class MyCA(core.CellularAutomaton):
    # ...

class MyExperiment(core.Experiment):
    # ...

ca = MyCA(MyExperiment)
ca.set_viewport((320, 200))

# run CA manually for 100 steps
for i in range(100):
    ca.step()
# render current timestep
frame = ca.render()

# or run the whole process interactively with Moire
gui = moire.GUI(runnable=ca)
gui.run()
```

Parameters **experiment_class** – `Experiment` instance, holding all necessary parameters

for the field initialization.

apply_speed(*dval*)
Change the simulation speed.

Usable only in conduction with Moire, although you can use the `speed` value in your custom GUI too.

Parameters **dval** – Delta by which speed is changed.

load(*filename*)
Load the CA state from `filename` file.

render()
Render the field at the current timestep.

You must call *set_viewport()* before do any rendering.

Returns NumPy array of `np.uint8` values, `width * height * 3` size. The RGB values are consecutive.

save(*filename*)
Save the CA state into `filename` file.

set_viewport(*size*)
Set the viewport (camera) size and initialize GPU array for it.

Parameters **size** – tuple with the width and height in pixels.

step()
Perform a single simulation step.

`timestep` attribute will hold the current step number.

toggle_pause()
Toggle `paused` flag.

When paused, the `step()` method does nothing.

class xentica.core.base.**CachedNeighbor**
Bases: `object`

The utility class, intended to hold the main and buffered CA state.

2.1.2 Experiments (xentica.core.experiment)

The collection of classes to describe experiments for CA models.

The experiment is a class with CA parameters stored as class variables. Different models may have a different set of parameters. To make sure all set correctly, you should inherit your experiments from *Experiment* class.

A quick example:

```
from xentica import core, seeds

class MyExperiment(core.Experiment):
    # RNG seed string
    word = "My Special String"
    # field size
    size = (640, 360, )
    # initial field zoom
    zoom = 3
    # initial field shift
```

(continues on next page)

(continued from previous page)

```
    pos = [0, 0]
    # A pattern used in initial board state generation.
    # BigBang is a small area initialized with high-density random values.
    seed = seeds.patterns.BigBang(
        # position Big Bang area
        pos=(320, 180),
        # size of Big Bang area
        size=(100, 100),
        # algorithm to generate random values
        vals={
            "state": seeds.random.RandInt(0, 1),
        }
    )
```

class xentica.core.experiment.**Experiment**

Bases: object

The base class for all experiments.

Right now doing nothing, but will be improved in future versions. So it is advised to inherit your experiments from it.

2.1.3 Properties (xentica.core.properties)

The collection of classes to describe properties of CA models.

Warning: Do not confuse with Python properties.

Xentica properties are declaring as class variables and helping you to organize CA state into complex structures.

Each *CellularAutomaton* instance should have at least one property declared. The property name is up to you. If your model has just one value as a state (like in most classic CA), the best practice is to call it `state` as follows:

```
from xentica import core

class MyCA(core.CellularAutomaton):
    state = core.IntegerProperty(max_val=1)
    # ...
```

Then, you can use it in expressions of `emit()`, `absorb()` and `color()` functions as:

`self.main.state` to get and set main state;

`self.buffers[i].state` to get and set i-th buffered state;

`self.neighbors[i].buffer.state` to get and set i-th neighbor buffered state.

Xentica will take care of all other things, like packing CA properties into binary representation and back, storing and getting corresponding values from VRAM, etc.

Most of properties will return `DeferredExpression` on access, so you can use them safely in mixed expressions:

```
self.buffers[i].state = self.main.state + 1
```

class xentica.core.properties.**Property**

Bases: *xentica.core.expressions.DeferredExpression*, *xentica.core.mixins.BscaDetectorMixin*

Base class for all properties.

It has a vast set of default functionality already in places. Though, you are free to re-define it all to implement really custom behavior.

best_type
Get the type that suits best to store a property.

Returns tuple representing best type: `(bit_width, numpy_dtype, gpu_c_type)`

bit_width
Get the number of bits necessary to store a property.

Returns Positive integer, a property's bit width.

buf_num
Get a buffer's index, associated to a property.

calc_bit_width()
Calculate the property's bit width.

This is the method you most likely need to override. It will be called from *bit_width()*.

Returns Positive integer, the calculated property's width in bits.

coords_declared
Test if the coordinates variables are declared.

ctype
Get the C type, based on the result of *best_type()*.

Returns C type that suits best to store a property.

declare_once()
Generate the C code to declare a variable holding a cell's state.

You must push the generated code to the BSCA via `self.bsca.append_code()`, then declare necessary stuff via `self.bsca.declare()`.

You should also take care of skipping the whole process if things are already declared.

declared
Test if the state variable is declared.

dtype
Get the NumPy dtype, based on the result of *best_type()*.

Returns NumPy dtype that suits best to store a property.

nbr_num
Get a neighbor's index, associated to a property.

pack_cast(*expr*)
Fix given expression for safe use with packing.

Parameters **expr** – Any C expression.

Returns Fixed, correctly casted expression.

unpack_cast(*expr*)
Fix given expression for safe use with unpacking.

Parameters **expr** – Any C expression.

Returns Fixed, correctly casted expression.

width
Get the number of memory cells to store a property.

In example, if `ctype == "int"` and `bit_width == 64`, you need 2 memory cells.

Returns Positive integer, a property's width.

class xentica.core.properties.**IntegerProperty**(*max_val*)
Bases: *xentica.core.properties.Property*

Most generic property for you to use.

It is just a positive integer with the upper limit of `max_val`.

calc_bit_width()
Calculate the bit width, based on `max_val`.

class xentica.core.properties.**ContainerProperty**
Bases: *xentica.core.properties.Property*

A property acting as a holder for other properties.

Currently is used only for inner framework mechanics, in particular, to hold, pack and unpack all top-level properties.

It will be enhanced in future versions, and give you the ability to implement nested properties structures.

> **Warning:** Right now, direct use of this class is prohibited.

buf_num
Get a buffer's index, associated to a property.

calc_bit_width()
Calculate the bit width as a sum of inner properties' bit widths.

declare_once()
Do all necessary declarations for inner properties.

Also, implements the case of the off-board neighbor access.

Parameters **init_val** – The default value for the property.

deferred_write()
Pack the state and write its value to the VRAM.

`CellularAutomaton` calls this method at the end of the kernel processing.

nbr_num
Get a neighbor's index, associated to a property.

properties
Get inner properties.

unpacked
Test if inner properties are unpacked from the memory.

values()
Iterate over properties, emulating `dict` functionality.

2.1.4 Parameters (xentica.core.parameters)

The collection of classes to describe the model's meta-parameters.

The *parameter* is the value that influences the whole model's behavior in some way. Each parameter has a default value. Then you could customize them per each experiment or change interactively using `Bridge`.

There are two types of parameters in Xentica:

Non-interactive are constant during a single experiment run. The change of this parameter is impossible without a whole model's source code being rebuilt. The engine makes sure those params are correctly defined globally with the `#define` directive. So actually, even if you'll change their values at runtime, it doesn't affect the model in any way.

Interactive could be effectively changed at runtime, since engine traits them as extra arguments to CUDA kernels. That means, as long as you'll set a new value to an interactive parameter, it will be passed into the kernel(s) at the next timestep. Be warned though: every parameter declared as interactive, will degrade the model's performance further.

The example of parameters' usage:

```
from xentica import core
from xentica.tools.rules import LifeLike
from examples.game_of_life import GameOfLife, GOLExperiment


class LifelikeCA(GameOfLife):
    rule = core.Parameter(
        default=LifeLike.golly2int("B3/S23"),
        interactive=True,
    )

    def absorb(self):
        # parent's clone with parameter instead of hardcoded rule
        neighbors_alive = core.IntegerVariable()
        for i in range(len(self.buffers)):
            neighbors_alive += self.neighbors[i].buffer.state
        is_born = (self.rule >> neighbors_alive) & 1
        is_sustain = (self.rule >> 9 >> neighbors_alive) & 1
        self.main.state = is_born | is_sustain & self.main.state


class DiamoebaExperiment(GOLExperiment):
    rule = LifeLike.golly2int("B35678/S5678")


model = LifelikeCA(DiamoebaExperiment)
```

class xentica.core.parameters.**Parameter**(*default=0*, *interactive=False*)

Bases: *xentica.core.mixins.BscaDetectorMixin*

The implementation of Xentica meta-parameter.

Parameters

- **default** – The default value for the parameter to use when it's omitted in the experiment class.
- **interactive** – `True` if the parameter could be safely changed at runtime (more details above in the module description).

ctype
Get the parameter's C type.

dtype
Get the parameter's NumPy type.

name
Get the parameter's name.

value
Get the parameter's value directly.

2.1.5 Variables (xentica.core.variables)

The collection of classes to declare and use C variables and constants.

If the logic of your `emit()`, `absorb()` or `color()` functions requires the intermediate variables, you must declare them via classes from this module in the following way:

```
from xentica import core

class MyCA(core.CellularAutomaton):
    # ...

    def emit(self):
        myvar = core.IntegerVariable()
```

Then you can use them in mixed expressions, like this:

```
myvar += self.neighbors[i].buffer.state
self.main.state = myvar & 1
```

You may also define constants or other `#define` patterns with *Constant* class.

class xentica.core.variables.**Constant**(*name*, *value*)
Bases: *xentica.core.mixins.BscaDetectorMixin*

The class for defining constants and `#define` patterns.

Once you instantiate *Constant*, you must feed it to `CellularAutomaton.define_constant()` in order to generate the correct C code:

```
const = Constant("C_NAME", "some_value")
self.bsca.define_constant(const)
```

Parameters

- **name** – The name to use in `#define`.
- **value** – A value for the define, it will be converted to a string with `str()`.

get_define_code()
Get the C code for `#define`.

name
Get the name of the constant.

class xentica.core.variables.**Variable**(*val=None*, *name='var'*)
Bases: *xentica.core.expressions.DeferredExpression*, *xentica.core.mixins.BscaDetectorMixin*

The base class for all variables.

Most of the functionality for variables is already implemented in it. Though, you are free to re-define it all to implement really custom behavior.

Parameters

- **val** – The initial value for the variable.
- **name** – Fallback name to declare the variable with.

code
Get the variable name as a C code.

declare_once()
Declare the variable and assign the initial value to it.

var_name
Get the variable name.

class xentica.core.variables.**IntegerVariable**(*val='0'*, ***kwargs*)
Bases: *xentica.core.variables.Variable*

The variable intended to hold a positive integer.

var_type = 'unsigned int'
C type to use in definition.

class xentica.core.variables.**FloatVariable**(*val='0.0f'*, ***kwargs*)
Bases: *xentica.core.variables.Variable*

The variable intended to hold a 32-bit float.

var_type = 'float'
C type to use in definition.

2.1.6 Expressions (xentica.core.expressions)

The module holding base expressions classes.

class xentica.core.expressions.**DeferredExpression**(*code=''*)
Bases: object

The base class for other classes intended to be used in mixed expressions.

In particular, it is used in base *Variable* and *Property* classes.

Most of the magic methods dealing with binary and unary operators as well as augmented assigns are automatically overridden for this class. As a result, you can use its subclasses in mixed expressions with ordinary Python values. See the example in variables module description.

Allowed binary ops +, −, *, /, %, >>, <<, &, ^, |, <, <=, >, >=, ==, !=

Allowed unary ops +, −, ~, abs, int, float, round

Allowed augmented assigns +=, −=, *=, /=, %=, <<=, >>=, &=, ^=, |=

2.1.7 Color Effects (xentica.core.color_effects)

The collection of decorators for the color() method, each CA model should have.

The method should be decorated by one of the classes below, otherwise the correct model behavior will not be guaranteed.

All decorators are get the `(red, green, blue)` tuple from `color()` method, then process it to create some color effect.

A minimal example:

```
from xentica import core
from xentica.core import color_effects

class MyCA(core.CellularAutomaton):

    state = core.IntegerProperty(max_val=1)

    # ...

    @color_effects.MovingAverage
    def color(self):
        red = self.main.state * 255
        green = self.main.state * 255
        blue = self.main.state * 255
        return (red, green, blue)
```

class xentica.core.color_effects.**ColorEffect**(*func*)

Bases: *xentica.core.mixins.BscaDetectorMixin*

The base class for other color effects.

You may also use it as a standalone color effect decorator, it just doing nothing, storing the calculated RGB value directly.

To create your own class inherited from *ColorEffect*, you should override `__call__` method, and place a code of the color processing into `self.effect`. The code should process values of `new_r`, `new_g`, `new_b` variables and store the result back to them.

An example:

```
class MyEffect(ColorEffect):

    def __call__(self, *args):
        self.effect = "new_r += 20;"
        self.effect += "new_g += 15;"
        self.effect += "new_b += 10;"
        return super().__call__(*args)
```

class xentica.core.color_effects.**MovingAverage**(*func*)

Bases: *xentica.core.color_effects.ColorEffect*

Apply the moving average to each color channel separately.

With this effect, 3 additional settings are available for you in `Experiment` classes:

fade_in The maximum delta by which a channel could *increase* its value in a single timestep.

fade_out The maximum delta by which a channel could *decrease* its value in a single timestep.

smooth_factor The divisor for two previous settings, to make the effect even smoother.

2.1.8 Renderers (xentica.core.renderers)

The collection of classes implementing render logic.

The renderer takes the array of cells' colors and renders the screen frame from it. Also, it is possible to expand a list of user actions, adding ones specific to the renderer, like zoom, scroll, etc.

The default renderer is *RendererPlain*. Though there are no other renderers yet, you may try to implement your own and apply it to CA model as follows:

```
from xentica.core import CellularAutomaton
from xentica.core.renderers import Renderer

class MyRenderer(Renderer):
    # ...

class MyCA(CellularAutomaton):
    renderer = MyRenderer()
    # ...
```

class xentica.core.renderers.**Renderer**

Bases: *xentica.core.mixins.BscaDetectorMixin*

Base class for all renderers.

For correct behavior, renderer classes should be inherited from this class. Then at least *render_code()* method should be implemented.

However, if you are planning to add user actions specific to your renderer, more methods should be overridden:

- `__init__()`, where you expand a list of kernel arguments in `self.args`;
- *get_args_vals()*, where you expand the list of arguments' values;
- *setup_actions()*, where you expand a dictionary of bridge actions;

See *RendererPlain* code as an example.

get_args_vals(*bsca*)

Get a list of kernel arguments' values.

The order should correspond to `self.args`, with the values themselves as either PyCUDA `GpuArray` or correct NumPy instance. Those values will be used directly as arguments to PyCUDA kernel execution.

Parameters **bsca** – `xentica.core.CellularAutomaton` instance.

render_code()

Generate C code for rendering.

At a minimum, it should process cells colors stored in `col` GPU-array, and store the resulting pixel's value into `img` GPU-array. It can additionally use other custom arguments if any of them set up.

setup_actions(*bridge*)

Expand bridge with custom user actions.

You can do it as follows:

```
class MyRenderer(Renderer):
    # ...

    @staticmethod
    def my_awesome_action():
        def func(ca, gui):
            # do something with ``ca`` and ``gui``
        return func

    def setup_actions(self):
```

(continues on next page)

(continued from previous page)

```
        bridge.key_actions.update({
            "some_key": self.my_awesome_action(),
        })
```

Parameters **bridge** – `xentica.bridge.Bridge` instance.

class `xentica.core.renderers.`**RendererPlain**(*projection_axes=None*)
Bases: *xentica.core.renderers.Renderer*

Render board as 2D plain.

If your model has more than 2 dimensions, a projection over `projection_axes` tuple will be made. The default is two first axes, which corresponds to `(0, 1)` tuple.

static apply_move(*bsca*, **args*)
Apply a move action to CA class.

Parameters **bsca** – `xentica.core.CellularAutomaton` instance.

static apply_zoom(*bsca*, *dval*)
Apply a zoom action to CA class.

Parameters

- **bsca** – `xentica.core.CellularAutomaton` instance.
- **dval** – Delta by which the field is zoomed.

get_args_vals(*bsca*)
Extend kernel arguments values.

static move(*delta_x*, *delta_y*)
Move over a game field by some delta.

Parameters

- **dx** – Delta by x-axis.
- **dy** – Delta by y-axis.

render_code()
Implement the code for the render kernel.

setup_actions(*bridge*)
Extend the bridge with the scroll and zoom user actions.

static zoom(*dzoom*)
Zoom a game field by some delta.

Parameters **dzoom** – Delta by which field is zoomed.

2.1.9 Exceptions (xentica.core.exceptions)

The collection of exceptions, specific to the framework.

exception `xentica.core.exceptions.`**XenticaException**
Bases: `Exception`

The basic Xentica framework exception.

2.1.10 Mixins (xentica.core.mixins)

The collection of mixins to be used in core classes.

Would be interesting only if you are planning to hack into Xentica core functionality.

class xentica.core.mixins.**BscaDetectorMixin**
Bases: object

Add a functionality to detect CellularAutomaton class instances holding current class.

bsca
Get a CellularAutomaton instance holding current class.

The objects tree is scanned up to the top and the first instance found is returned.

class xentica.core.mixins.**DimensionsMixin**
Bases: object

The base functionality for classes, operating on a number of dimensions.

Adds dimensions property to a class, and checks it automatically over a list of allowed dimensions.

allowed_dimension(*num_dim*)
Test if a particular dimensionality is allowed.

Parameters **num_dim** – The number of dimensions to test.

Returns Boolean value, either dimensionality is allowed or not.

dimensions
Get a number of dimensions.

supported_dimensions = []
A list of integers, containing a supported dimensionality. You must set it manually for every class using *DimensionsMixin*.

2.2 The Topology (xentica.core.topology)

This package helps you build the topology for CA models.

All xentica.core.CellularAutomaton subclasses **must** have Topology class declared inside. This class describes:

- dimensions: the number of dimensions your CA model operates on.
- lattice: the type of lattice of your CA board. Built-in lattice types are available in *xentica.core.topology.lattice* module.
- neighborhood: the type of neighborhood for a single cell. Built-in neighborhood types are available in *xentica.core.topology.neighborhood* module.
- border: the type of border effect, e.g. how to process off-board cells. Built-in border types are available in *xentica.core.topology.border* module.

For example, you can declare the topology for a 2-dimensional orthogonal lattice with Moore neighborhood wrapped to a 3-torus, as follows:

```
class Topology:

    dimensions = 2
```

(continues on next page)

(continued from previous page)

```
    lattice = core.OrthogonalLattice()
    neighborhood = core.MooreNeighborhood()
    border = core.TorusBorder()
```

2.2.1 Lattice (xentica.core.topology.lattice)

The collection of classes describing different lattice topologies.

All classes there are intended for use inside `Topology` for `lattice` class variable definition. They are also available via *xentica.core* shortcut. The example:

```
from xentica.core import CellularAutomaton, OrthogonalLattice

class MyCA(CellularAutomaton):
    class Topology:
        lattice = OrthogonalLattice()
        # ...
    # ...
```

class xentica.core.topology.lattice.**Lattice**

Bases: *xentica.core.mixins.DimensionsMixin*, *xentica.core.mixins.BscaDetectorMixin*

The base class for all lattices.

For correct behavior, lattice classes should be inherited from this class. You should also implement the following functions:

- *index_to_coord_code()*
- *index_to_coord()*
- *coord_to_index_code()*
- *is_off_board_code()*

See the detailed description below.

coord_to_index_code(*coord_prefix*)

Generate C code for obtaining the cell's index by coordinates.

This is an abstract method, you must implement it in *Lattice* subclasses.

Parameters **coord_prefix** – The prefix for variables, containing coordinates.

Returns A string with the C code calculating cell's index. No assignment, only a valid expression needed.

index_to_coord(*idx*, *bsca*)

Obtain the cell's coordinates by its index, in pure Python.

This is an abstract method, you must implement it in *Lattice* subclasses.

Parameters

- **idx** – Cell's index, a positive integer, or a NumPy array of indices.
- **bsca** – `xentica.core.CellularAutomaton` instance, to access the field's size and number of dimensions.

Returns Tuple of integer coordinates, or NumPy arrays of coords per each axis.

index_to_coord_code(*index_name*, *coord_prefix*)
Generate C code to obtain coordinates by the cell's index.

This is an abstract method, you must implement it in *Lattice* subclasses.

Parameters

- **index_name** – The name of a variable containing the cell's index.
- **coord_prefix** – The prefix for resulting variables, containing coordinates.

Returns A string with the C code, doing all necessary to process the index variable and store coordinates to variables with the given prefix.

is_off_board_code(*coord_prefix*)
Generate C code to test if the cell's coordinates are off board.

This is an abstract method, you must implement it in *Lattice* subclasses.

Parameters **coord_prefix** – The prefix for variables, containing coordinates.

Returns A string with the C code testing coordinates' variables. No assignment, only a valid expression with boolean result needed.

width_prefix = '_w'
The prefix to be used in C code for field size constants.

class xentica.core.topology.lattice.**OrthogonalLattice**
Bases: *xentica.core.topology.lattice.Lattice*

N-dimensional orthogonal lattice.

Points are all possible positive integer coordinates.

coord_to_index_code(*coord_prefix*)
Generate C code for obtaining the cell's index by coordinates.

See *Lattice.coord_to_index_code()* for details.

index_to_coord(*idx*, *bsca*)
Obtain the cell's coordinates by its index, in pure Python.

See *Lattice.index_to_coord()* for details.

index_to_coord_code(*index_name*, *coord_prefix*)
Generate C code for obtaining the cell's index by coordinates.

See *Lattice.index_to_coord_code()* for details.

is_off_board_code(*coord_prefix*)
Generate C code to test if the cell's coordinates are off board.

See *Lattice.is_off_board_code()* for details.

supported_dimensions = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
Overridden value for supported dimensions.

2.2.2 Neighborhood (xentica.core.topology.neighborhood)

The collection of classes describing different neighborhood topologies.

All classes there are intended for use inside `Topology` for `neighborhood` class variable definition. They are also available via *xentica.core* shortcut. The example:

```
from xentica.core import CellularAutomaton, MooreNeighborhood

class MyCA(CellularAutomaton):
    class Topology:
        neighborhood = MooreNeighborhood()
        # ...
    # ...
```

class xentica.core.topology.neighborhood.**Neighborhood**

Bases: *xentica.core.mixins.DimensionsMixin*

The base class for all neighborhood topologies.

For correct behavior, neighborhood classes should be inherited from this class. You should also implement the following functions:

- *neighbor_coords()*
- *neighbor_state()*

See the detailed description below.

neighbor_coords(*index*, *coord_prefix*, *neighbor_prefix*)

Generate the C code to obtain neighbor coordinates by its index.

This is an abstract method, you must implement it in *Neighborhood* subclasses.

Parameters

- **index** – Neighbor's index, a non-negative integer less than the number of neighbors.
- **coord_prefix** – The prefix for variables containing main cell's coordinates.
- **neighbor_prefix** – The prefix for resulting variables containing neighbor coordinates.

Returns A string with the C code doing all necessary to get neighbor's state from the RAM. No assignment, only a valid expression needed.

neighbor_state(*neighbor_index*, *state_index*, *coord_prefix*)

Generate the C code to obtain a neighbor's state by its index.

This is an abstract method, you must implement it in *Neighborhood* subclasses.

Parameters

- **neighbor_index** – Neighbor's index, a non-negative integer less than the number of neighbors.
- **state_index** – State's index, a non-negative integer less than the number of neighbors for buffered states or -1 for main state.
- **coord_prefix** – The prefix for variables containing neighbor coordinates.

Returns A string with the C code doing all necessary to process neighbor's coordinates and store them to neighbor's coordinates variables.

num_neighbors = None

Number of neighbors, you must re-define it in sub-classes.

topology = None

A reference to `Topology` holder class, will be set in `BSCA` metaclass.

class xentica.core.topology.neighborhood.**OrthogonalNeighborhood**
Bases: *xentica.core.topology.neighborhood.Neighborhood*

The base class for neighborhoods on an orthogonal lattice.

It is implementing all necessary *Neighborhood* abstract methods, the only thing you should override is `dimensions()` setter. In `dimensions()`, you should correctly set `num_neighbors` and `_neighbor_deltas` attributes.

neighbor_coords(*index*, *coord_prefix*, *neighbor_prefix*)
Generate the C code to obtain neighbor coordinates by its index.

See *Neighborhood.neighbor_coords()* for details.

neighbor_state(*neighbor_index*, *state_index*, *coord_prefix*)
Generate the C code to obtain a neighbor's state by its index.

See *Neighborhood.neighbor_coords()* for details.

supported_dimensions = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
Any number of dimentions is supported, 100 is just to limit your hyperspatial hunger.

class xentica.core.topology.neighborhood.**MooreNeighborhood**
Bases: *xentica.core.topology.neighborhood.OrthogonalNeighborhood*

N-dimensional Moore neighborhood implementation.

The neighbors are all cells, sharing at least one vertex.

dimensions
Get a number of dimensions.

class xentica.core.topology.neighborhood.**VonNeumannNeighborhood**
Bases: *xentica.core.topology.neighborhood.OrthogonalNeighborhood*

N-dimensional Von Neumann neighborhood implementation.

The neighbors are adjacent cells in all possible orthogonal directions.

dimensions
Get a number of dimensions.

2.2.3 Border (xentica.core.topology.border)

The collection of classes describing different types of field's borders.

All classes there are intended for use inside `Topology` for `border` class variable definition. They are also available via *xentica.core* shortcut. The example:

```
from xentica.core import CellularAutomaton, TorusBorder

class MyCA(CellularAutomaton):
    class Topology:
        border = TorusBorder()
        # ...
    # ...
```

class xentica.core.topology.border.**Border**
Bases: *xentica.core.mixins.DimensionsMixin*

The base class for all types of borders.

You should not inherit your borders directly from this class, use either *WrappedBorder* or *GeneratedBorder* base subclasses instead.

class xentica.core.topology.border.**WrappedBorder**
Bases: *xentica.core.topology.border.Border*

The base class for borders that wraps the field into different manifolds.

For correct behavior, you should implement *wrap_coords()* method.

See the detailed description below.

wrap_coords(*coord_prefix*)
Generate C code to translate off-board coordinates to on-board ones.

This is an abstract method, you must implement it in *WrappedBorder* subclasses.

Parameters **coord_prefix** – The prefix for variables containing the cell's coordinates.

class xentica.core.topology.border.**GeneratedBorder**
Bases: *xentica.core.topology.border.Border*

The base class for borders that generates states of the off-board cells.

For correct behavior, you should implement *off_board_state()* method.

See the detailed description below.

off_board_state(*coord_prefix*)
Generate C code to obtain off-board cell's state.

This is an abstract method, you must implement it in *GeneratedBorder* subclasses.

Parameters **coord_prefix** – The prefix for variables containing the cell's coordinates.

class xentica.core.topology.border.**TorusBorder**
Bases: *xentica.core.topology.border.WrappedBorder*

Wraps the entire field into N-torus manifold.

This is the most common type of border, allowing you to generate seamless tiles for wallpapers.

supported_dimensions = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
Any number of dimentions is supported, 100 is just to limit your hyperspatial hunger.

wrap_coords(*coord_prefix*)
Implement coordinates wrapping to torus.

See *WrappedBorder.wrap_coords()* for details.

class xentica.core.topology.border.**StaticBorder**(*value=0*)
Bases: *xentica.core.topology.border.GeneratedBorder*

Generates a static value for every off-board cell.

This is acting like your field is surrounded by cells with the same pre-defined state.

The default is just an empty (zero) state.

off_board_state(*coord_prefix*)
Impement off-board cells' values obtaining.

See *GeneratedBorder.off_board_state()* for details.

supported_dimensions = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,

2.3 Tools (xentica.tools)

The package with different tools serving as helpers in other modules.

xentica.tools.**xmath**
alias of *xentica.tools.xmath.Xmath*

2.3.1 Color Manipulation Functions (xentica.tools.color)

The collection of color conversion helpers.

class xentica.tools.color.**GenomeColor**
Bases: object

Different approaches for rendering the genome's color.

static modular(*genome*, *divider*)
Convert genome bit value to RGB color using modular division.

This algorithm simply divides the genome value by some modulo, normalize it and use as a hue with maximum saturation/value.

As a result, you could check by eye how genomes behave inside each group, since similar genomes most likely will have distinctive colors.

Parameters

- **genome** – Genome as integer (bit) sequence.
- **divider** – Divider for the modular division.

Returns tuple (red, green, blue)

static positional(*genome*, *num_genes*)
Convert genome bit value to RGB color using ones' positions.

This algorithm treats positions of '1' in binary genome representation as a hue with maximum saturation/value (as in HSV model), then blends them together to produce the final RGB color. Genome length (second argument) is essential to calculate genes' positions in [0, 1] range.

As a result, two genomes will look similar visually if they have a little difference in genes. That could help in the quick detection of genome groups by eye.

Parameters

- **genome** – Genome as integer (bit) sequence.
- **num_genes** – Genome length in bits.

Returns tuple (red, green, blue)

xentica.tools.color.**hsv2rgb**(*hue*, *saturation*, *value*)
Convert HSV color to RGB format.

Parameters

- **hue** – Hue value [0, 1]
- **saturation** – Saturation value [0, 1]
- **value** – Brightness value [0, 1]

Returns tuple (red, green, blue)

2.3.2 Genetics Functions (xentica.tools.genetics)

The collection of functions for genetics manipulations.

xentica.tools.genetics.**genome_crossover**(*state*, *num_genes*, **genomes*, *max_genes=None*, *mutation_prob=0*, *rng_name='rng'*)

Crossover given genomes in stochastic way.

Parameters

- **state** – A container holding model's properties.
- **num_genes** – Genome length, assuming all genomes has the same number of genes.
- **genomes** – A list of genomes (integers) to crossover
- **max_genes** – Upper limit for '1' genes in the resulting genome.
- **mutation_prob** – Probability of a single gene's mutation.
- **rng_name** – Name of `RandomProperty`.

Returns Single integer, a resulting genome.

2.3.3 CA Rules Functions (xentica.tools.rules)

The module with different helpers for CA rules.

class xentica.tools.rules.**LifeLike**

Bases: `object`

Life-like rules helpers.

static golly2int(*golly_str*)

Convert a string in the Golly format to inner rule representation.

Parameters **golly_str** – Rule in the Golly format (e.g. B3/S23)

Returns Integer representation of the rule for inner use.

static int2golly(*rule*)

Convert inner rule representation to string in the Golly format.

Parameters **rule** – Integer representation of the rule.

Returns Golly-formatted rule (e.g. B3/S23).

2.3.4 Math Functions (xentica.tools.xmath)

The module with bindings to CUDA math functions.

class xentica.tools.xmath.**Xmath**

Bases: `object`

Static class holding all math functions.

static exp(*val*)

Calculate exponent.

static float(*val*)

Cast a value to float.

static int(*val*)
Cast a value to int.

static max(**args*)
Calculate the maximum over list of args.

static min(**args*)
Calculate the minimum over list of args.

static popc(*val*)
Count the number of bits that are set to '1' in a 32 bit integer.

2.4 The Seeds (xentica.seeds)

The package for the initial CA state (seed) generation.

Classes from modules below are intended for use in `Experiment` classes.

For example, to initialize the whole board with random values:

```
from xentica import core, seeds

class MyExperiment(core.Experiment):
    # ...
    seed = seeds.patterns.PrimordialSoup(
        vals={
            "state": seeds.random.RandInt(0, 1),
        }
   )
```

2.4.1 Patterns (xentica.seeds.patterns)

The module containing different patterns for CA seed initialization.

Each pattern class have one mandatory method `generate()` which is called automatically at the initialization stage.

Patterns are intended for use in `Experiment` classes. See the example of general usage above.

class xentica.seeds.patterns.**RandomPattern**(*vals*)
Bases: `object`

The base class for random patterns.

Parameters **vals** – Dictionary with mixed values. May contain descriptor classes.

generate(*cells*, *bsca*)
Generate the entire initial state.

This is an abstract method, you must implement it in *RandomPattern* subclasses.

Parameters

- **cells** – NumPy array with cells' states as items. The seed will be generated over this array.
- **bsca** – `xentica.core.CellularAutomaton` instance, to access the field's size and other attributes.

random
Get the random stream.

class xentica.seeds.patterns.**BigBang**(*vals*, *pos=None*, *size=None*)
Bases: *xentica.seeds.patterns.RandomPattern*

Random init pattern, known as *"Big Bang"*.

Citation from The Concept:

> *"A small area of space is initialized with a high amount of energy and random parameters per each quantum. Outside the area, quanta has either zero or minimum possible amount of energy. This is a good test for the ability of energy to spread in empty space."*

The current implementation generates a value for every cell inside a specified N-cube area. Cells outside the area remain unchanged.

Parameters

- **vals** – Dictionary with mixed values. May contain descriptor classes.
- **pos** – A tuple with the coordinates of the lowest corner of the Bang area.
- **size** – A tuple with the size of the Bang area per each dimension.

generate(*cells*, *bsca*)
Generate the entire initial state.

See *RandomPattern.generate()* for details.

class xentica.seeds.patterns.**PrimordialSoup**(*vals*)
Bases: *xentica.seeds.patterns.RandomPattern*

Random init pattern, known as *"Primordial Soup"*.

Citation from The Concept:

> *"Each and every quantum initially has an equally small amount of energy, other parameters are random. This is a good test for the ability of energy to self-organize in clusters from the completely uniform distribution."*

The current implementation populates the entire board with generated values.

Parameters **vals** – Dictionary with mixed values. May contain descriptor classes.

generate(*cells*, *bsca*)
Generate the entire initial state.

See *RandomPattern.generate()* for details.

class xentica.seeds.patterns.**ValDict**(*dct*, *parent=None*)
Bases: object

A wrapper over the Python dictionary.

It can keep descriptor classes along with regular values. When you get the item, the necessary value is automatically obtaining either directly or via descriptor logic.

Read-only, you should set all dictionary values at the class initialization.

The example of usage:

```
>>> from xentica.seeds.random import RandInt
>>> from xentica.seeds.patterns import ValDict
>>> d = {'a': 2, 's': RandInt(11, 23), 'd': 3.3}
>>> vd = ValDict(d)
>>> vd['a']
2
```

(continues on next page)

(continued from previous page)

```
>>> vd['s']
14
>>> vd['d']
3.3
```

Parameters

- **dct** – Dictionary with mixed values. May contain descriptor classes.
- **parent** – A reference to the class holding the dictionary. Optional.

items()
Iterate over dictionary items.

keys()
Iterate over dictionary keys.

2.4.2 RNG (xentica.seeds.random)

The module for package-wide RNG.

The main intention is to keep separate deterministic random streams for every *CellularAutomaton* instance. So, if you've initialized RNG for a particular CA with some seed, you're geting the guarantee that the random sequence will be the same, no matter how many other CA's you're running in parallel.

class xentica.seeds.random.**LocalRandom**(*seed=None*)
Bases: object

The holder class for the RNG sequence.

It is encapsulating both standard Python random stream and NumPy one.

Once instantiated, you can use them as follows:

```
from xentica.seeds.random import LocalRandom

random = LocalRandom()
# get random number from standard stream
val = random.standard.randint(1, 10)
# get 100 random numbers from NumPy stream
vals = random.numpy.randint(1, 10, 100)
```

load(*rng*)
Load a random state from the class.

Parameters **rng** – *LocalRandom* instance.

class xentica.seeds.random.**RandInt**(*min_val*, *max_val*, *constant=False*)
Bases: xentica.core.expressions.PatternExpression

Class, generating a sequence of random integers in some interval.

It is intended for use in Experiment seeds. See the example of initializing CA property above.

Parameters

- **min_val** – Lower bound for a random value.
- **max_val** – Upper bound for a random value.

- **constant** – If `True`, will force the use of the standard random stream.

2.5 The Bridge (xentica.bridge)

The bridge between Xentica and GUI interface.

This package contains all necessary stuff to connect Xentica framework to custom interactive visualization environments.

Right now, only one environment (Moire) is available. This is the official environment, evolving along with the main framework. You are free to implement your own environments. If so, please make a PR on Github and we'll include your solution to the bridge.

Bridge functions are automatically used when you run the simulation like this:

```
import moire
ca = MyCellularAutomaton(MyExperiment)
gui = moire.GUI(runnable=ca)
gui.run()
```

2.5.1 Base (xentica.bridge.base)

This module contains the main class to be used in custom bridges.

Methods from *Bridge* class should be used in other bridges

class xentica.bridge.base.**Bridge**
Bases: object

Main bridge class containing basic functions.

static exit_app(*_model*, *gui*)
Exit GUI application.

static noop(*_model*, *_gui*)
Do nothing.

static speed(*dspeed*)
Change simulation speed.

static toggle_pause(*model*, *_gui*)
Pause/unpause simulation.

static toggle_sysinfo(*_model*, *gui*)
Turn system info panel on/off.

2.5.2 Moire (xentica.bridge.moire)

Module with the bridge to Moire UI.

class xentica.bridge.moire.**MoireBridge**
Class incaplulating the actions for Moire UI.

`[` Speed simulation down.

`]` Speed simulation up.

`SPACEBAR` Pause/unpause simulation.

`F12` Toggle system info.

`ESC` Exit app.

2.6 Utilities (xentica.utils)

The collection of utilities.

2.6.1 Formatters (xentica.utils.formatters)

The collection of formatters.

`xentica.utils.formatters.`**`sizeof_fmt`**(*num*, *suffix=''*)

Format the number to a humanized order of magnitude.

For example, `11234` become `11.23K`.

Parameters

- **`num`** – The positive integer.
- **`suffix`** – Additional suffix added to the resulting string.

Returns Formatted number as a string.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

x

xentica.bridge, 37
xentica.bridge.base, 37
xentica.bridge.moire, 37
xentica.core, 13
xentica.core.base, 14
xentica.core.color_effects, 22
xentica.core.exceptions, 25
xentica.core.experiment, 16
xentica.core.expressions, 22
xentica.core.mixins, 26
xentica.core.parameters, 20
xentica.core.properties, 17
xentica.core.renderers, 23
xentica.core.topology, 26
xentica.core.topology.border, 30
xentica.core.topology.lattice, 27
xentica.core.topology.neighborhood, 28
xentica.core.variables, 21
xentica.seeds, 34
xentica.seeds.patterns, 34
xentica.seeds.random, 36
xentica.tools, 32
xentica.tools.color, 32
xentica.tools.genetics, 33
xentica.tools.rules, 33
xentica.tools.xmath, 33
xentica.utils, 38
xentica.utils.formatters, 38

Index

A

allowed_dimension() (*xentica.core.mixins.DimensionsMixin method*), 26
apply_move() (*xentica.core.renderers.RendererPlain static method*), 25
apply_speed() (*xentica.core.base.CellularAutomaton method*), 16
apply_zoom() (*xentica.core.renderers.RendererPlain static method*), 25

B

best_type (*xentica.core.properties.Property attribute*), 18
BigBang (*class in xentica.seeds.patterns*), 34
bit_width (*xentica.core.properties.Property attribute*), 18
Border (*class in xentica.core.topology.border*), 30
Bridge (*class in xentica.bridge.base*), 37
BSCA (*class in xentica.core.base*), 15
bsca (*xentica.core.mixins.BscaDetectorMixin attribute*), 26
BscaDetectorMixin (*class in xentica.core.mixins*), 26
buf_num (*xentica.core.properties.ContainerProperty attribute*), 19
buf_num (*xentica.core.properties.Property attribute*), 18

C

CachedNeighbor (*class in xentica.core.base*), 16
calc_bit_width() (*xentica.core.properties.ContainerProperty method*), 19
calc_bit_width() (*xentica.core.properties.IntegerProperty method*), 19
calc_bit_width() (*xentica.core.properties.Property method*), 18
CellularAutomaton (*class in xentica.core.base*), 15
code (*xentica.core.variables.Variable attribute*), 22
ColorEffect (*class in xentica.core.color_effects*), 23
Constant (*class in xentica.core.variables*), 21
ContainerProperty (*class in xentica.core.properties*), 19
coord_to_index_code() (*xentica.core.topology.lattice.Lattice method*), 27
coord_to_index_code() (*xentica.core.topology.lattice.OrthogonalLattice method*), 28
coords_declared (*xentica.core.properties.Property attribute*), 18
ctype (*xentica.core.parameters.Parameter attribute*), 20
ctype (*xentica.core.properties.Property attribute*), 18

D

declare_once() (*xentica.core.properties.ContainerProperty method*), 19
declare_once() (*xentica.core.properties.Property method*), 18
declare_once() (*xentica.core.variables.Variable method*), 22
declared (*xentica.core.properties.Property attribute*), 18
deferred_write() (*xentica.core.properties.ContainerProperty method*), 19
DeferredExpression (*class in xentica.core.expressions*), 22
dimensions (*xentica.core.mixins.DimensionsMixin attribute*), 26
dimensions (*xentica.core.topology.neighborhood.MooreNeighborhood attribute*), 30
dimensions (*xentica.core.topology.neighborhood.VonNeumannNeighbor attribute*), 30
DimensionsMixin (*class in xentica.core.mixins*), 26

dtype (*xentica.core.parameters.Parameter attribute*), 21
dtype (*xentica.core.properties.Property attribute*), 18

E

exit_app() (*xentica.bridge.base.Bridge static method*), 37
exp() (*xentica.tools.xmath.Xmath static method*), 33
Experiment (*class in xentica.core.experiment*), 17

F

float() (*xentica.tools.xmath.Xmath static method*), 33
FloatVariable (*class in xentica.core.variables*), 22

G

generate() (*xentica.seeds.patterns.BigBang method*), 35
generate() (*xentica.seeds.patterns.PrimordialSoup method*), 35
generate() (*xentica.seeds.patterns.RandomPattern method*), 34
GeneratedBorder (*class in xentica.core.topology.border*), 31
genome_crossover() (*in module xentica.tools.genetics*), 33
GenomeColor (*class in xentica.tools.color*), 32
get_args_vals() (*xentica.core.renderers.Renderer method*), 24
get_args_vals() (*xentica.core.renderers.RendererPlain method*), 25
get_define_code() (*xentica.core.variables.Constant method*), 21
golly2int() (*xentica.tools.rules.LifeLike static method*), 33

H

hsv2rgb() (*in module xentica.tools.color*), 32

I

index_to_coord() (*xentica.core.topology.lattice.Lattice method*), 27
index_to_coord() (*xentica.core.topology.lattice.OrthogonalLattice method*), 28
index_to_coord_code() (*xentica.core.topology.lattice.Lattice method*), 27
index_to_coord_code() (*xentica.core.topology.lattice.OrthogonalLattice method*), 28
int() (*xentica.tools.xmath.Xmath static method*), 33
int2golly() (*xentica.tools.rules.LifeLike static method*), 33
IntegerProperty (*class in xentica.core.properties*), 19
IntegerVariable (*class in xentica.core.variables*), 22
is_off_board_code() (*xentica.core.topology.lattice.Lattice method*), 28
is_off_board_code() (*xentica.core.topology.lattice.OrthogonalLattice method*), 28
items() (*xentica.seeds.patterns.ValDict method*), 36

K

keys() (*xentica.seeds.patterns.ValDict method*), 36

L

Lattice (*class in xentica.core.topology.lattice*), 27
LifeLike (*class in xentica.tools.rules*), 33
load() (*xentica.core.base.CellularAutomaton method*), 16
load() (*xentica.seeds.random.LocalRandom method*), 36
LocalRandom (*class in xentica.seeds.random*), 36

M

mandatory_fields (*xentica.core.base.BSCA attribute*), 15
max() (*xentica.tools.xmath.Xmath static method*), 34
min() (*xentica.tools.xmath.Xmath static method*), 34
modular() (*xentica.tools.color.GenomeColor static method*), 32
MoireBridge (*class in xentica.bridge.moire*), 37
MooreNeighborhood (*class in xentica.core.topology.neighborhood*), 30
move() (*xentica.core.renderers.RendererPlain static method*), 25
MovingAverage (*class in xentica.core.color_effects*), 23

N

name (*xentica.core.parameters.Parameter attribute*), 21
name (*xentica.core.variables.Constant attribute*), 21
nbr_num (*xentica.core.properties.ContainerProperty attribute*), 19
nbr_num (*xentica.core.properties.Property attribute*), 18
neighbor_coords() (*xentica.core.topology.neighborhood.Neighborhood method*), 29
neighbor_coords() (*xentica.core.topology.neighborhood.OrthogonalNeighborhood method*), 30

neighbor_state() (*xentica.core.topology.neighborhood.Neighborhood method*), 29
neighbor_state() (*xentica.core.topology.neighborhood.OrthogonalNeighborhood method*), 30
Neighborhood (*class in xentica.core.topology.neighborhood*), 29
noop() (*xentica.bridge.base.Bridge static method*), 37
num_neighbors (*xentica.core.topology.neighborhood.Neighborhood attribute*), 29

O

off_board_state() (*xentica.core.topology.border.GeneratedBorder method*), 31
off_board_state() (*xentica.core.topology.border.StaticBorder method*), 31
OrthogonalLattice (*class in xentica.core.topology.lattice*), 28
OrthogonalNeighborhood (*class in xentica.core.topology.neighborhood*), 29

P

pack_cast() (*xentica.core.properties.Property method*), 18
Parameter (*class in xentica.core.parameters*), 20
popc() (*xentica.tools.xmath.Xmath static method*), 34
positional() (*xentica.tools.color.GenomeColor static method*), 32
PrimordialSoup (*class in xentica.seeds.patterns*), 35
properties (*xentica.core.properties.ContainerProperty attribute*), 19
Property (*class in xentica.core.properties*), 17

R

RandInt (*class in xentica.seeds.random*), 36
random (*xentica.seeds.patterns.RandomPattern attribute*), 34
RandomPattern (*class in xentica.seeds.patterns*), 34
render() (*xentica.core.base.CellularAutomaton method*), 16
render_code() (*xentica.core.renderers.Renderer method*), 24
render_code() (*xentica.core.renderers.RendererPlain method*), 25
Renderer (*class in xentica.core.renderers*), 24
RendererPlain (*class in xentica.core.renderers*), 25

S

save() (*xentica.core.base.CellularAutomaton method*), 16
set_viewport() (*xentica.core.base.CellularAutomaton method*), 16
setup_actions() (*xentica.core.renderers.Renderer method*), 24
setup_actions() (*xentica.core.renderers.RendererPlain method*), 25
sizeof_fmt() (*in module xentica.utils.formatters*), 38
speed() (*xentica.bridge.base.Bridge static method*), 37
StaticBorder (*class in xentica.core.topology.border*), 31
step() (*xentica.core.base.CellularAutomaton method*), 16
supported_dimensions (*xentica.core.mixins.DimensionsMixin attribute*), 26
supported_dimensions (*xentica.core.topology.border.StaticBorder attribute*), 31
supported_dimensions (*xentica.core.topology.border.TorusBorder attribute*), 31
supported_dimensions (*xentica.core.topology.lattice.OrthogonalLattice attribute*), 28
supported_dimensions (*xentica.core.topology.neighborhood.OrthogonalNeighborhood attribute*), 30

T

toggle_pause() (*xentica.bridge.base.Bridge static method*), 37
toggle_pause() (*xentica.core.base.CellularAutomaton method*), 16
toggle_sysinfo() (*xentica.bridge.base.Bridge static method*), 37
topology (*xentica.core.topology.neighborhood.Neighborhood attribute*), 29
TorusBorder (*class in xentica.core.topology.border*), 31

U

unpack_cast() (*xentica.core.properties.Property method*), 18
unpacked (*xentica.core.properties.ContainerProperty attribute*), 19

V

ValDict (*class in xentica.seeds.patterns*), 35
value (*xentica.core.parameters.Parameter attribute*), 21

values() (*xentica.core.properties.ContainerProperty method*), 19
var_name (*xentica.core.variables.Variable attribute*), 22
var_type (*xentica.core.variables.FloatVariable attribute*), 22
var_type (*xentica.core.variables.IntegerVariable attribute*), 22
Variable (*class in xentica.core.variables*), 21
VonNeumannNeighborhood (*class in xentica.core.topology.neighborhood*), 30

W

width (*xentica.core.properties.Property attribute*), 18
width_prefix (*xentica.core.topology.lattice.Lattice attribute*), 28
wrap_coords() (*xentica.core.topology.border.TorusBorder method*), 31
wrap_coords() (*xentica.core.topology.border.WrappedBorder method*), 31
WrappedBorder (*class in xentica.core.topology.border*), 31

X

xentica.bridge (*module*), 37
xentica.bridge.base (*module*), 37
xentica.bridge.moire (*module*), 37
xentica.core (*module*), 13
xentica.core.base (*module*), 14
xentica.core.color_effects (*module*), 22
xentica.core.exceptions (*module*), 25
xentica.core.experiment (*module*), 16
xentica.core.expressions (*module*), 22
xentica.core.mixins (*module*), 26
xentica.core.parameters (*module*), 20
xentica.core.properties (*module*), 17
xentica.core.renderers (*module*), 23
xentica.core.topology (*module*), 26
xentica.core.topology.border (*module*), 30
xentica.core.topology.lattice (*module*), 27
xentica.core.topology.neighborhood (*module*), 28
xentica.core.variables (*module*), 21
xentica.seeds (*module*), 34
xentica.seeds.patterns (*module*), 34
xentica.seeds.random (*module*), 36
xentica.tools (*module*), 32
xentica.tools.color (*module*), 32
xentica.tools.genetics (*module*), 33
xentica.tools.rules (*module*), 33
xentica.tools.xmath (*module*), 33
xentica.utils (*module*), 38
xentica.utils.formatters (*module*), 38
XenticaException, 25
Xmath (*class in xentica.tools.xmath*), 33
xmath (*in module xentica.tools*), 32

Z

zoom() (*xentica.core.renderers.RendererPlain static method*), 25