`galactic` developer guide

The Galactic Organization <contact@thegalactic.org>

0.0.8

Contents

1 Introduction **2**

2 Data readers **2**

3 Attributes **3**

4 Strategies **4**

4.1 Basic strategies 4

4.2 Meta strategies 5

5 Measures **5**

1 Introduction

py-galactic is a python package for studying formal concept analysis. [1]

This guide has been designed to allow developers to extend the *py-galactic-core* engine using 4 kinds of extensions:

- data reader plugins
- attribute plugins
- strategy plugins
- measure plugins

2 Data readers

Data readers are plugins of the *py-galactic-core* engine that allow to read new data file formats.

A data reader plugin must declare a class inheriting from the `galactic.concepts.DataReader` class and implementing the `read` method and defining the `extensions` property.

The `read` method is responsible for reading an already opened file and must return the data read either as a dictionary (if the individuals are named) or as a list.

[1] © 2018-2019 the Galactic Organization. This document is licensed under CC-by-nc-nd (https://creativecommons.org/licenses/by-nc-nd/4.0/deed.en)

The extensions property must return an iterator over all the file extensions supported by the plugin.

```
import io
from typing import Union, Iterable, Mapping

from galactic.concepts import DataReader

class MyDataReader(DataReader):
    @classmethod
    def read(cls, data_file: io.TextIOBase) -> Union[Iterable, Mapping]:
        # must return the data read from data_file

    @property
    def extensions(self):
        # must return an iterator over the supported file extensions
        # here: .my
        return iter([".my"])

def get_reader():
    return MyDataReader()
```

Finally, using the *setuptools* system, the package must declare an entry point in the setup.py file.

```
    entry_points={
        "py_galactic_data_reader": [
            "my_reader = my_data_reader:get_reader"
        ]
    },
```

3 Attributes

Attributes plugins are plugins of the *py-galactic-core* engine that allow to define new predicates and new description spaces on individuals.

An attribute plugin should declare a class inheriting from the

galactic.attributes.Predicate

class or preferably from one of its subclass and implementing the `__call__` method. This method must return True or False.

```python
from galactic.attribute import Predicate

class MyPredicate(Predicate):
    def __call__(self, individual=None):
        # Return True or False
```

Usually, an attribute plugin declares a class inheriting from the

`galactic.attributes.Description`

class and implementing the `__call__` method. This method must yield a set of minimal predicates describing a set of individuals.

```python
class MyDescription(Description):
    def __call__(
        self,
        individuals: Iterable = None
    ) -> Iterator[Predicate]:
        # yield instances of MyPredicate
```

4 Strategies

Strategies are plugins of the *py-galactic-core* engine that allow to create new predicates applicable to individuals.

There are two kinds of strategies:

- basic strategy
- meta strategy

4.1 Basic strategies

A basic strategy plugin must declare a class inheriting from the

`galactic.strategies.BasicStrategy`

class and implementing the `_selection` method. This method must produce an iterator over predicates defined by an attribute plugin.

The `__init__` method must set a value for the `_description` field.

```python
from typing import Iterable

from galactic.attributes import Attribute
from my_attribute_plugin import MyDescription, MyPredicate
from galactic.strategies import BasicStrategy


class MyStrategy(BasicStrategy):

    def __init__(self, attribute: Attribute):
        self._description = MyDescription(attribute)

    def _selection(self, individuals: Iterable = None):
        # yield instances of MyPredicate

def get_classes():
    return {"strategy.myplugin.MyStrategy": MyStrategy}
```

Finally, using the *setuptools* system, the package must declare an entry point in the `setup.py` file.

```
    entry_points={
        "py_galactic_extension": [
            "my_plugin = my_plugin:get_classes"
        ]
    },
```

4.2 Meta strategies

A meta strategy plugin must declare a class inheriting from the

`galactic.strategies.MetaStrategy`

class and implementing the `_selection` method. This method must produce an iterator over predicates defined by the inner strategies. There are two predefined meta-strategies in the core library:

- `galactic.strategies.LimitFilter`
- `galactic.strategies.SelectionFilter`

which use a measure to choose which predicates to keep. This is the best way to use meta-strategies.

5 Measures

Measures are plugins of the *py-galactic-core* engine that allow to use the predefined meta-strategies

- `galactic.strategies.LimitFilter`
- `galactic.strategies.SelectionFilter`

A measure plugin must declare a class inheriting from the

`galactic.strategies.Measure`

class and implementing the `__call__` method. This method must return a float value evaluating an attribute applied to a set of individuals.

```python
from typing import Iterable

from galactic.attributes import Attribute
from galactic.strategies import Measure


class MyMeasure(Measure):
    def __call__(
        self,
        individuals: Iterable = None,
        attribute: Attribute = None
    ):
        # must return a float value evaluating the attribute
        # applied on the individuals

def get_classes():
    return {"measure.myplugin.MyMeasure": MyMeasure}
```

Finally, using the *setuptools* system, the package must declare an entry point in the `setup.py` file.

```python
    entry_points={
        "py_galactic_extension": [
            "my_plugin = my_plugin:get_classes"
        ]
    },
```