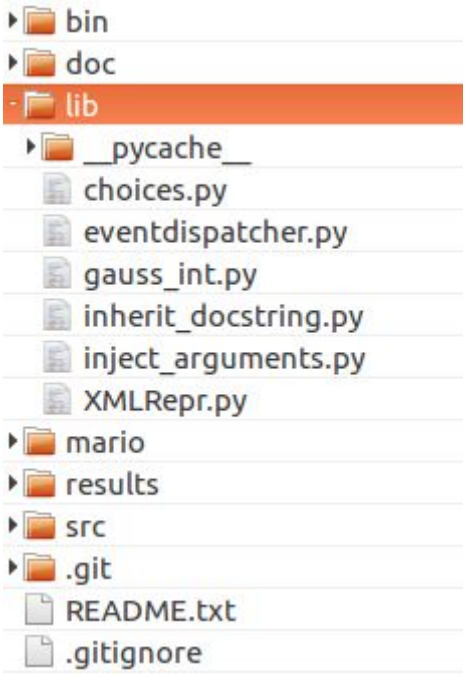
```python
#!/usr/bin/python3.4
# -*-coding:Utf-8 -*

from argparse import ArgumentParser
from math import inf
from time import time

from lib.eventdispatcher import EventDispatcher
from mario.bridge.frame_reader import FrameReader
from mario.bridge.config import Config
from .EvolutiveGenerator.Generator import Generator
from .factories.IAFactory import IAFactory
from .graduators.IAGraduator import IAGraduator
from .graduators.GameOptimizer import GameOptimizer
from .Writer.Writer import Writer
from .Logger.FileLogger import FileLogger
from .Logger.ConsoleLogger import ConsoleLogger
from .Writer.PathManager import PathManager
from .Writer.Reader import Reader


def instanciateGenerator(show):
    event_dispatcher = EventDispatcher()
    FrameReader(event_dispatcher)
    GameOptimizer(event_dispatcher)
    return Generator(IAFactory, IAGraduator(event_dispatcher, show), [Writer(), FileLogger(), ConsoleLogger()],
        lambda state: True in [score.percent >= 1. for score, individual in state.grading]
    )

def checkProcessusExists(processus_id):
    if not Reader.processusExists(processus_id):
        raise ValueError("Processus with id={} doesn't exist.".format(processus_id))

def new(args):
    """New processus"""
    population = instanciateGenerator(args.show).process(
        PathManager.newProcessusId(), args.generations, args.pop_length, args.proportion, args.chance
    )

def resume(args):
    """Resume a processus"""
    checkProcessusExists(args.processus_id)
    population = instanciateGenerator(args.show).resume(Reader.getProcessusState(args.processus_id))

def play(args):
    """Play the best individual of a processus' last generation  """
    checkProcessusExists(args.processus_id)
    # Get IA
    if args.ia_id is None:
        ia, generation_id = Reader.getBestIa(args.processus_id, args.generation_id)
        print('The best AI is {}.'.format(ia.id), flush=True)
    else:
        ia, generation_id = Reader.getIa(args.processus_id, args.ia_id)
    # Play IA
    event_dispatcher = EventDispatcher()
    FrameReader(event_dispatcher)
    graduator = IAGraduator(event_dispatcher, show=True)
    if args.as_grading:
        print(
            "Attention : Malgré que le visionnage présenté soit le plus proche possible des conditions d'évaluation, des aléas subsistent. "
            "Si vous cherchez à visionner une performance difficile à reproduire, n'hésitez pas à rééssayer plusieurs fois.    "
        , flush=True)
        GameOptimizer(event_dispatcher)
        graduator.grade(ia, generation_id)
    else:
        graduator.gradeIAWithConfig(ia, Config(True, event_dispatcher))

def print_data(args):
    checkProcessusExists(args.processus_id)

    data = Reader.getData(args.processus_id)
    txt1 = 'Générations,Scores des intelligences '
    for generation_id, grading in data:
        txt1 += '\n' + str(generation_id)
        for result, ia_id in grading:
            txt1 += ',' + str(result['score'])
    txt2 = 'Générations,Scores des intelligences '
    for generation_id, grading in data:
        txt2 += '\n' + str(generation_id)
        for result, ia_id in grading:
            txt2 += ',' + str(result['max_x'])

    path1 = PathManager.getPath(args.processus_id, read_only=True).parent / 'data' / (str(time()) + '.score.csv')
    path2 = PathManager.getPath(args.processus_id, read_only=True).parent / 'data' / (str(time()) + '.distance.csv')
    PathManager.makeDir(path1.parent)
    path1.write_text(txt1)
    path2.write_text(txt2)


# Build parser
parser = ArgumentParser()
subparsers = parser.add_subparsers()

new_parser = subparsers.add_parser('new')
new_parser.add_argument('pop_length', type=int)
new_parser.add_argument('--generations', default=inf, type=int)
new_parser.add_argument('--proportion', default=0.5, type=float)
new_parser.add_argument('--chance', default=0, type=float)
```

```
new_parser.add_argument('--show', dest='show', action='store_true')
new_parser.set_defaults(commano=new, show=False)

resume_parser = subparsers.add_parser('resume')
resume_parser.add_argument('processus_id', type=int)
resume_parser.add_argument('--show', dest='show', action='store_true')
resume_parser.set_defaults(command=resume, show=False)

play_parser = subparsers.add_parser('play')
play_parser.add_argument('processus_id', type=int)
play_parser.add_argument('--generation_id', type=int)
play_parser.add_argument('--ia_id', type=int)
play_parser.add_argument('--as_grading', dest='as_grading', action='store_true')
play_parser.set_defaults(command=play, as_grading=False)

print_parser = subparsers.add_parser('print')
print_parser.add_argument('processus_id', type=int)
print_parser.set_defaults(command=print_data)

# Parse arguments
args = parser.parse_args()
if hasattr(args, 'command'):
    args.command(args)
else:
    print('No command given, use --help')
```

Bibliothèques utilisées

Dossier /lib



- *EventDispatcher* (créé par moi sur d'autres projets)
- *XMLRepr* (créé par moi pour l'occasion)
- *inject_arguments* (créé par moi pour l'occasion)
- *inherit_doctring* (pris sur Internet)
- *gauss_int et choices* (créé par moi pour l'occasion)

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


# The MIT License (MIT)
#
# Copyright (c) 2015-2016 Rémi Blaise <remi.blaise@gmx.fr> "http://php-zzortell.rhcloud.com/"
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.
#


import re


class EventDispatcher:
    '''
    A simple event dispatcher

    Author: Rémi Blaise (alias Zzortell) "http://php-zzortell.rhcloud.com/"

    '''


    def __init__(self, propagation=False):
        '''
        Init the event dispatcher

        Parameter:
        {bool} propagation = False If dispatching an event should also dispatch its parents

        '''

        self.propagation = propagation
        self.listeners = {}


    def listen(self, name, listener, priority=0):
        '''
        Add an event listener

        If name is 'all', the listener will listen all events.

        Parameters:
        {str}         name             The name of the event
        {function}     listener         The event listener
        {int}        priority = 0     The priority of the listener

        Return: {tuple} id The ID of the listener

        '''

        # Register listener
        if name not in self.listeners:
            self.listeners[name] = {}
        if priority not in self.listeners[name]:
            self.listeners[name][priority] = []
        self.listeners[name][priority].append(listener)

        return (name, priority, listener)


    def on(self, name):
        '''Inscribe given listener, to use as decorator '''
        def decorator(function):
            self.listen(name, function)
            return function
        return decorator


    def detach(self, id):
        '''
        Detach an event listener

        Parameter:
        {tuple} id The ID of the listener

        '''

        name, priority, listener  = id
        self.listeners[name][priority].remove(listener)


```

```
    def dispatch(self, name, event=None, propagation=None):
        '''
        Dispatch an event

        If propagation is set, dispatch all the parent events.

        Parameters:
        {str}         name                The name of the event
        {object}     event = None         The event to dispatch
        {bool}         propagation = None     Override self.propagation

        '''

        if name == 'all':
            raise ValueError("'all' is a reserved keyword, not an event name. ")
        propagation = propagation if propagation is not None else self.propagation

        # Get existing keys among ('all', name)
        names = []
        if 'all' in self.listeners:
            names.append('all')
        if name in self.listeners:
            names.append(name)

        # Get sorted list of priorities
        priorities = set()
        for name in names:
            priorities = priorities.union(set(self.listeners[name].keys()))
        priorities = list(priorities)
        priorities.sort()

        # Iterate over priorities
        for priority in priorities:
            # Get listeners
            listeners = []
            for name in names:
                if priority in self.listeners[name]:
                    listeners.extend(self.listeners[name][priority])

            # Iterate over listeners
            for listener in listeners:
                listener(event)

        # If propagation dispatch the parent event
        if propagation:
            parent_name = self.getParent(name)
            if parent_name:
                self.dispatch(parent_name, event)


    def getParent(self, name):
        '''
        Get the name of the parent event

        Used if the propagation option is True.
        The event name has to match the format "parent.event".

        Parameters:
        {str} name The name of the event

        Return: {str}     The name of the parent event
                None     If the event has no parent

        '''

        if re.search(r'^(?:\w+\.)*\w+$', name) is None:
            raise AssertionError("The event name has to match with r'^(?:\w+\.)*\w+$'. ")

        if re.search(r'\.', name):
            return re.search(r'^((?:\w+\.)*)\w+$', name).group(1)[:-1]
        else:
            return None
```

```
  1  #!/usr/bin/env python3
  2  # -*-coding:Utf-8 -*
  3
  4  # The MIT License (MIT)
  5  #
  6  # Copyright (c) 2016 Rémi Blaise <remi.blaise@gmx.fr> "http://php-zzortell.rhcloud.com/"
  7  #
  8  # Permission is hereby granted, free of charge, to any person obtaining a copy
  9  # of this software and associated documentation files (the "Software"), to deal
 10  # in the Software without restriction, including without limitation the rights
 11  # to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 12  # copies of the Software, and to permit persons to whom the Software is
 13  # furnished to do so, subject to the following conditions:
 14  #
 15  # The above copyright notice and this permission notice shall be included in all
 16  # copies or substantial portions of the Software.
 17  #
 18  # THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 19  # IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 20  # FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 21  # AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 22  # LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 23  # OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 24  # SOFTWARE.
 25
 26
 27  from textwrap import indent
 28  from operator import itemgetter, attrgetter
 29
 30
 31  class XMLRepr:
 32      """
 33      Awesome XML representation base class
 34
 35      Inherit to have an XML-like repr of instances.
 36
 37      __repr__ expects:
 38          attributes to be a list of attribute names to filter and order.
 39          __dict__ to be a dict, substitute of self.__dict__
 40          displayChildrenNames to be a bool.
 41          displaySequencesNames to be a bool.
 42          indent_prefix to be a string.
 43
 44      Features:
 45          - Use class name as tag name.
 46          - Use non-XMLRepr non-XMLRepr-containing-sequence attributes as
 47            attributes: names are used as names and values as values.
 48          - Use XMLRepr attributes as children, printed as it.
 49          - Use sequence attributes containing exclusively XMLRepr items as
 50            children: name is used as tag name and items as children.
 51          - If attributes is not given, order attributes and children by asc.
 52          - If displayChildrenNames is set True, children are preceded by their
 53            attr name. Ex: <brick>: <AwesomeBrick id=0 content='Red mushroom'/>
 54          - Filter attributes with the attribute names given by attributes parameter.
 55            Futhermore, it indicates the order of attributes.
 56          - Substitute self.__dict__ by __dict__.
 57          - If displaySequencesNames is set False, sequences' children are displayed
 58            without wrapping.
 59
 60      Example:
 61          class MyAwesomeClass(XMLRepr):
 62              def __init__(self):
 63                  self.color = 'pink'
 64                  self.checked = True
 65                  self.brick = AwesomeBrick(0)
 66                  self.bricks = [AwesomeBrick(1), AwesomeBrick(2)]
 67          class AwesomeBrick(XMLRepr):
 68              def __init__(self, id):
 69                  self.content = 'Red mushroom'
 70                  self.id = id
 71
 72          awesome_object = MyAwesomeClass()
 73          print(awesome_object)
 74
 75      Output:
 76          <MyAwesomeClass color='pink' checked=True>
 77              <AwesomeBrick id=0 content='Red mushroom'/>
 78              <bricks>
 79                  <AwesomeBrick id=1 content='Red mushroom'/>
 80                  <AwesomeBrick id=2 content='Red mushroom'/>
 81              </bricks>
 82          </MyAwesomeClass>
 83      """
 84
 85      def __repr__(self,
 86              attributes = None, __dict__ = None,
 87              displayChildrenNames = False, displaySequencesNames = True,
 88              indent_prefix = '  '
 89          ):
 90          if __dict__ is None:
 91              __dict__ = self.__dict__
 92          if attributes is None:
 93              attributes_and_children  = __dict__.items()
 94          else:
 95              attributes_and_children  = [(attr, __dict__[attr]) for attr in attributes]
 96          attributeList = []
 97          children = []
 98          sequences = []
 99          for name, value in attributes_and_children :
```

```
            if isinstance(value, XMLRepr):
                if displayChildrenNames:
                    children.append((name, value))
                else:
                    children.append(value)
            elif hasattr(value, '__iter__') and all(isinstance(item, XMLRepr) for item in value):
                sequences.append((name, value))
            else:
                attributeList.append((name, value))

        if attributes is None:
            attributeList.sort(key=itemgetter(0))
            if displayChildrenNames:
                children.sort(key=itemgetter(0))
            else:
                children.sort(key=attrgetter('__class__.__name__'))
            sequences.sort(key=itemgetter(0))

        def formatAttributes(attributeList):
            formatted_attributes = ''
            for name, value in attributeList:
                formatted_attributes += '{}={} '.format(name, repr(value))
            return formatted_attributes.rstrip(' ')

        def formatChildren(children):
            formatted_children = ''
            for value in children:
                formatted_children += '{}\n'.format(repr(value))
            return indent(formatted_children, indent_prefix)

        def formatChildrenWithNames(children):
            formatted_children = ''
            for name, value in children:
                formatted_children += '<{}>: {}\n'.format(name, repr(value))
            return indent(formatted_children, indent_prefix)

        def formatSequences(sequences):
            formatted_sequences = ''
            for name, seq in sequences:
                formatted_sequences += formatChildren(seq)
            return formatted_sequences

        def formatSequencesWithNames(sequences):
            formatted_sequences = ''
            for name, seq in sequences:
                formatted_sequences += '<{0}>\n{1}</{0}>\n'.format(name, formatChildren(seq))
            return indent(formatted_sequences, indent_prefix)

        if children or sequences:
            return '<{0} {1}>\n{2}{3}</{0}>'.format(
                self.__class__.__name__,
                formatAttributes(attributeList),
                formatChildrenWithNames(children) if displayChildrenNames \
                else formatChildren(children),
                formatSequencesWithNames(sequences) if displaySequencesNames \
                else formatSequences(sequences)
            )

        return '<{0} {1}/>'.format(
            self.__class__.__name__,
            formatAttributes(attributeList)
        )


if __name__ == '__main__':
    class MyAwesomeClass(XMLRepr):
        def __init__(self):
            self.color = 'pink'
            self.checked = True
            self.brick = AwesomeBrick(0)
            self.awesome = SuperAwesomeBrick(42)
            self.bricks = [AwesomeBrick(1), AwesomeBrick(2)]
    class AwesomeBrick(XMLRepr):
        def __init__(self, id):
            self.content = 'Red mushroom'
            self.id = id
    class SuperAwesomeBrick(AwesomeBrick):
        pass

    awesome_object = MyAwesomeClass()
    print(69*'-')
    print(awesome_object)

    class DisplayNamesAwesomeClass(MyAwesomeClass):
        def __repr__(self):
            return super().__repr__(displayChildrenNames=True, indent_prefix='    ')
    print(DisplayNamesAwesomeClass())

    class FilterAwesomeClass(MyAwesomeClass):
        def __repr__(self):
            return super().__repr__(attributes=['color', 'bricks'], indent_prefix='\t')
    print(FilterAwesomeClass())

    class SubstituteAwesomeClass(MyAwesomeClass):
        def __repr__(self):
            return super().__repr__(__dict__={'color': 'blood'}, indent_prefix='\t')
    print(SubstituteAwesomeClass())

    class WithoutSequencesNamesAwesomeClass(MyAwesomeClass):
        def __repr__(self):
            return super().__repr__(displaySequencesNames=False)
```

```
201         print(WithoutSequencesNamesAwesomeClass ())
202         print(69*'-')
```

```python
#!/usr/bin/env python3
# -*-coding:Utf-8 -*


# The MIT License (MIT)
#
# Copyright (c) 2016 Rémi Blaise <remi.blaise@gmx.fr> "http://php-zzortell.rhcloud.com/"
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.


def inject_arguments(in_function):
    """Inject arguments of a method as attributes

    To use as decorator.
    """

    def out_function(*args, **kwargs):
        _self = args[0]

        # Get all of argument's names of the in_function
        all_names = in_function.__code__.co_varnames[1:in_function.__code__.co_argcount]

        ## Add default values for non-specified arguments
        defaults = in_function.__defaults__
        if defaults:
            _self.__dict__.update(zip(all_names[-len(defaults):], defaults))

        ## Add kwargs
        _self.__dict__.update(kwargs)

        ## Add args
        # Get only the names that don't belong to kwargs
        names = [n for n in all_names if not n in kwargs]
        # Match argument names with values
        _self.__dict__.update(zip(names, args[1:]))

        return in_function(*args, **kwargs)

    return out_function


if __name__=='__main__':
    import unittest

    class ArgumentInjectionTest(unittest.TestCase):
        def test(self):
            class Test:
                @inject_arguments
                def __init__(self, name, surname, default = 'lol'):
                    pass

            t = Test('mickey', surname='mouse')
            self.assertEqual('mickey', t.name)
            self.assertEqual('mouse', t.surname)
            self.assertEqual('lol', t.default)

        def test_defaultAlone(self):
            class Test:
                @inject_arguments
                def __init__(self, default='lol'):
                    pass

            t = Test('given')
            self.assertEqual('given', t.default)

        def test_inheritance(self):
            class A():
                @inject_arguments
                def __init__(self, a1):
                    pass

            class B(A):
                @inject_arguments
                def __init__(self, b1 = None, b2 = None, *args, **kwargs):
                    super().__init__(*args, **kwargs)

            b = B(0, 1, 2)
            self.assertEqual(0, b.b1)
            self.assertEqual(1, b.b2)
            self.assertEqual(2, b.a1)

        def test_defaultInheritance(self):
```

```
            class Test:
                @inject_arguments
                def __init__(self, default='lol'):
                    pass

            class Child(Test):
                @inject_arguments
                def __init__(self, minus = None, malus = None, *args, **kwargs):
                    super().__init__(*args, **kwargs)

            c = Child(1, -1)
            self.assertEqual(1, c.minus)
            self.assertEqual(-1, c.malus)
            self.assertEqual('lol', c.default)

            c = Child(1, -1, 'hey')
            self.assertEqual(1, c.minus)
            self.assertEqual(-1, c.malus)
            self.assertEqual('hey', c.default)

        def test_giveLastDefaultArgument (self):
            class TestLastGivenDefault :
                @inject_arguments
                def __init__(self, default1=1, default2=2):
                    pass

            t = TestLastGivenDefault (default2=3)
            self.assertEqual(1, t.default1)
            self.assertEqual(3, t.default2)

    unittest.main()
```

```
"""
Inherit docstrings

Found here: http://code.activestate.com/recipes/578587-inherit-method-docstrings-without-breaking-decorat/

Simple Use:
    1) Import this module
    2) Inherit metaclass InheritableDocstrings
    3) Apply decorator inherit_docstring

Example:
    from lib.inherit_docstring import InheritableDocstrings, inherit_docstring

    class Animal:
        def move_to(self, dest):
            '''Move to *dest*'''
            pass

    class Bird(Animal, metaclass=InheritableDocstrings):
        @inherit_docstring
        def move_to(self, dest):
            self._fly_to(dest)

    assert Animal.move_to.__doc__ == Bird.move_to.__doc__

"""


from functools import partial

# Replace this with actual implementation from
# http://code.activestate.com/recipes/577748-calculate-the-mro-of-a-class/
# (though this will work for simple cases)
def mro(*bases):
    return bases[0].__mro__

# This definition is only used to assist static code analyzers
def inherit_docstring(fn):
    '''Copy docstring for method from superclass

    For this decorator to work, the class has to use the `InheritableDocstrings`
    metaclass.
    '''
    raise RuntimeError('Decorator can only be used in classes  '
                       'using the `InheritableDocstrings` metaclass ')

def _inherit_docstring(mro, fn):
    '''Decorator to set docstring for *fn* from *mro* '''

    if fn.__doc__ is not None:
        raise RuntimeError('Function already has docstring ')

    # Search for docstring in superclass
    for cls in mro:
        super_fn = getattr(cls, fn.__name__, None)
        if super_fn is None:
            continue
        fn.__doc__ = super_fn.__doc__
        break
    else:
        raise RuntimeError("Can't inherit docstring for %s: method does not "
                           "exist in superclass" % fn.__name__)

    return fn

class InheritableDocstrings(type):
    @classmethod
    def __prepare__(cls, name, bases, **kwds):
        classdict = super().__prepare__(name, bases, *kwds)

        # Inject decorators into class namespace
        classdict['inherit_docstring'] = partial(_inherit_docstring, mro(*bases))

        return classdict

    def __new__(cls, name, bases, classdict):

        # Decorator may not exist in class dict if the class (metaclass
        # instance) was constructed with an explicit call to `type`.
        # (cf http://bugs.python.org/issue18334)
        if 'inherit_docstring' in classdict:

            # Make sure that class definition hasn't messed with decorators
            copy_impl = getattr(classdict['inherit_docstring'], 'func', None)
            if copy_impl is not _inherit_docstring:
                raise RuntimeError('No inherit_docstring attribute may be created '
                                   'in classes using the InheritableDocstrings metaclass ')

            # Delete decorators from class namespace
            del classdict['inherit_docstring']

        return super().__new__(cls, name, bases, classdict)
```

```python
from math import floor
from random import gauss

def gauss_int(a, b):
    n = b + 1
    while n > b or n < a:
        n = floor(gauss(b, (b-a)))
    return n

if __name__ == '__main__':
    count = [0] * 39
    for i in range(1000000):
        count[gauss_int(0, 38)] += 1
    print(count)


--------------------------------------------------------------------------------

"""This is the standard Python 3.6 implementation of choices  """

from random import random
import itertools as _itertools
import bisect as _bisect

def choices(population, weights=None, *, cum_weights=None, k=1):
    """Return a k sized list of population elements chosen with replacement.

    If the relative weights or cumulative weights are not specified,
    the selections are made with equal probability.

    """
    if cum_weights is None:
        if weights is None:
            _int = int
            total = len(population)
            return [population[_int(random() * total)] for i in range(k)]
        cum_weights = list(_itertools.accumulate(weights))
    elif weights is not None:
        raise TypeError('Cannot specify both weights and cumulative weights  ')
    if len(cum_weights) != len(population):
        raise ValueError('The number of weights does not match the population  ')
    bisect = _bisect.bisect
    total = cum_weights[-1]
    return [population[bisect(cum_weights, random() * total)] for i in range(k)]
```